

Spring Microservices IN ACTION

SECOND EDITION

John Carnell
Hillary Huaylupo Sánchez



MANNING



MEAP Edition
Manning Early Access Program
Spring Microservices in Action
Second Edition
Version 8

© Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/spring-microservices-in-action-second-edition/discussion>

Copyright 2020 Manning Publications

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

welcome

Thank you for purchasing the MEAP for Spring Microservices in Action. I hope you find this second edition useful for your professional development and enjoy reading this book as much as I enjoyed writing it.

As software developers, we sit in the middle of a sea of chaos and change. Sometimes, just when we think we have achieved the best solution for our problem, some new technologies and approaches appear on the scene, causing us to reevaluate how we deliver solutions for our customers. Microservices is a perfect example of this. When the microservices concept crept into the software development community around 2014, it made us realize that we did not have everything under control with the architectures implemented at that time, and that we had to migrate to something more flexible and innovative.

Microservices came as a distributed, loosely coupled software services solution that carries out a small number of well-defined tasks. With this book I'm going to teach you how to create a microservice solution so you can migrate from a non-microservices architecture to a microservices architecture or create an enterprise architecture from scratch using microservices. But why microservices with Spring? Spring is the de facto development framework for building Java-based applications, and the way it adapts perfectly with microservices is one of the main reasons I

chose to take on the project of writing the second edition of this book. In it, I'll show you step by step how to create a microservice and the entire architecture using Java, several Spring projects, and other projects you will find very useful.

When you finish reading this book, you will have learned a lot of concepts about microservice architectures, but the most important thing is that you will know how to implement all of those concepts seen in the book in your day-to-day life. You will be able to create a containerized enterprise microservices architecture from scratch using several projects to log, monitor, route, balance, and secure your services. Also, you will be able to deploy your microservices to the cloud using AWS as the cloud provider.

Thank you again and please let me know if you have any questions, comments, or suggestions in the [liveBook discussion forum](#). Your feedback is really important in developing the best book possible.

—Hillary Huaylupo Sánchez

brief contents

1 Welcome to the cloud, Spring

2 Exploring the microservices world with Spring Cloud

3 Building microservices with Spring Boot

4 Welcome to Docker

5 Controlling your configuration with Spring Cloud configuration server 6
On service discovery

6 On service discovery

7 When bad things happen: Resiliency patterns with Spring Cloud and Resilience4j

8 Service routing with Spring Cloud Gateway

9 Securing your microservices

10 Event-driven architecture with Spring Cloud Stream

11 Distributed tracing with Spring Cloud Sleuth and Zipkin

12 Deploying your microservices

APPENDIX

Appendix A: Microservice Architecture Best Practices

Appendix B: OAuth2 Grant Types

Appendix C: Monitoring your microservices

1 Welcome to the cloud, Spring

This chapter covers

- Understanding the difference between monolithic and microservices architectures
- Understanding microservices and why companies use them
- Using Spring, Spring Boot, and Spring cloud for building microservices
- Understanding the concept of cloud and the cloud-based computing models

Microservices are distributed, loosely coupled software services that carry out a small number of well-defined tasks. A distributed system is a system composed of several services that are separated and located on a network. These services communicate by passing messages between them. Implementing a microservice architecture is not an easy task; it comes with many challenges, such as application scalability, service discovery, monitoring, distributed tracing, security, management, and more. However, this book will teach you how to tackle all those challenges, introduce you to the world of microservices in Java, and show you the importance of applying these architectures to your business applications. You'll learn how to achieve this using

technologies such as Spring Cloud, Spring Boot, Swagger, Docker, Kubernetes, ELK (Elasticsearch, Logstash, and Kibana), Stack, Grafana, Prometheus, and more.

If you are a Java developer, this book will provide a smooth migration path from building traditional, monolithic Spring applications to microservice applications that can be deployed to the cloud. This book uses practical examples, diagrams, and descriptive texts to provide further details of how microservice architectures are implemented.

In the end, you will have learned how to implement technologies and techniques such as client-load balancing, dynamic scaling, distributed tracing, and more to create flexible, modern, and autonomous microservice-based business applications with Spring Boot and Spring Cloud. You will also be able to create your own build/deploy pipelines to achieve continuous delivery and integration in your business by applying technologies such as Kubernetes, Jenkins, and Docker.

1.1 The evolution towards a microservices architecture

Software architecture refers to all the fundamental parts that allow us to establish the structure, operation, and interaction between the components of the software. To create a good architecture, we can use design patterns as a guide; these patterns provide the necessary framework to guide the development of software, allowing developers to share the

same line of work and cover all objectives and application restrictions.

This book explains how to create a microservice architecture, but to better understand this, I will first explain what a monolithic architecture is and the differences between microservices and monolithic architectures. The idea is to give you a good starting point if you are still working with monolithic architectures. Figure 1.1 shows the comparison between a monolithic and a microservice architecture.

Concepts such as microservices architecture, monolith, cloud, and deployment can have many variations in their description and meaning. To avoid possible confusion, in this book, we are going to define the following concepts as:

- **Monolith.** As the word says, a monolith is composed all in one piece. This book refers to the monolith application as a backend that contains all of its components in the same application. This means that all of its components are interconnected, and it doesn't make any distinctions on the type of functionalities each service is delivering.
- **Microservice Architecture.** Architectural style in which an application is divided into several loosely coupled, highly maintainable, independent services that work together to deliver a complete software product. The main objective in this architecture is to separate services taking into consideration the business goals or functionalities.
- **Cloud.** The cloud refers to the internet. So, when we refer to the cloud, this means having all of the services we develop available on the internet.
- **Deployment.** Deployment refers to the process of running an application into a production server. In this process, we can highlight all the activities that allow us to make an application available for use.

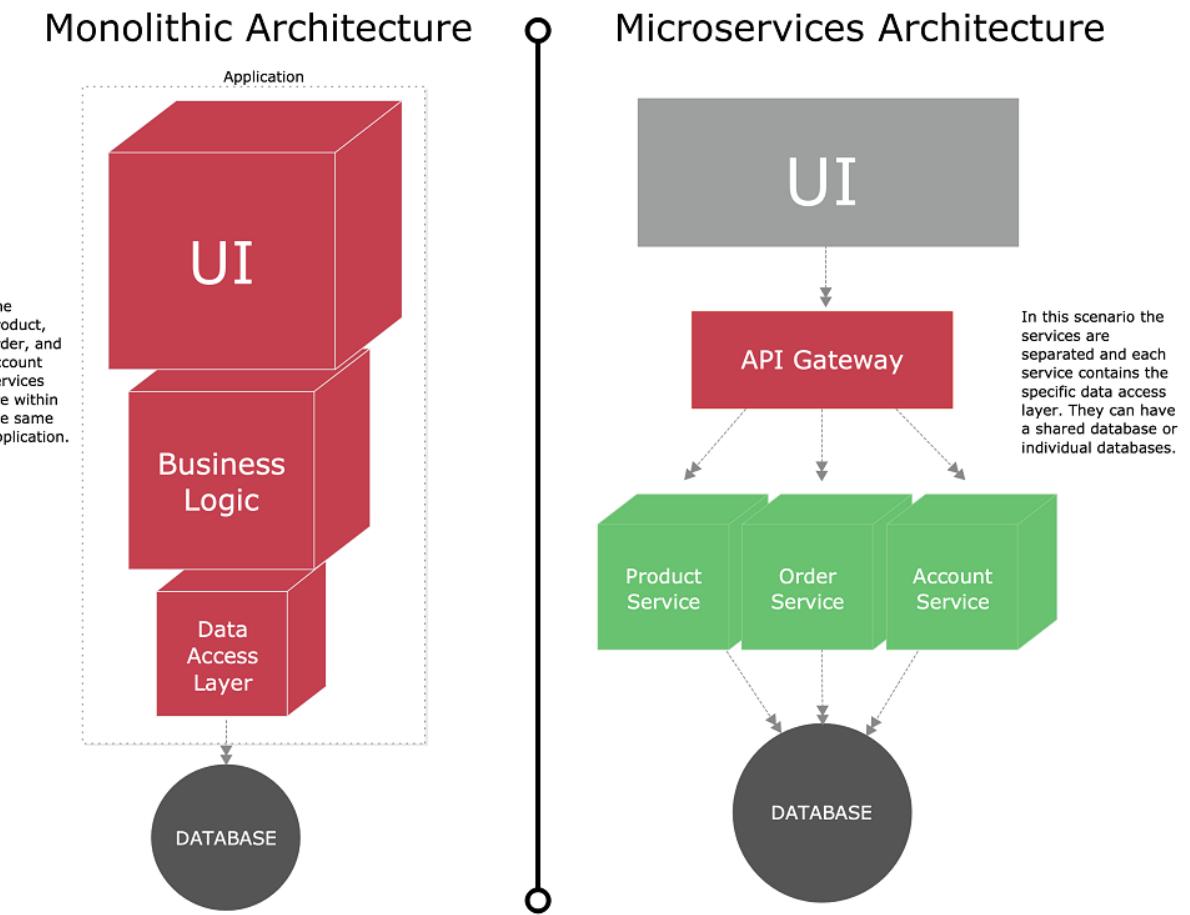


Figure 1.1 Comparison between a monolithic architecture and a microservices architecture.

1.1.1 What's a monolithic architecture?

Before the concept of microservices evolved, most web-based applications were built using a monolithic architectural style. In a monolithic architecture, an application is delivered as a single deployable software artifact. All the UI (user interface), business, and database access logic are packaged together into a unique application artifact and deployed to

an application server. Figure 1.2 shows the basic architecture of this application.

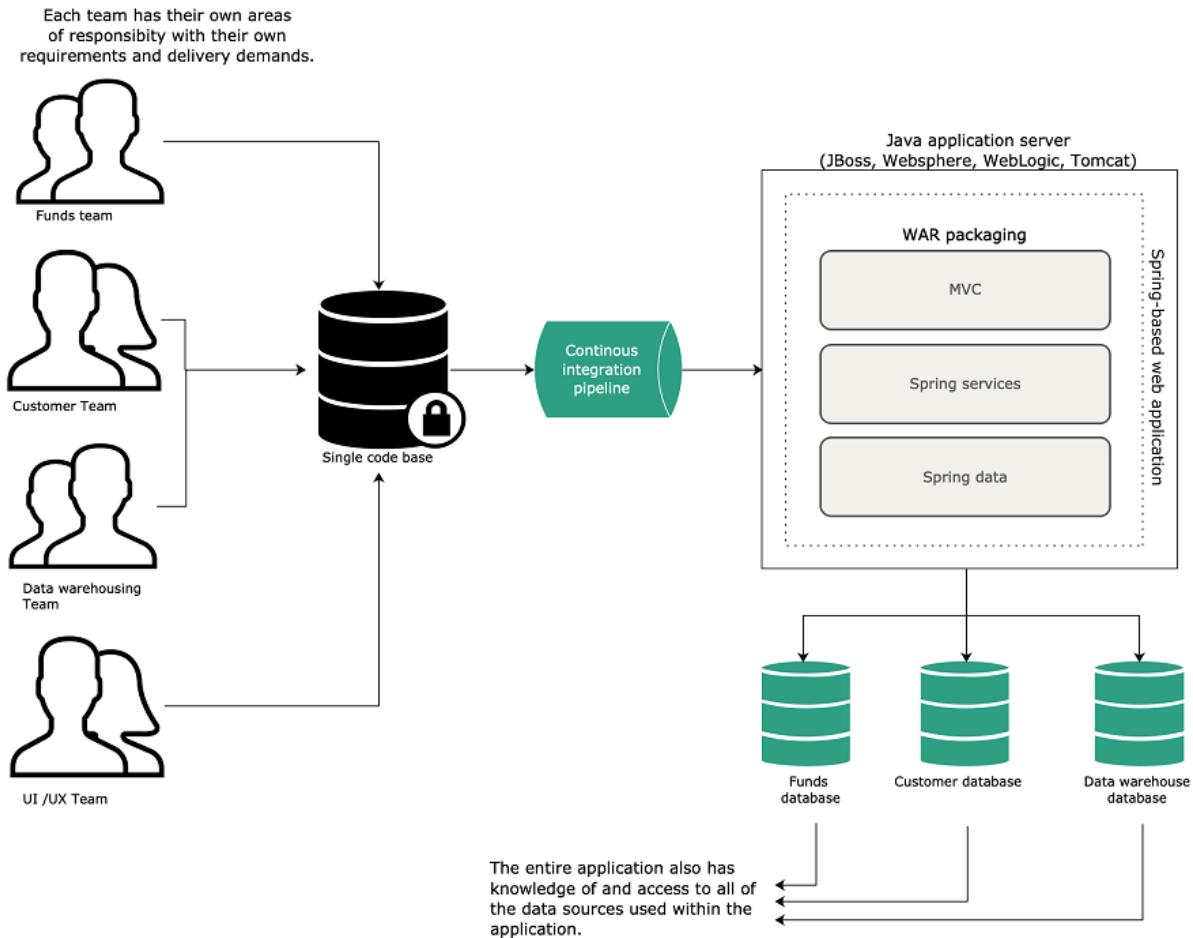


Figure 1.2 Monolithic applications force multiple development teams to artificially synchronize their delivery because their code needs to be built, tested, and deployed as an entire unit.

While an application might be deployed as a single unit of work, most of the time, there will be multiple development teams working on the application. Each development team will have their own discrete pieces of the application they're

responsible for, usually specific customers they're serving with their functional piece. For example, imagine a scenario where we have an in-house, custom-built customer relations management (CRM) application that involves the coordination of multiple teams including the UI/UX, the customer, the data warehouse and the funds team.

1.1.2 What's a microservice?

The concept of a microservice initially crept into the software development community's consciousness as a direct response to many of the challenges of trying to scale both technically and organizationally large, monolithic applications. Remember, a microservice is a small, loosely coupled, distributed service. Microservices allow you to take an extensive application and decompose it into easy-to-manage components with narrowly defined responsibilities. Microservices help combat the traditional problems of complexity in a large code base by decomposing the large code base down into small, well-defined pieces. The key concept you need to embrace as you think about microservices is decomposing and unbundling the functionality of your applications, so they're entirely independent of one another. If we take the CRM application, as shown in figure 1.2 and decompose it into microservices; it might look like the figure 1.3.

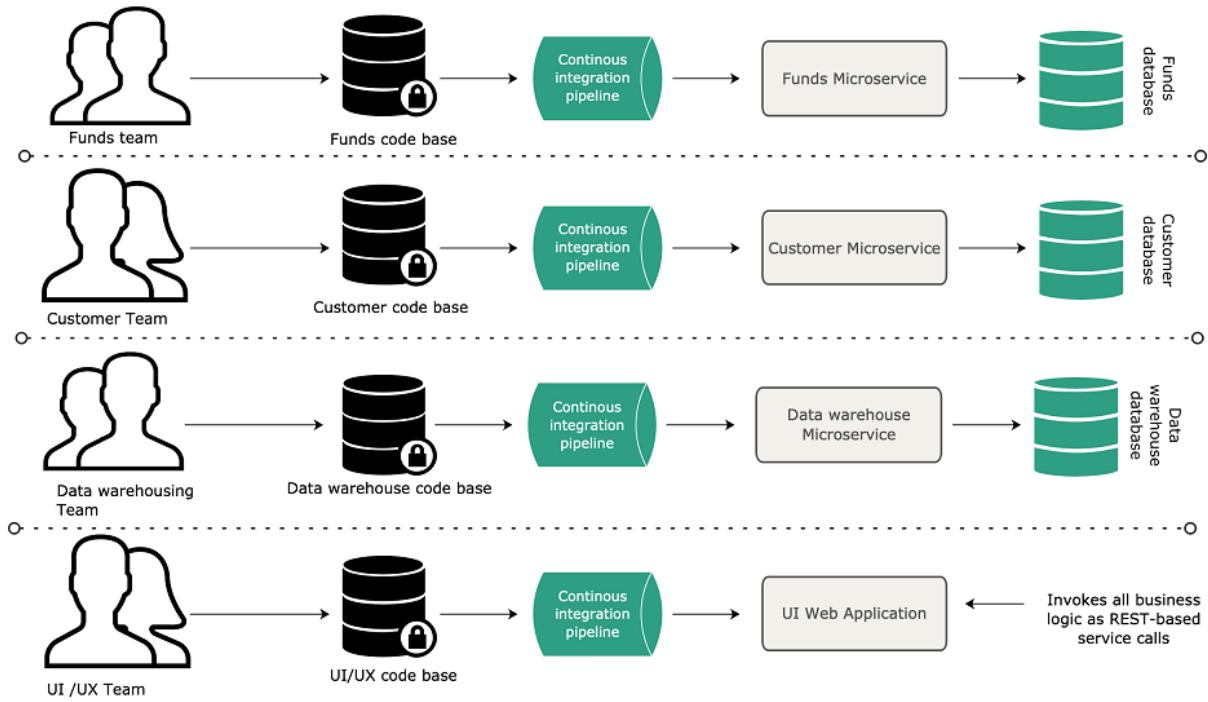


Figure 1.3 Using a microservice architecture the CRM application is decomposed into a set of microservices completely independent of each other, allowing each development team to move at their own pace.

Figure 1.3 shows how each functional team completely owns their service code and service infrastructure. They can build, deploy, and test independently of each other because their code, source control repository, and the infrastructure (app server and database) are now entirely independent of the different parts of the application.

A microservice architecture has the following characteristics:

- Application logic is broken down into small-grained components with well-defined boundaries of responsibility that coordinate to deliver a solution.

- Each component has a small domain of responsibility and is deployed completely independently of one another. Microservices should have responsibility for a **single part** of a business domain.
- Microservices communicate based on a few basic principles and employ lightweight communication protocols such as HTTP and JSON (JavaScript Object Notation) for exchanging data between the service consumer and service provider.
- The underlying technical implementation of the service is irrelevant because the applications always communicate with a technology-neutral format (JSON is the most common). This means an application built using a microservice approach could be built with multiple languages and technologies.
- Microservices—by their small, independent, and distributed nature—allow organizations to have small development teams with well-defined areas of responsibility. These teams might work toward a single goal such as delivering an application, but each team is responsible only for the services on which they're working.

1.1.3 Why change the way we build applications?

Companies that used to serve local markets are suddenly finding that they can reach out to a global customer base. However, with a broader global customer base also comes global competition. Having more competition impacts the way developers have to think about building applications. Some examples are:

- **Complexity has gone way up.** Customers expect that all parts of an organization know who they are. “Siloed”

applications that talk to a single database and don't integrate with other applications are no longer the norm. Today's applications need to communicate to multiple services and databases residing not only inside a company's data center, but also to external service providers over the internet.

- **Customers want faster delivery.** Customers no longer want to wait for the next annual release or version of a software package. Instead, they expect the features in a software product to be unbundled so that new functionality can be released quickly in weeks (even days).
- **Performance and scalability.** Global applications make it extremely difficult to predict how much transaction volume is going to be handled by an application, and when that transaction volume is going to hit. Applications need to scale up across multiple servers quickly and then scale back down when the volume needs have passed.
- **Customers expect their applications** to be available. Because customers are one click away from a competitor, a company's applications must be highly resilient. Failures or problems in one part of the application shouldn't bring down the entire application.

To meet these expectations, we, as application developers, have to embrace the mystery that to build high-scalable and highly redundant applications, we need to break our applications into small services that can be built and deployed independently of one another. If we "unbundle" our applications into small services and move them away from a single monolithic artifact, we can build systems that are:

- **Flexible.** Decoupled services can be composed and rearranged to quickly deliver new functionality. The smaller the unit of code that one is working with, the less complicated it is to change the code and the less time it takes to test deploy the code.

- **Resilient.** Decoupled services mean an application is no longer a single “ball of mud” where a degradation in one part of the application causes the whole application to fail. Failures can be localized to a small part of the application and contained before the entire application experiences an outage. This also enables the applications to degrade gracefully in case of an unrecoverable error.
- **Scalable.** Decoupled services can easily be distributed horizontally across multiple servers, making it possible to scale the features/services appropriately. With a monolithic application where all the logic for the application is intertwined, the entire application needs to scale even if only a small part of the application is the bottleneck. Scaling on small services is localized and much more cost-effective.

To this end, as we begin our discussion of microservices, keep the following in mind:

Small, Simple, and Decoupled Services = Scalable, Resilient, and Flexible Applications

It's important to understand that the systems and the organization itself can benefit from a microservices approach. To obtain benefits in the organization, we can apply Conway's law in reverse. This law indicates several points that can improve the communication and structure of a company. Conway's law (written in April 1968 by Melvin R. Conway in the article “How do Committees Invent”) states that "Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations." Basically, what it indicates is that the way teams communicate within the team and with other teams is directly reflected in the code they produce.

So, if we apply Conway's law in reverse (also known as inverse Conway maneuver) and design the company structure based on the microservice architecture, the communication, stability of the applications, and the organizational structure will improve by creating loosely coupled and autonomous teams as the microservices.

1.2 Microservices with Spring

Spring has become the most popular development framework for building Java-based applications. At its core, Spring is based on the concept of dependency injection.

A dependency injection framework allows you to more efficiently manage large Java projects by externalizing the relationship between objects within your application through convention (and annotations) rather than those objects having hard-coded knowledge about each other. Spring sits as an intermediary between the different Java classes of your application and manages their dependencies. Spring essentially lets you assemble your code like a set of Lego bricks that snap together.

What's impressive about the Spring framework and a testament to its development community is its ability to stay relevant and reinvent itself. The Spring development team quickly saw that many development teams were moving away from monolithic applications where the application's presentation, business, and data access logic were packaged together and deployed as a single artifact. Instead, teams were moving to highly distributed models where services

were being built as small, distributed services that could be quickly deployed to the cloud. In response to this shift, the Spring development team launched two projects: Spring Boot and Spring Cloud.

Spring Boot is a re-envisioning of the Spring framework. While it embraces core features of Spring, Spring Boot strips away many of the “enterprise” features found in Spring and instead delivers a framework geared toward Java-based, REST-oriented (Representational State Transfer) microservices. With a few simple annotations, a Java developer can quickly build a REST service that can be packaged and deployed without the need for an external application container.

NOTE While I cover REST in more detail in chapter 3, the core concept behind REST is that your services should embrace the use of HTTP verbs (GET, POST, PUT and DELETE) to represent the core actions of the service and use a lightweight web-oriented data serialization protocol, such as JSON, for requesting and receiving data from the service.

The key features of Spring Boot are:

- Embedded web server to avoid complexity in the application deployment: Tomcat (default), Jetty, or Undertow. Remember, this is one essential concept of Spring Boot; the chosen web server is part of the deployable JAR. For the spring boot applications, the only requisite to deploy the app is to have Java installed on the server.
- Suggested configuration to start quickly with a project (Starters)
- Automatic configuration for Spring functionally – whenever it’s possible
- Wide range of features ready for production, such as metrics, security, status verification, externalized

configuration, and more.

Using Spring Boot offers the following benefits for our microservices:

- Reduces development time and increased efficiency and productivity
- Offers an embedded HTTP server to run the web applications
- Allows you to avoid writing a lot of boilerplate
- Facilitates the integration with the Spring Ecosystem, which includes Spring Data, Spring Security, Spring Cloud, and more
- Provides a set of various development plugins that developers can work with

Because microservices have become one of the more common architectural patterns for building cloud-based applications, the Spring development community has given us Spring Cloud.

The Spring cloud framework makes it simple to operationalize and deploy microservices to a private or public cloud. Spring Cloud wraps several popular cloud-management microservice frameworks under a common framework and makes the use and deployment of these technologies as easy to use as annotating your code. I cover the different components within Spring Cloud in the next chapter.

1.3 What are we building?

This book offers a step-by-step guide on creating a complete microservices architecture using Spring Boot, Spring Cloud,

and other useful and modern technologies. Figure 1.X shows a high-level overview of some of the services and technology integrations that we will do throughout the book.

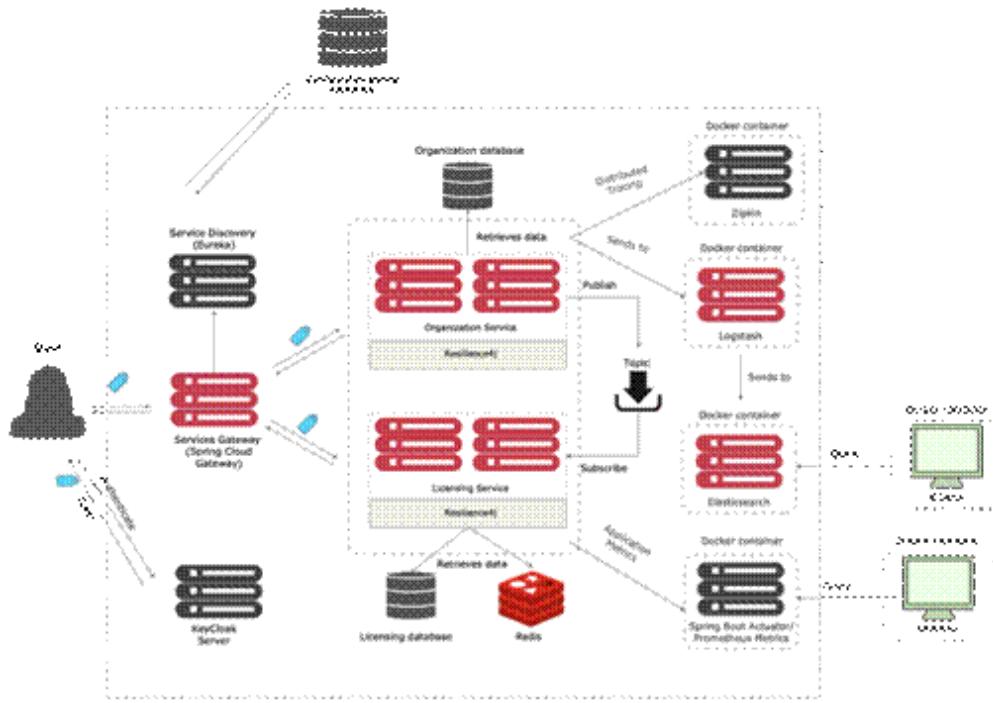


Figure 1.X High-level overview of the services and technologies that we are going to use and create throughout the entire book.

Figure 1.X describes a client request to update and retrieve the organization's information in the microservice architecture we will create. To start the request, the user first needs to authenticate with KeyCloak to get an access token. Once the token is obtained, the client makes a request to the Spring Cloud API Gateway. The API Gateway service is the entry point to our entire architecture; this service will communicate to the service discovery to retrieve the

locations of the organization and licensing services and then call the specific microservice.

Once the request arrives at the organization service, this service validates the access token against Keycloak to see if the token is valid and if the user has permission to continue the process. Once this is validated, the organization service will update and retrieve its information from the organization database and send it back to the client as an HTTP response. As an alternative path, once the organization information is updated, the organization service will add a message to the Kafka topic so the licensing service can be aware of the change.

Once the message arrives at the licensing service, this service will store the specific information in Redis's in-memory database. Throughout this process, the architecture will use distributed tracing from Zipkin, ElasticSearch, Logstash, and Zipkin to manage and display the logs and Spring Boot Actuator, Prometheus, and Grafana to expose and display the application metrics.

As we move forward, we will see topics such as Spring Boot, Spring Cloud, Elasticsearch, Logstash, Kibana, Prometheus, Grafana, and Kafka, among others. All these technologies may sound complicated, but we will see how to create and integrate the different components that make up the diagram as we progress through the book.

1.4 What is this book about?

The scope of this book is broad; it covers everything from basic definitions to more complex implementations to create a microservices architecture.

1.4.1 What you'll learn in this book

This book is about building microservice-based applications using a variety of Spring projects such as Spring Boot and Spring Cloud that can be deployed locally, in a private cloud run by your company or a public cloud such as Amazon, Google, or Azure. This book covers the following topics:

- What a microservice is, best practices, and design considerations that go into building a microservice-based application.
- When you shouldn't build a microservice-based application.
- How to build microservices using the Spring Boot framework.
- The core operational patterns that need to be in place to support microservice applications, particularly a cloud-base application.
- What is docker and how to integrate it with a microservice-based application.
- How you can use Spring Cloud to implement the operational patterns I will describe further in this chapter.
- How to create application metrics and visualize them in a monitoring tool.
- How to achieve a distributed tracing with Zipkin, Sleuth.
- How to manage application logs with ELK Stack.

- How to take what you've learned and build a deployment pipeline that can be used to deploy your services locally, to a private internally managed cloud or a public cloud provider.

By the time you're done reading this book, you should have the knowledge needed to build and deploy a Spring Boot-based microservice. You'll also understand the key design decisions needed to operationalize your microservices. You'll realize how service configuration management, service discovery, messaging, logging and tracing, security all fit together to deliver a robust microservices environment. Finally, you'll see how your microservices can be deployed using different technologies.

1.4.2 Why is this book relevant to you?

I suspect that if you have reached this point, it is because you:

- Are a Java Developer or have a strong grasp in Java
- Have a background in Spring.
- Are interested in learning how to build microservice-based applications.
- Are interested in how to use microservices to build cloud-based applications.
- Want to know if Java and Spring are relevant technologies for building microservice-based applications.
- Want to know what the cutting-edge technologies are to achieve a microservice architecture.
- Are interested in seeing what goes into deploying a microservice-based application to the cloud.

This book aims to offer a detailed guide on how to implement a microservices architecture in Java. It will provide descriptive and visual information and a lot of hands-on code examples to give a programmatic guide of how to implement such architecture using the latest versions of different Spring projects such as Spring Boot and Spring Cloud.

Additionally, this book aims to provide an introduction to the microservice patterns, best practices, infrastructure technologies that go hand by hand with this type of architectures to simulate a real-world application development environment.

Let's shift gears for a moment and walk through building a simple microservice using Spring Boot.

1.5 Cloud and microservice-based applications

In this section, we'll see how to create a microservice using Spring Boot and why the cloud is relevant to microservice-based applications.

1.5.1 Building a microservice with Spring Boot

This section will not provide a detailed walkthrough of much of the code on how to create the microservices but just a brief introduction on how to create a service. The main idea is to show how simple it is to create a service using Spring

Boot; for this, we're going to create a simple REST-service of "Hello World." With one main endpoint that uses the GET HTTP verb. This service endpoint will receive the parameters as request parameters, URL parameters (also known as path variables). Chapter 2 goes into much more detail.

This example is by no means exhaustive or even illustrative of how you should build a production-level microservice, but it should cause you to take a pause because of how little code it took to write it. We won't go through how to set up the project build files or the details of the code until chapter 2. If you'd like to see the Maven pom.xml file and the actual code, you can find it in the chapter 1 section of the downloadable code. All the source code for chapter 1 can be retrieved from the GitHub repository for the book at <https://github.com/ihuaylupo/manning-smia/tree/master/chapter1>

Figure 1.4 shows what the REST-service is going to do and the general flow of how Spring Boot microservice will process a user's request.

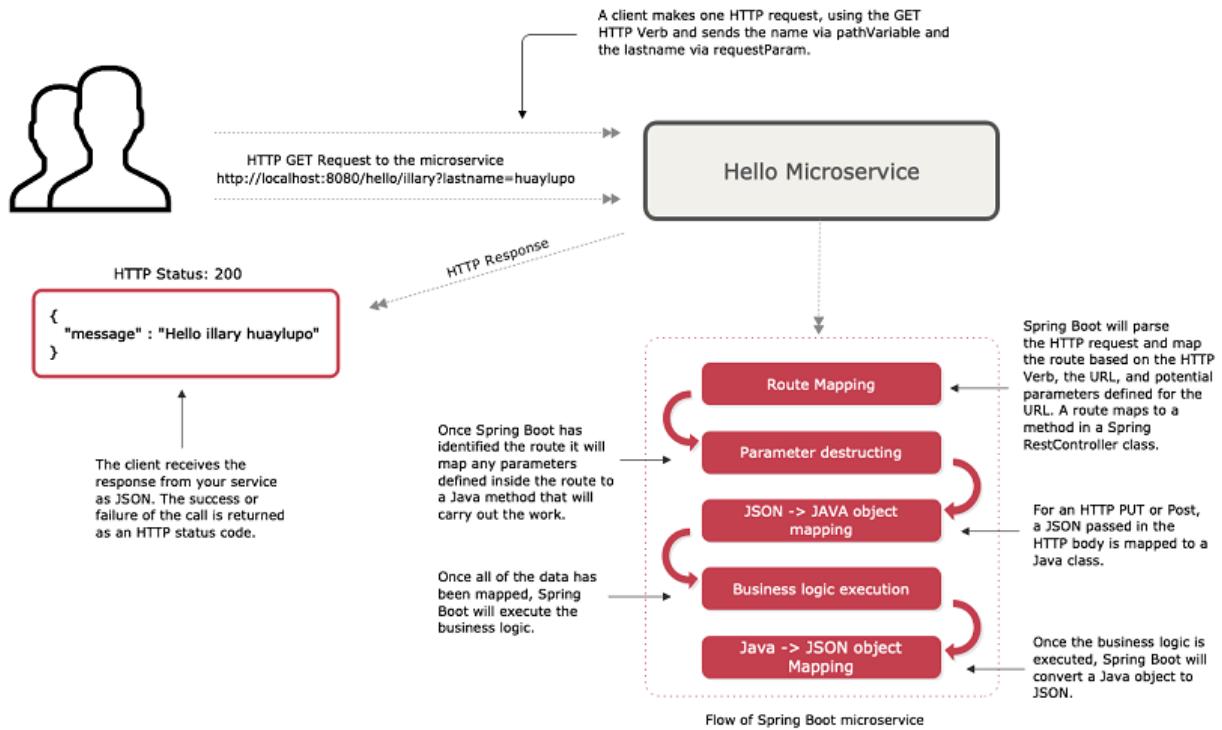


Figure 1.4 Spring Boot abstracts away the common REST microservice task (routing to business logic, parsing HTTP parameters from the URL, mapping JSON to/from Java Objects), and lets the developer focus on the business logic for the service. This figure shows three different ways to pass parameters to our controller.

For this example, you're going to have a single Java class called `com/huaylupo/spmia/ch01/SimpleApplication/Application.java` that will be used to expose a REST endpoint called `/hello`.

The following listing shows the code for `Application.java`.

Listing 1.1 Hello World with Spring Boot: a (very) simple Spring Microservice

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication #A
@RestController #B
@RequestMapping(value="hello") #C
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @GetMapping(value="/{firstName}") #D
    public String helloGET(
        @PathVariable("firstName") String firstName, #E
        @RequestParam("lastName") String lastName) { #E
        return String.format("{\"message\":\"Hello %s %s\"}", #F
            firstName, lastName);
    }
}

class HelloRequest{ #G

    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

#A Tells the Spring Boot framework that this class is the entry point for the Spring

Boot service.

#B Tells Spring Boot we are going to expose the code in this class as a Spring RestController class.

#C All URLs exposed in this application will be prefaced with /hello prefix.

#D Spring Boot will expose an endpoint as GET-based REST that will take two parameters in it firstName via @PathVariable and lastName via @RequestParam.

#E Maps the firstName and lastName parameters passed in to two variables passed into the hello function.

#F Returns a simple JSON string that we manually build. Later in Chapter 2, I will not be creating any JSON.

#G HelloRequest object containing the fields of the JSON structure sent by the user.

In listing 1.1, you're basically exposing one endpoint with a GET HTTP Verb that takes two parameters (firstName and lastName) on the URL, one from path variable and another one as request parameter; This endpoint returns a simple JSON string that has a payload containing the message "Hello firstName lastName". To call the GET endpoint /hello/illary?lastName=huaylupo on your service, the return of the call would be

```
{"message": "Hello illary huaylupo"}
```

Let's start the Spring Boot application. In order to do this, let's execute the following command on the command line.

```
mvn spring-boot:run
```

NOTE If you are running the command from the command line, make sure you are in the root directory. The root directory is the one that contains the pom.xml file. Otherwise, you will run into the No plugin found for prefix 'spring-boot' in the current project and in the plugin groups error.

The maven command, mvn, will use a Spring Boot plugin define in the pom.xml file to start the application using an embedded Tomcat server.

Java vs. Groovy and Maven vs. Gradle

The Spring Boot framework has strong support for both Java and the Groovy programming languages. Spring Boot also supports both Maven, and Gradle build tools. Gradle introduces a Groovy-based DSL (domain specific language) to declare the project configuration instead of an XML file like Maven. Although Gradle is a very powerful, flexible and top-rated tool, Maven is still very used by Java developers community. So, this book will only contain examples in Maven, to

keep the book manageable and the material focused, and it is intended to reach the largest audience possible.

Once the mvn spring-boot:run command is executed you'll see the following output on your command line.

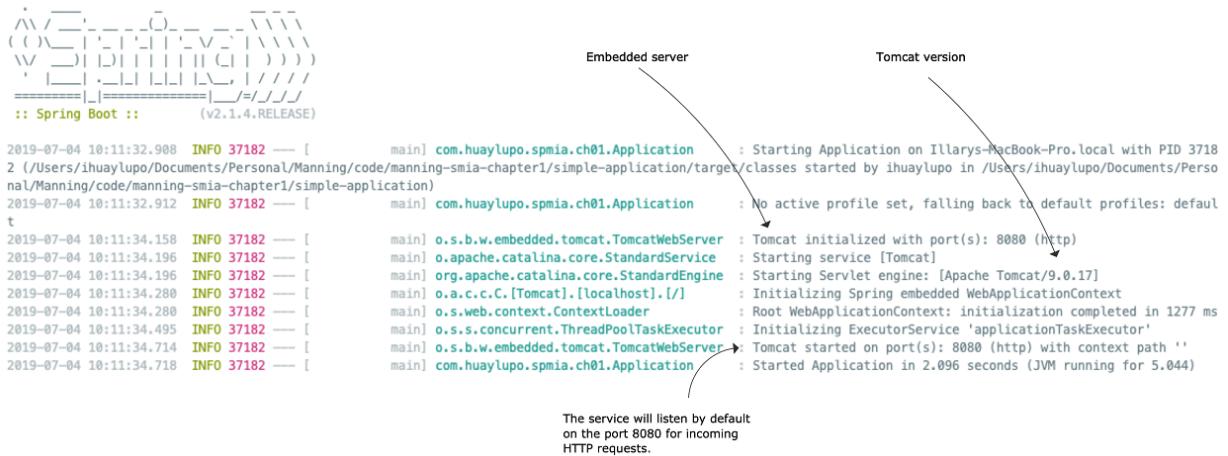


Figure 1.5 The Spring Boot service will communicate the port of the service via the console.

If everything starts correctly, you should see what's shown in figure 1.5 from your command-line window.

To execute the services you need to use a browser-based REST tool, there are many tools, both graphical and command line for invoking REST-based services, but for this book will be using POSTMAN (<https://www.getpostman.com/>).

Figure 1.6 and 1.7 shows two different POSTMAN calls to the endpoints with the results returned from the services.

HTTP GET for the /hello/illary?lastName=huaylupo

The screenshot shows the Postman interface with the following details:

- Request Method:** GET
- Request URL:** http://localhost:8080/hello/illary?lastName=huaylupo
- Params Tab:** Selected. Shows a table with one row:

	KEY	VALUE
<input checked="" type="checkbox"/>	lastName	huaylupo
	Key	Value
- Body Tab:** Selected. Shows a JSON response payload:


```

1 {
2   "message": "Hello illary huaylupo"
3 }
```
- Annotations:**
 - A black arrow points from the text "HTTP GET for the /hello/illary?lastName=huaylupo" to the request URL.
 - A black arrow points from the text "JSON payload returned back from the service" to the JSON response body.

Figure 1.6 The response from the GET /hello endpoint shows the data you've requested represented as a JSON payload.

The following figure 1.7 shows a brief example of how to make a call using the POST HTTP verb. It is essential to highlight that this is only for a demonstration purpose. In the following chapters, you'll see that the POST method is preferred to be used when it involves creating new records into our service.



Figure 1.7 The response from the POST /hello endpoint shows the request and the response data represented as a JSON payload.

This simple example code doesn't demonstrate the full power of Spring Boot neither the best practices to create a service. But what it shows is that you can write a full HTTP JSON Rest-based service with route-mapping of URL and parameters in Java with a few lines of code. Java, while being a powerful language, has acquired a reputation of being wordy compared to other languages, with Spring now we can accomplish a lot with just a few lines of code.

Next, let's walk through why and when a microservice approach is justified for building applications.

1.5.2 What exactly is cloud computing?

Cloud computing is the delivery of computing and virtualized IT services - databases, networking, software, servers, analytics, and more - through the Internet to provide a flexible, secure, and easy-to-use environment. Cloud computing offers significant advantages in the internal management of a company such as low initial investment, ease of use and maintenance, scalability, among others.

The cloud computing models allow the user to choose the level of control over the information and services that they will provide. They are known by their acronyms, being the generic representation of the same "**X** as a service"—an acronym that means anything as a service—.

The most common cloud computing models are:

- **Infrastructure as a Service (IaaS).** The vendor provides the infrastructure offering access to computing resources such as servers, storage, and network. In this model, the user is responsible for everything related to the maintenance of the infrastructure and the scalability of the application. E.g., AWS (EC2), Azure Virtual Machines, Google Compute Engine, Kubernetes.
- **Container as a Service (CaaS).** This cloud model is an intermediate point between the IaaS and the PaaS. It refers to a form of container-based virtualization in which enables developers to build and deploy their microservices as portable virtual containers (such as

Docker) to a cloud provider. Unlike an IaaS model, where you a developer have to manage the virtual machine the service is deployed to, with CaaS you're deploying your services in a lightweight virtual container. The cloud provider runs the virtual server the container is running on as well as the provider's comprehensive tools for building, deploying, monitoring, and scaling containers. E.g. Google Container Engine (GKE), Amazon's Elastic Container Service (ECS). In chapter 11 of this book, we'll see how to deploy the microservices you've built to Amazon ECS.

- **Platform as a Service (PaaS).** Model that provides a platform and an environment that allows users to focus on the development, execution, and maintenance of the application. These applications can be created even with tools that are provided by the vendor, for example, operating system, database management systems, technical support, storage, hosting, network, and more. With PaaS, users do not need to invest in physical infrastructure, nor spend time managing it, allowing them to concentrate exclusively on the development of applications. E.g. Google App Engine, CloudFoundry, Heroku, AWS (Beanstalk).
- **Function as a Service (FaaS).** This model is also known as serverless architecture. Despite the name, this architecture doesn't mean running a specific code without a server, what it does mean is a way of executing functionalities in the cloud in which the vendor provides all the required servers. Serverless architecture allows us to only focus on the development of services without having to worry about scaling, provisioning, and server administration. Instead, we must solely concentrate on uploading our functions without handling any administration infrastructure. FaaS platforms available: AWS (Lambda), Google Cloud Function, Azure functions.
- **Software as a Service (SaaS).** Also known as software on demand. This model offers users to use a specific

application without having to make any deployment, or to maintain it. In most cases, the access is through a web browser. In this model everything is managed by the service provider: application, data, operating system, virtualization, servers, storage, and network. The user just hires the service and uses the software. E.g. Salesforce, SAP, Google Business.

Figure 1.8 shows the differences between the cloud computing models.

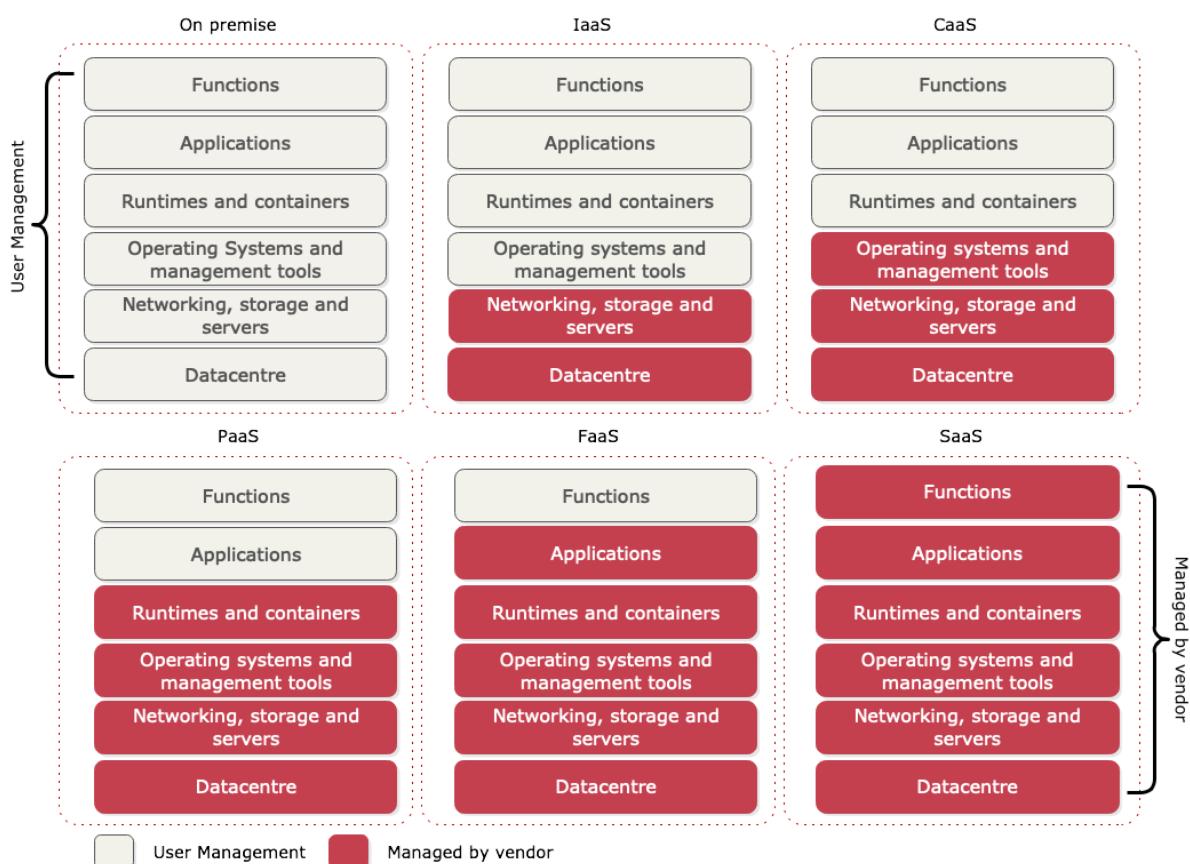


Figure 1.8 The different cloud computing models come down to who's responsible for what: User management or cloud vendor.

NOTE If you’re not careful, FaaS-based platforms can lock your code into a cloud vendor platform because your code is deployed to a vendor-specific runtime engine. With a FaaS-based model, you might be writing your service using a general programming language (Java, Python, JavaScript, and so on), but you’re still tying yourself heavily to the underlying vendor APIs and runtime engine that your function will be deployed to.

1.5.3 Why the cloud and microservices?

One of the core concepts of a microservice-based architecture is that each service is packaged and deployed as its own discrete and independent artifact. Service instances should be brought up quickly, and each instance of the service should be indistinguishable from another.

As a developer writing a microservice, sooner or later you’re going to have to decide whether your service is going to be deployed to one of the following:

- **Physical server.** While you can build and deploy your microservices to a physical machine(s), few organizations do this because physical servers are constrained. You can’t quickly ramp up the capacity of a physical server, and it can become extremely costly to scale your microservice horizontally across multiple physical servers.
- **Virtual machine images.** One of the key benefits of microservices is their ability to quickly start up and shut down microservice instances in response to scalability and service failure events. Virtual machines are the heart and soul of the major cloud providers.
- **Virtual container.** Virtual containers are a natural extension of deploying your microservices on a virtual machine image. Rather than deploying a service to a full

virtual machine, many developers deploy their services as Docker containers (or equivalent container technology) to the cloud. Virtual containers run inside a virtual machine; using a virtual container, you can segregate a single virtual machine into a series of self-contained processes that share the same virtual machine image. A microservice can be packaged up and multiple instances of the service can then be quickly deployed and started in either an IaaS private or public cloud.

The advantage of cloud-based microservices centers around the concept of elasticity. Cloud service providers allow you to quickly spin up new virtual machines and containers in a matter of minutes. If your capacity needs for your services drop, you can spin down containers to avoid additional costs. Using a cloud provider to deploy your microservices gives you significantly more horizontal scalability (adding more servers and service instances) for your applications. Server elasticity also means that your applications can be more resilient. If one of your microservices is having problems and is falling over, spinning up new service instances can keep your application alive long enough for your development team to gracefully resolve the issue.

For this book, all the microservices and corresponding service infrastructure will be deployed to a CaaS-based cloud provider using Docker containers. This is a common deployment topology used for microservices. The most common characteristics of CaaS cloud providers are:

- **Simplified infrastructure management.** CaaS cloud providers give you the ability to have the most control over your services. New services can be started and stopped with simple API calls.

- **Massive horizontal scalability.** CaaS cloud providers allow you to quickly and succinctly start one or more instances of a service. This capability means you can quickly scale services and route around misbehaving or failing servers.
 - **High redundancy through geographic distribution.** By necessity, CaaS providers have multiple data centers. By deploying your microservices using a CaaS cloud provider, you can gain a higher level of redundancy beyond using clusters in a data center.
-

Why not PaaS-based microservices?

Earlier in the chapter I discussed five types of cloud platforms (Infrastructure as a Service, Container as a Service, Platform as a service, Function as a Service, and Software as a Service). This book focuses specifically on building microservices using a CaaS approach. While certain cloud providers will let you abstract away the deployment infrastructure of your microservice, this book will teach you how to remain vendor-independent and deploy all parts of the application (including the servers).

For instance, Cloud Foundry, AWS (Beanstalk), Google App Engine, and Heroku give you the ability to deploy your services without having to know about the underlying application container. They provide a web interface and CLI to allow you to deploy your application as a WAR or JAR file. Setting up and tuning the application server and the corresponding Java container are abstracted away from you. While this is convenient, each cloud provider's platform has different idiosyncrasies related to its individual PaaS solution.

The services built in this book are packaged as Docker containers; the main reason is that Docker is deployable to all major cloud providers. In later chapters, we'll see what Docker is and how to integrate Docker to run all the services and infrastructure used in this book.

1.6 Microservices are more than writing the code

While the concepts around building individual microservices are easy to understand, running and supporting a robust microservice application (especially when running in the cloud) involves more than writing the code for the service. Figure 1.9 shows some guides to take in consideration while writing/building a microservice.

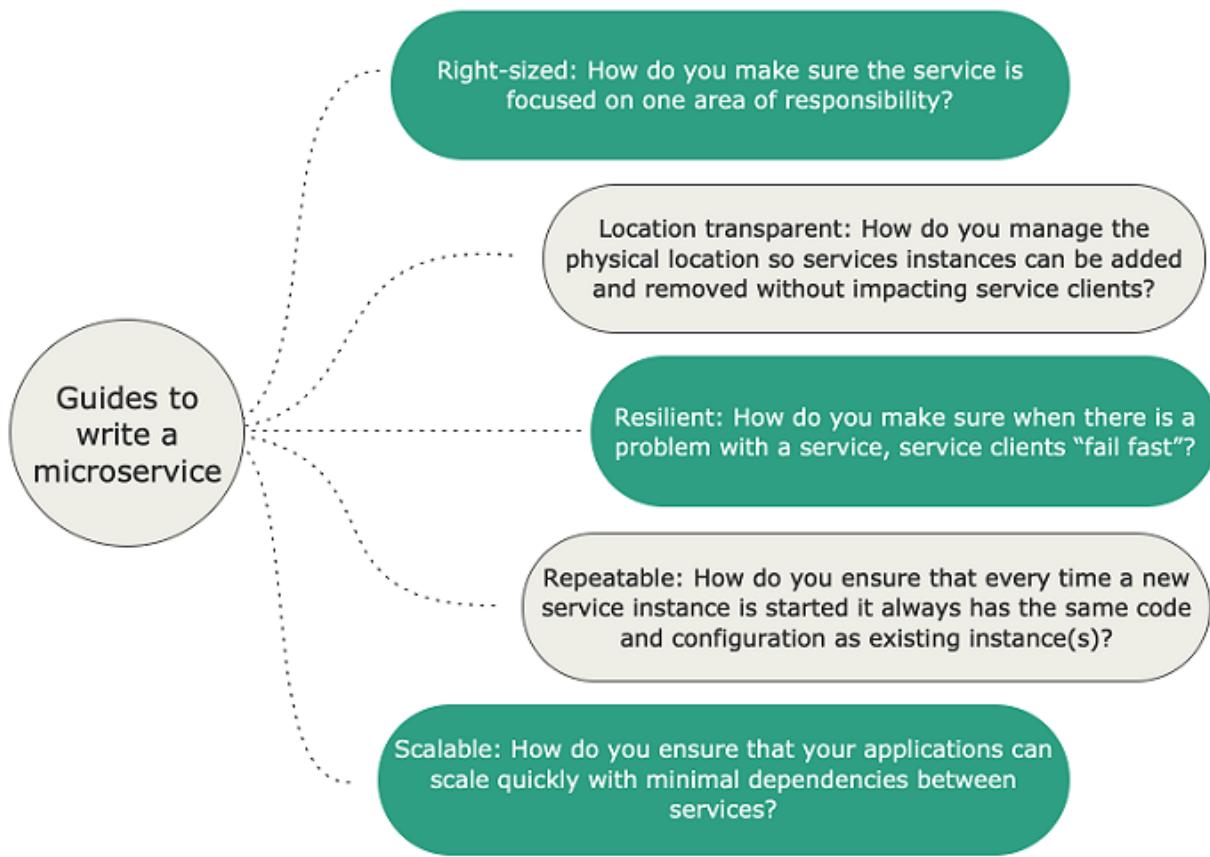


Figure 1.9 Microservices are more than the business logic. You need to think about the environment where

the services are going to run and how the services will scale and be resilient.

Writing a robust service includes considering several topics. Let's walk through the items shown in figure 1.9 in more detail:

- **Right-sized.** How do you ensure that your microservices are properly sized so that you don't have a microservice take on too much responsibility? Remember, properly sized, a service allows you to make changes to an application quickly and reduces the overall risk of an outage to the entire application.
- **Location transparent.** How do you manage the physical details of service invocation when in a microservice application, multiple service instances can quickly start and shut down?
- **Resilient.** How do you protect your microservice consumers and the overall integrity of your application by routing around failing services and ensuring that you take a "fail-fast" approach?
- **Repeatable.** How do you ensure that every new instance of your service brought up is guaranteed to have the same configuration and code base as all the other service instances in production?
- **Scalable.** How do you establish a communication that minimizes the direct dependencies between your services and ensure that you can gracefully scale your microservices?

This book takes a patterns-based approach as we answer these questions. With a patterns-based approach, we'll look at common designs that can be used across different technology implementations. While we've chosen to use Spring Boot and Spring Cloud to implement the patterns we're going to use in this book, nothing will keep you from

taking the concepts presented here and using them with other technology platforms. Specifically, we'll cover the following microservice patterns:

- Core development pattern
- Routing patterns
- Client resiliency patterns
- Security patterns
- Logging and tracing patterns
- Application metrics patterns
- Build and deployment pattern

It's important to understand that there isn't a formal definition of how to create a microservice, but in the next section, we'll see a list of common aspects you have to take into consideration while building a microservice.

1.7 Core microservice development pattern

The core development microservice development pattern addresses the basics of building a microservice. Figure 1.10 highlights the topics we'll cover around basic service design.

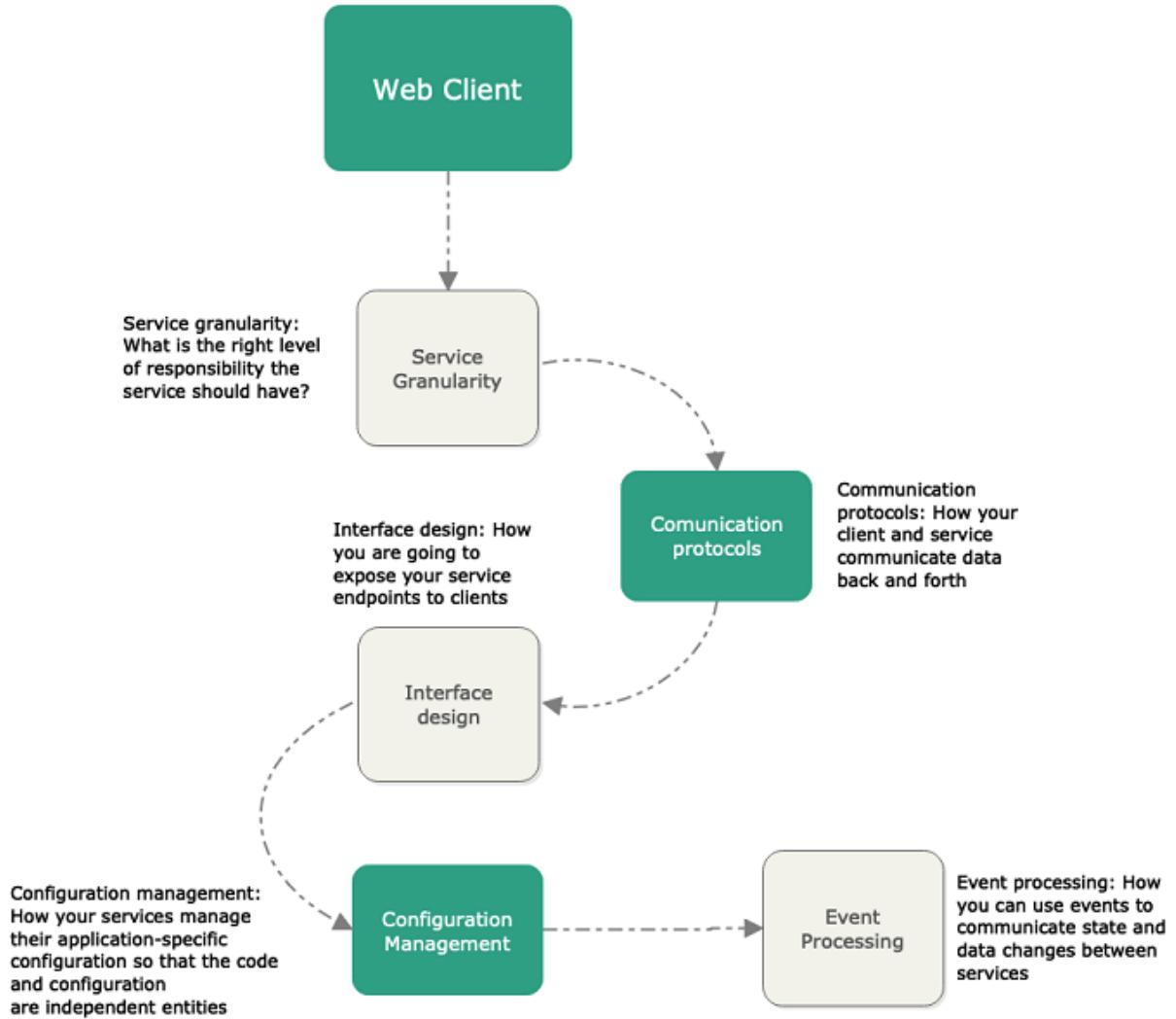


Figure 1.10 When designing your microservice, you have to think about how the service will be consumed and communicated with.

The following patterns show the basics of building a microservice:

- **Service granularity.** How do you approach decomposing a business domain down into microservices so that each microservice has the right level of responsibility? Making a service too coarse-grained with responsibilities that

overlap into different business problems domains makes the service difficult to maintain and change over time. Making the service too fine-grained increases the overall complexity of the application and turns the service into a “dumb” data abstraction layer with no logic except for that needed to access the data store. Service granularity is covered in chapter 3.

- **Communication protocols.** How will developers communicate with your service? The first step is to define whether we want a synchronous or asynchronous protocol. For synchronous the most common communication is HTTP-based REST using XML (Extensible Markup Language), JSON (JavaScript Object Notation), or binary protocol such as Thrift to send data back and forth your microservices. For asynchronous, the most popular protocol is AMQP (Advanced Message Queuing Protocol) and it can be implemented one-to-one (queue) or one-to-many(topic) with message brokers such as RabbitMQ, Apache Kafka and SQS. In later chapters, we'll learn about the communication protocols.
- **Interface design.** What's the best way to design the actual service interfaces that developers are going to use to call your service? How do you structure your services? What are the best practices? Best practices and interface design are covered in the next chapters.
- **Configuration management of service.** How do you manage the configuration of your microservice so that it moves between different environments in the cloud? This can be managed with externalized configuration and profiles seen in chapter 5.
- **Event processing between services.** How do you decouple your microservice using events so that you minimize hardcoded dependencies between your services and increase the resiliency of your application? Using an event-driven architecture with Spring Cloud Stream, this is covered in chapter 10.

1.8 Microservice routing patterns

The microservice routing patterns deal with how a client application that wants to consume a microservice discovers the location of the service and is routed over to it. In a cloud-based application, it is possible to have hundreds of microservices instances running. To enforce security and content policies is required to abstract the physical IP address of those services and have a single point of entry for the service calls. How? The following patterns are going to answer that question.

- **Service discovery.** With a service discovery and its key feature service registry you can make your microservice discoverable so client applications can find them without having the location of the service hardcoded into their application. How? I explain this in chapter 6. Remember the service discovery is an internal service, not a client facing service.

NOTE In this book, I use Netflix Eureka Service Discovery, but there are other service registries such as etcd, Consul, and Apache Zookeeper. Also, some systems do not have an explicit service registry. Instead they used an inter-service communication infrastructure known as a service mesh. Service Mesh is covered in chapter 13.

- **Service routing.** With an API Gateway, you can provide a single entry point for all of your services so that security policies and routing rules are applied uniformly to multiple services and service instances in your microservices applications. How? I explain this in chapter 8 with Spring Cloud API Gateway.

Figure 1.11 shows how service discovery and service routing appear to have a hard-coded sequence of events between them (first comes service routing and then the service discovery). However, the two patterns aren't dependent on one another. For instance, we can implement service discovery without service routing, and you can implement service routing without service discovery (even though its implementation is more difficult).

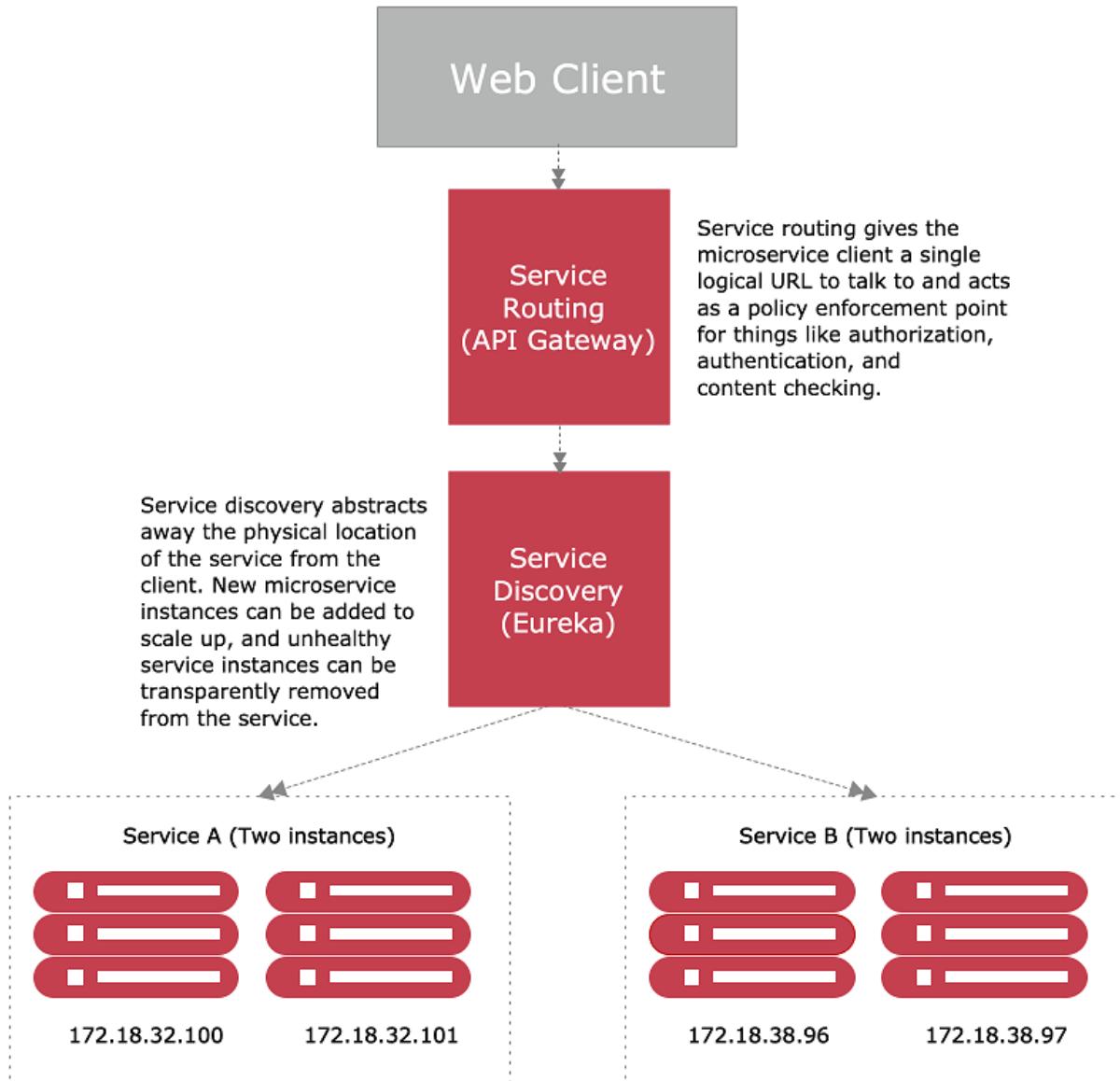


Figure 1.11 Service discovery and routing are key parts of any large-scale microservice application.

1.9 Microservice client resiliency

Because microservice architectures are highly distributed, you have to be extremely sensitive in how you prevent a problem in a single service (or service instance) from cascading up and out to the consumers of the service. To this end, we'll cover four client resiliency patterns:

- **Client-side load balancing.** How do you cache the location of your service instances on the service so that calls to multiple instances of a microservice are load balanced to all the healthy instances of that microservice?
- **Circuit breakers pattern.** How do you prevent a client from continuing to call a service that's failing or suffering performance problems? When a service is running slowly, it consumes resources on the client calling it. You want failing microservice calls to fail fast so that the calling client can quickly respond and take an appropriate action.
- **Fallback pattern.** When a service call fails, how do you provide a "plug-in" mechanism that will allow the service client to try to carry out its work through alternative means other than the microservice being called?
- **Bulkhead pattern.** Microservice applications use multiple distributed resources to carry out their work. How do you compartmentalize these calls so that the misbehavior of one service call doesn't negatively impact the rest of the application?

Figure 1.12 shows how these patterns protect the consumer of service from being impacted when a service is misbehaving. These topics are covered in chapter 7.

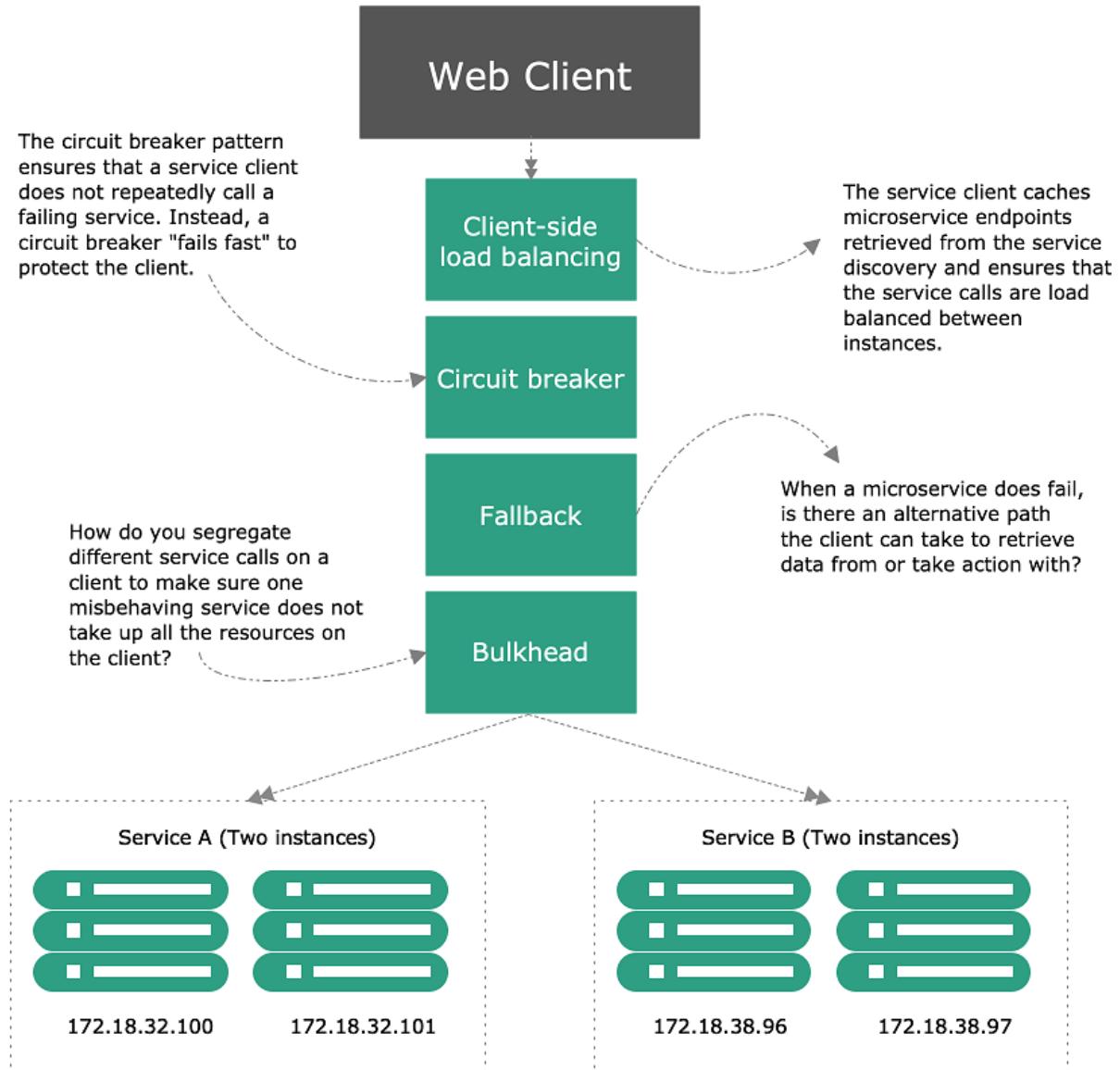


Figure 1.12 With microservices, you must protect the service caller from a poorly behaving service. Remember, a slow or down service can cause disruptions beyond the immediate service.

1.10 Microservice security patterns

To ensure that the microservices are not open to the public, it is important to apply the following security patterns to the architecture. In order to ensure that only granted requests with proper credentials can invoke the services. These patterns are:

- **Authentication.** How do you determine the service client calling the service is who they say they are?
- **Authorization.** How do you determine whether the service client calling a microservice is allowed to undertake the action they're trying to undertake?
- **Credential management and propagation.** How do you prevent a service client from constantly having to present their credentials for service calls involved in a transaction? To achieve this in this book, we'll look at how token-based security standards such as OAuth2 and JSON Web Tokens (JWT) can be used to obtain a token that can be passed from service call to service call to authenticate and authorize the user.

Figure 1.13 shows how you can implement those three patterns described previously to build an authentication service that can protect your microservices.

What is OAuth 2.0?

OAuth2 is a token-based security framework that allows a user to authenticate themselves with a third-party authentication service. If the user successfully authenticates, they will be presented a token that must be sent with every request. The main goal behind OAuth2 is that when multiple services are called

to fulfill a user's request, the user can be authenticated by each service without having to present their credentials to each service processing their request.

While OAuth is covered in chapter 9, it's worthwhile to read the OAuth 2.0 documentation by Aaron Parecki (<https://www.oauth.com/>).

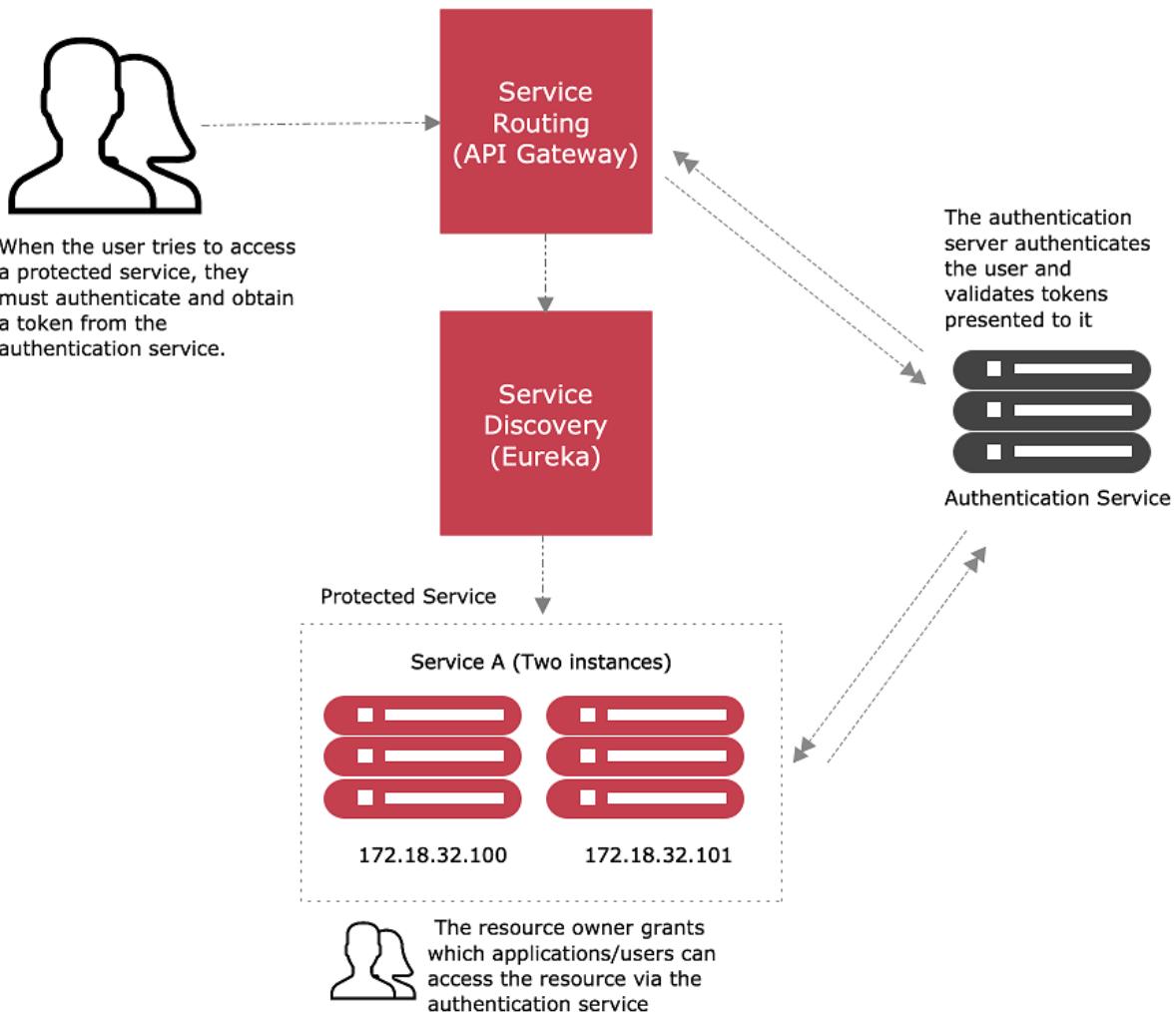


Figure 1.13 Using a token-based security scheme, you can implement service authentication and authorization without passing around client credentials.

1.11 Microservice logging and tracing patterns

The downside of a microservice architecture is that it's much more difficult to debug, trace and monitor the issues, because one simple action can trigger a bunch of microservices calls within your application. Further chapters will cover how to implement a distributed tracing with Spring Cloud Sleuth, Zipkin and the ELK stack. For this reason, we'll look at the following three core logging and tracing patterns to achieve a distributed tracing.

- **Log correlation.** How do you tie together all the logs produced between services for a single user transaction? With this pattern, we'll look at how to implement a correlation ID, which is a unique identifier that will be carried across all service calls in a transaction and can be used to tie together log entries produced from each service.
- **Log aggregation.** With this pattern we'll look at how to pull together all of the logs produced by your microservices (and their individual instances) into a single queryable database across all the services involved and understand the performance characteristics of services involved in the transaction.
- **Microservice tracing.** Finally, we'll explore how to visualize the flow of a client transaction across all the services involved and understand the performance characteristics of services involved in the transaction.

Figure 1.14 shows how these patterns fit together. We'll cover the logging and tracing patterns in greater detail in chapter 11.

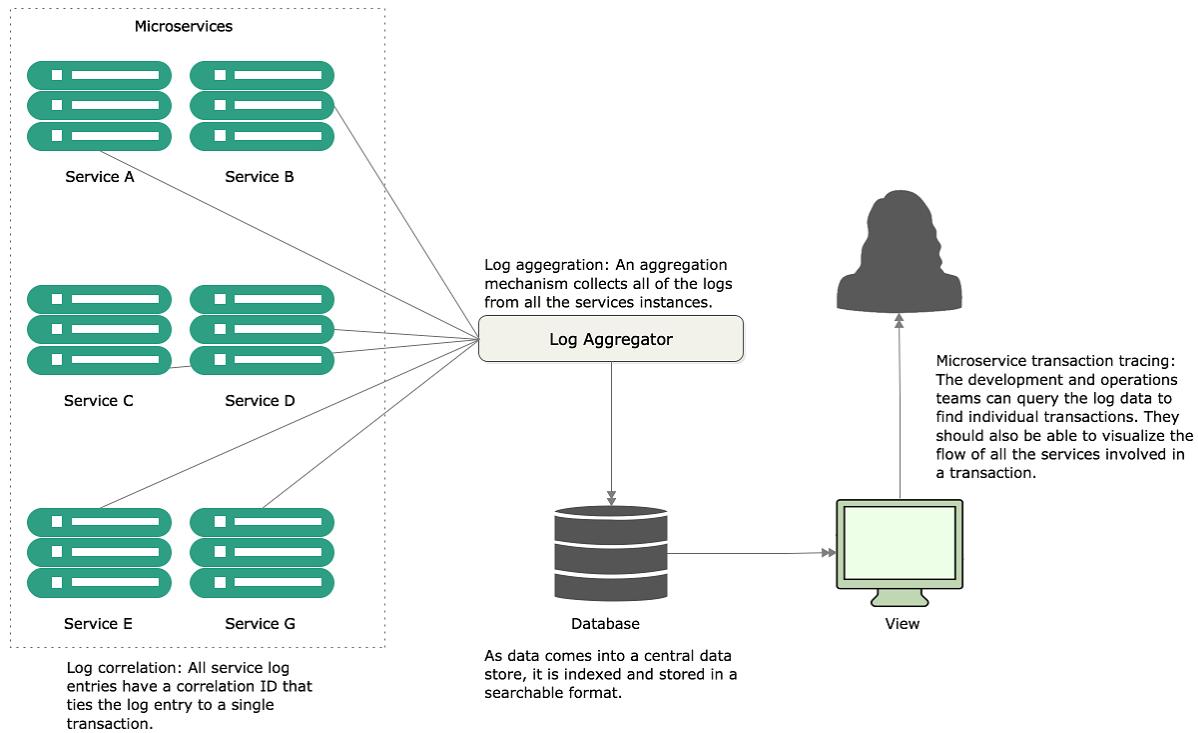


Figure 1.14 A well-thought-out logging and tracing strategy makes debugging transactions across multiple services manageable.

1.12 Application metrics pattern

The application metrics pattern deals with how the application it's going to monitor metrics and alert possible causes of failure within our application. This pattern shows, how the metrics service is responsible for getting (scraping), storing, and querying business-related data in order to prevent potential performance issues in our services.

The pattern contains the following three main components.

- **Metrics.** How can you create critical information about the health of your application? How can we expose those metrics?
- **Metrics service.** Where can we store and query the application metrics?
- **Metrics visualization suite.** Where can we visualize time business-related data for the application and the infrastructure?

Figure 1.15 shows how the metrics generated by the microservices are highly dependent to the metrics service and the visualization suite; it would be useless to have metrics that generate and show infinite information if there is no way to understand and analyze that information. The metrics service can obtain the metrics using the pull or push style. With the push style, the service instance invokes a service API exposed by the metrics services in order to send the application data, with the pull style the metrics service asks or queries a function to fetch the application data.

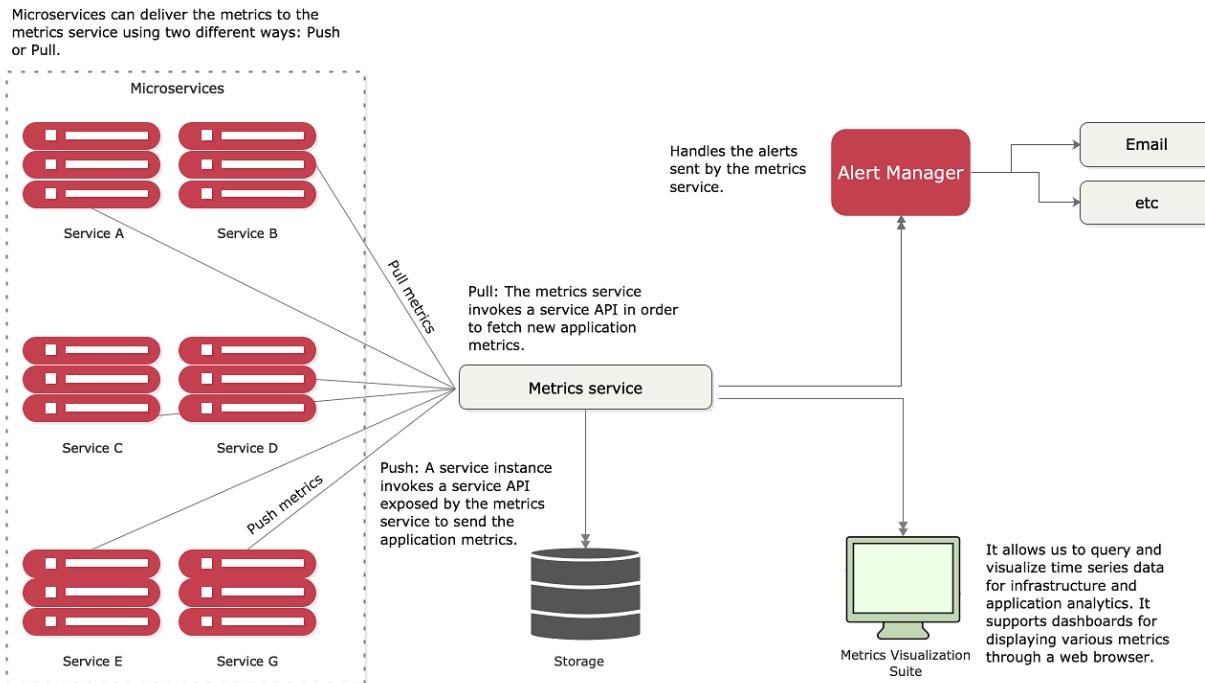


Figure 1.15 Metrics are pulled or pushed from the microservices and are collected and stored in the metrics service, to be shown and notified using a metrics visualization suite and an alert management tool.

It's important to understand that monitoring metrics is an essential aspect of the microservices architectures and that the monitoring requirements in this kind of architectures tend to be higher than monolithic structures due to their high distribution.

1.13 Microservice build/deployment patterns

One of the core parts of a microservice architecture is that each instance of a microservice should be identical to all its other instances. You can't allow "configuration drift" (something changes on a server after it's been deployed) to occur, because this can introduce instability in your applications.

The goal with this pattern is to integrate the configuration of your infrastructure right into your build-deployment process so that you no longer deploy software artifacts such as Java WAR or EAR to an already-running piece of infrastructure. Instead, you want to build and compile your microservice and the virtual server image it's running on as part of the build process. Then, when your microservice gets deployed, the entire machine image with the server running on it gets deployed.

Figure 1.16 illustrates this process. At the end of the book we'll look at how to create your build and deployment pipeline. In chapter 12 I cover the following patterns and topics:

- **Build and deployment pipeline.** How do you create a repeatable build and deployment process that emphasizes one-button builds and deployment to any environment in your organization?
- **Infrastructure as code.** How do you treat the provisioning of your services as code that can be executed and managed under source control?
- **Immutable servers.** Once a microservice image is created, how do you ensure that it's never changed after it has been deployed?
- **Phoenix servers.** The longer a server is running, the more opportunity for configuration drift. How do you ensure that servers that run individual containers get

torn down on a regular basis and recreated off an immutable image? A configuration drift could occur when ad hoc changes to a system configuration are unrecorded.

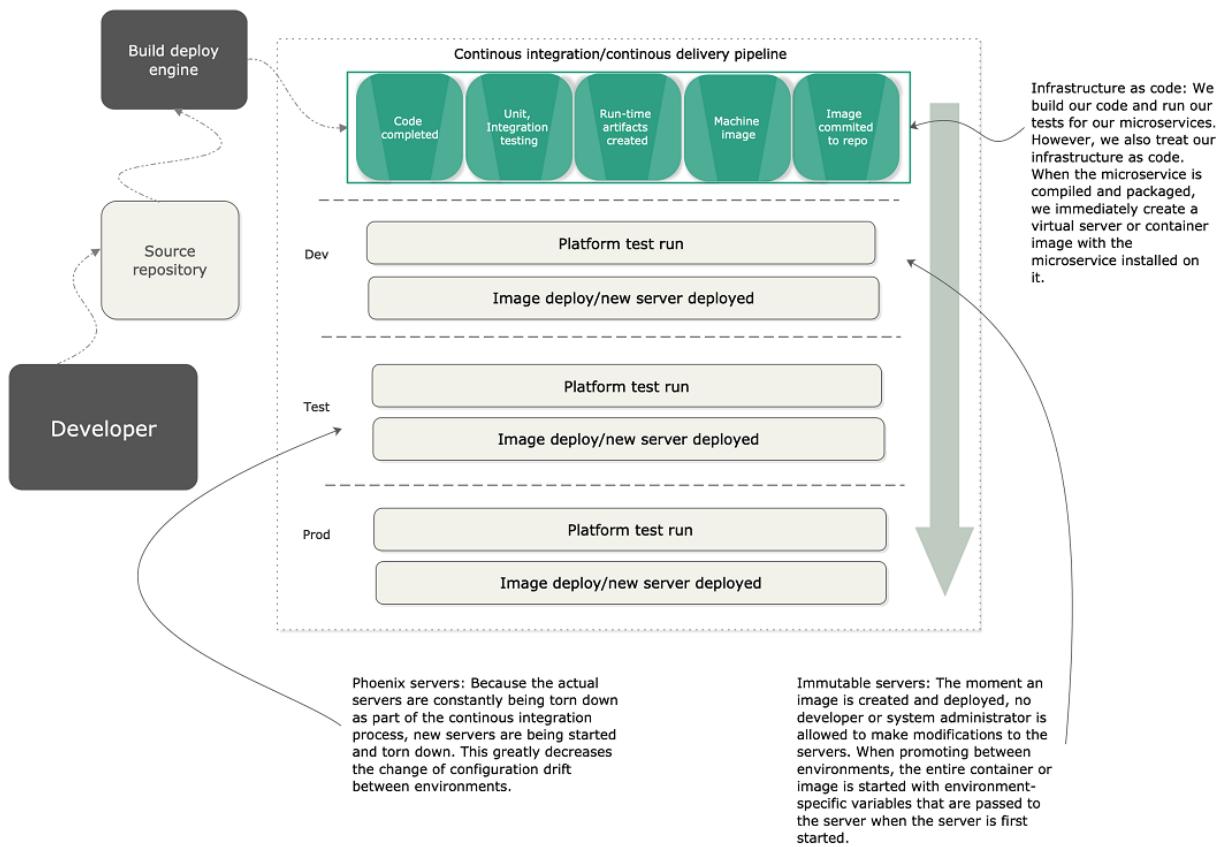


Figure 1.16 You want the deployment of the microservice and the server it's running on to be one atomic artifact that's deployed as a whole between environments.

Our goal with these patterns and topics is to ruthlessly expose and stamp out configuration drift as quickly as possible before it can hit your upper environments, such as stage or production.

NOTE For the code examples in this book (except chapter 12), everything will run locally on your desktop machine. The first chapters can be run natively directly from the command line. Starting in chapter 3, all the code will be compiled and run as Docker containers.

Now that we've covered the patterns that we will be using throughout the book, let's continue with the second chapter. In the next chapter, we'll cover the Spring Cloud technologies that we will use, some best practices for designing a cloud microservice oriented application, and the first steps to creating our first microservice using Spring Boot and Java.

1.14 Summary

- Monolithic architectures have all processes tightly coupled, and they run as a single service.
- Microservices are extremely small pieces of functionality that are responsible for one specific area of scope.
- Spring Boot allows you to create both types of architectures.
- Monolithic architectures tend to be ideal for simple, lightweight applications and microservices architectures are usually better for developing complex and evolving applications. In the end, selecting the software architecture will depend entirely on your project size, time, and requirements, among other factors.
- Spring Boot is used to simplify the building of REST-based/JSON microservices. Its goal is to make it possible for you to build microservices quickly with nothing more than a few annotations.
- Writing microservices is easy, but fully operationalizing them for production requires additional forethought. There are several categories of microservice development patterns, including core development,

routing patterns, client resiliency, security, application metrics, and build/deployment patterns.

- The microservice routing patterns deal with how a client application that wants to consume a microservice discovers the location of the service and is routed over to it.
- To prevent a problem in a service instance from cascading up and out to the consumers of the service, use the client resiliency patterns. Such as Circuit breaker pattern to avoid making calls to a failing service, Fallback pattern to create alternative paths in order to retrieve data or execute a specific action when a service fails, Client load balancing pattern to scale and remove all possible bottleneck or points of failure scenarios and Bulkhead pattern to limit the number of concurrent calls to a service in order to stop poor performance calls affect negatively other services.
- OAuth 2.0 is the most common user authorization protocol and is an excellent choice for securing a microservice architecture.
- The build/deploy pattern allows you to integrate the configuration of your infrastructure right into your build-deployment process so that you no longer deploy software artifacts such as Java WAR or EAR to an already-running piece of infrastructure.

2 Exploring the microservices world with Spring Cloud

This chapter covers

- Learning about the technologies of Spring Cloud
- Understanding the principles of cloud native applications
- Applying the Twelve-Factor App best practices
- Using Spring Cloud to build microservices

Designing, implementing, and maintaining microservices can quickly become a problem if they are not managed correctly. When we start working with microservices solutions, it is essential to apply best practices to keep the architecture as efficient and scalable as possible to avoid performance issues, bottlenecks, or operational problems inside our product. It is also easier to onboard new developers when using well-known patterns.

As we continue our discussion of microservices architectures, keep the following in mind:

The more distributed a system is, the more places it can fail.

By this, I mean that with a microservice architecture, we have higher points of failure because instead of having a single monolith application, we now have an ecosystem of multiple individual services that interact with each other. This is the main reason why many developers can encounter different administration and synchronization challenges or points of failure while creating microservices applications or architectures.

To avoid possible points of failure, we will use Spring Cloud. Spring Cloud offers a set of features such as service registration and discovery, circuit breakers, monitoring, and other services that allow us to quickly build microservices architectures with minimal configurations.

This chapter briefly introduces the Spring Cloud technologies that we will use throughout the book. This is a high-level overview; when you use each technology in this book, I'll teach you the details on each as needed. As we will use microservices throughout the next chapters, it is crucial to understand the concept of microservices, their benefits, and their development patterns

2.1 What is Spring Cloud?

Implementing all of the patterns I explained in the first chapter from scratch would be a tremendous amount of work. Fortunately for us, the Spring team has integrated a wide number of battle-tested open source projects into a Spring subproject collectively known as Spring Cloud (<http://projects.spring.io/spring-cloud/>).

Spring Cloud is a collection of tools that wraps the work of open source companies such as VMware, HashiCorp and Netflix in delivering patterns. Spring Cloud simplifies setting up and configuring the projects that give solutions to the most commonly encountered patterns into your Spring application so that you can focus on writing code, not getting buried in the details of configuring all the infrastructure that can go with building and deploying a microservice application.

Figure 2.1 maps the patterns listed in the previous chapter to the Spring Cloud projects that implement them.



Figure 2.1 You can map the technologies you're going to use directly to the microservice patterns we've

explored so far.

2.1.1 Spring Cloud Config

Spring Cloud Config handles the management of the application configuration data through a centralized service, so your application configuration data (particularly your environment-specific configuration data) is cleanly separated from your deployed microservice. This ensures that no matter how many microservice instances you bring up; they'll always have the same configuration. Spring Cloud Config has its own property management repository, but also integrates with open source projects such as the following:

- **Git.** Git (<https://git-scm.com/>) is an open-source version control system that allows you to manage and track changes to any text file. Spring Cloud Config can integrate with a Git-backend repository and read the application's configuration data out of the repository.
- **Consul.** Consul (<https://www.consul.io/>) is an open-source service discovery tool that allows service instances to register themselves with the service. Service clients can then ask Consul where the service instances are located. Consul also includes a key-value store based database that can be used by Spring Cloud Config to store application configuration data.
- **Eureka.** Eureka (<https://github.com/Netflix/eureka>) is an open source Netflix project that, like Consul, offers similar service discovery capabilities. Eureka also has a key-value database that can be used with Spring Cloud Config.

2.1.2 Spring Cloud Service Discovery

With Spring Cloud service discovery, you can abstract away the physical location (IP and/or server name) of where your servers are deployed from the clients consuming the service. Service consumers invoke business logic for the servers through a logical name rather than a physical location. Spring Cloud service discovery also handles the registration and deregistration of services instances as they're started up and shut down. Spring Cloud service discovery can be implemented using Consul (<https://www.consul.io/>), Zookeeper (<https://spring.io/projects/spring-cloud-zookeeper>) and Eureka (<https://github.com/Netflix/eureka>) as its service discovery engine.

NOTE Although Consul and Zookeeper are very powerful and flexible, Eureka is still very used by Java developers community. So, this book will contain examples with Eureka, to keep the book manageable, material focused and to reach the largest audience possible. But if you are interested in Consul or Zookeeper please make sure you read Appendix C and D. In the Appendix C I added an example of how we can use Consul as a service discovery and in Appendix D I added another example of how to use Zookeeper.

2.1.3 Spring Cloud Load Balancer and Resilience4j

Spring cloud heavily integrates with several open-source projects. For microservice client resiliency patterns, Spring Cloud wraps the Resilience4j library (<https://github.com/resilience4j/resilience4j>) and Spring Cloud Load Balancer project and makes using them from within your own microservices trivial to implement.

By using the Resilience4j libraries, you can quickly implement service client resiliency patterns such as the

circuit breaker, retry, bulkhead and more patterns.

While the Spring Cloud Load Balancer project simplifies integrating with service discovery agents such as Eureka, it also provides client-side load-balancing of service calls from a service consumer. This makes it possible for a client to continue making service calls even if the service discovery agent is temporarily unavailable.

2.1.4 Spring Cloud API Gateway

The API Gateway provides service routing capabilities for your microservice application. Like the name says is a service gateway that proxies service requests and makes sure that all calls to your microservices go through a single “front door” before the targeted service is invoked. With this centralization of service calls, you can enforce standard service policies such as security authorization, authentication, content filtering, and routing rules. The API Gateway can be implemented using Zuul (<https://github.com/Netflix/zuul>) and Spring Cloud Gateway (<https://spring.io/projects/spring-cloud-gateway>).

NOTE In this book, I'm going to use the Spring Cloud Gateway that was built with Spring Framework 5, Project Reactor (allowing integration with Spring Web Flux) and Spring Boot 2 to have better integration with our Spring projects.

2.1.5 Spring Cloud Stream

Spring Cloud Stream (<https://cloud.spring.io/spring-cloud-stream/>) is an enabling technology that allows you to easily integrate lightweight message processing into your

microservice. Using Spring Cloud Stream, you can build intelligent microservices that can use asynchronous events as they occur in your application. With Spring Cloud Stream, you can quickly integrate your microservices with message brokers such as RabbitMQ (<https://www.rabbitmq.com/>) and Kafka (<http://kafka.apache.org/>).

2.1.6 Spring Cloud Sleuth

Spring Cloud Sleuth (<https://cloud.spring.io/spring-cloud-sleuth/>) allows you to integrate unique tracking identifiers into the HTTP calls and message channels (RabbitMQ, Apache Kafka) being used within your application. These tracking numbers, sometimes referred to as correlation or trace ids, allow you to track a transaction as it flows across the different services in your application. With Spring Cloud Sleuth, these trace IDs are automatically added to any logging statements you make in your microservice.

The real beauty of Spring Cloud Sleuth is seen when it's combined with logging aggregation technology tools such as the ELK stack (<https://www.elastic.co/what-is/elk-stack>) and tracking tools such as Zipkin (<http://zipkin.io>).

The ELK stack is the acronym for three open-source projects: Elasticsearch(<https://www.elastic.co/>) a search and analytics engine, Logstash(<https://www.elastic.co/products/logstash>) a server-side data processing pipeline that consumes data, transforms it in order to send it to a "stash" and Kibana(<https://www.elastic.co/products/kibana>) is a client UI that allows the user to query and visualize the data of the whole stack.

Open Zipkin takes data produced by Spring Cloud Sleuth and allows you to visualize the flow of your service calls involved for a single transaction.

2.1.7 Spring Cloud Security

Spring Cloud Security (<https://cloud.spring.io/spring-cloud-security/>) is an authentication and authorization framework that can control who can access your services and what they can do with your services. Spring Cloud Security is token-based and allows services to communicate with one another through a token issued by an authentication server. Each service receiving a call can check the provided token in the HTTP call to validate the user's identity and their access rights with the service.

In addition, Spring Cloud Security supports the JSON Web Token (<https://jwt.io>). The JSON Web Token (JWT) standardizes the format of how an OAuth2 token is created and provides standards for digitally signing a generated token.

2.2 Spring Cloud by example

In the last section, I explained all the different Spring Cloud technologies that you're going to use as you build out your microservices. Because each of these technologies are independent services, it will take more than one chapter to explain all of them in detail. However, as I wrap up this chapter; I want to leave you with a small code example that

again demonstrates how easy it is to integrate these technologies into your own microservice development effort.

Unlike the first code example in listing 1.1, you can't run this code example because a number of supporting services need to be set up and configured to be used. Don't worry, though; the setup costs for these Spring Cloud services (configuration, service discovery) are a one-time cost in terms of setting up the service. Once they're set up, your individual microservices can use these capabilities over and over again. We couldn't fit all that goodness into a single code example at the beginning of the book.

The code shown in the following listing quickly demonstrates how the service discovery, and client-side load balancing of remote services were integrated into our "Hello World" example.

Listing 2.1 Hello World Service using Spring Cloud

```
package com.optimagrowth.simpleservice;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
@RestController
@RequestMapping(value="hello")
@EnableEurekaClient #A
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(ContactServerAppApplication.class, args);
    }
    public String helloRemoteServiceCall(String firstName, String lastName){
        RestTemplate restTemplate = new RestTemplate();
        ResponseEntity<String> restExchange =
            restTemplate.exchange(
                "http://logical-service-id/name/" #B
```

```

+ "{firstName}/{lastName}"
,HttpMethod.GET, null, String.class,
firstName, lastName);
return restExchange.getBody();
}
@RequestMapping(value="/{firstName}/{lastName}",method = RequestMethod.GET)
public String hello( @PathVariable("firstName") String firstName,
@PathVariable("lastName") String lastName) {
return helloRemoteServiceCall(firstName, lastName);
}
}

```

#A Tells the service that it should register itself with a Eureka service discovery agent and that service calls are to use service discovery to “lookup” the location of remote services

#B Uses a decorated RestTemplate class to take a “logical” service ID and Eureka under the covers to look up the physical location of the service

This code has a lot packed into it, so let’s walk through it. Keep in mind that this listing is only an example and isn’t found in the chapter 2 GitHub repository source code. I’ve included it here to give you a taste of what’s to come later in the book.

The first thing you should notice is the `@EnableEurekaClient` annotation. The `@EnableEurekaClient` annotation tells your microservice to register itself with a Eureka Service Discovery agent and that you’re going to use service discovery to lookup remote REST services endpoints in your code. Note that configuration is happening in a property file that will tell the simple service the location and port number of a Eureka server to contact.

The second and last thing to note is what’s occurring inside the `helloRemoteServiceCall` method. The presence of the `@EnableEurekaClient` has told Spring Boot that you are enabling the Eureka Client, it is important to highlight that this annotation is optional if you already have the spring-

cloud-starter-netflix-eureka-client dependency in the pom.xml. The RestTemplate class will allow you to pass in a logical service ID for the service you're trying to invoke:

```
 ResponseEntity<String> restExchange = restTemplate.exchange  
(http://logical-service-id/name/{firstName}/{lastName})
```

Under the covers, the RestTemplate class will contact the Eureka service and look up the physical location of one or more of the “name” service instances. As a consumer of the service, your code never has to know where that service is located.

Also, the RestTemplate class is using the Spring Cloud Load Balancer library. The Spring Cloud Load Balancer will retrieve a list of all the physical endpoints associated with a service. Every time the service is called by the client, it “round-robbins” the call to the different service instances without having to go through a centralized load balancer. By eliminating a centralized load balancer and moving it to the client, you eliminate another failure point (load balancer going down) in your application infrastructure.

I hope that at this point you’re impressed, because you’ve added a significant number of capabilities to your microservice with only a few annotations. That’s the real beauty behind Spring Cloud. You as a developer get to take advantage of battle-hardened microservice capabilities from premier cloud companies like Netflix and Consul. Spring

Cloud simplifies their use to literally nothing more than a few simple Spring Cloud annotations and configuration entries.

Before we start building our first microservice, let's make a pause and review the best practices to implement a cloud-native microservice are.

2.3 How to build a cloud-native microservice?

In this section, we must make a pause and understand what the best practices for designing cloud microservice-oriented application are. In the previous chapter, I explained the difference between the cloud computing models, but what exactly is the cloud? The cloud is not a place, instead is a technology resource management that offers replacement of local machines and private data centers by using virtual infrastructure. There are several levels or types of cloud applications, but in this section, we are only going to focus on two types of cloud applications — Cloud-Ready, and Cloud-native.

A cloud-ready application is an application that was once used on a computer or on-site server. These types of applications with the arrival of the cloud have changed their environment and have moved from static to dynamic environments with the aim of running in the cloud. For example, a cloud-unready application could be a local on-premise application that only contains one specific database configuration that must be customized in each installation environment (development, stage, production). In order to

make an application like this cloud-ready, we need to externalize the application's configuration so it can be quickly adapted to the different environments. By doing this, we can ensure the application can run into multiple environments without changing any source code during the builds.

A cloud-native application is an application designed specifically for a cloud computing architecture to take advantage of all its benefits and services. When creating this type of applications, the developers divide the functions into microservices, with scalable components such as containers in order to be able to run on several servers. These services are managed by virtual infrastructures through DevOps processes with continuous delivery workflows. It's important to understand that these types of applications do not require any change or conversion to work in the cloud and are designed to deal with the unavailability of downstream components.

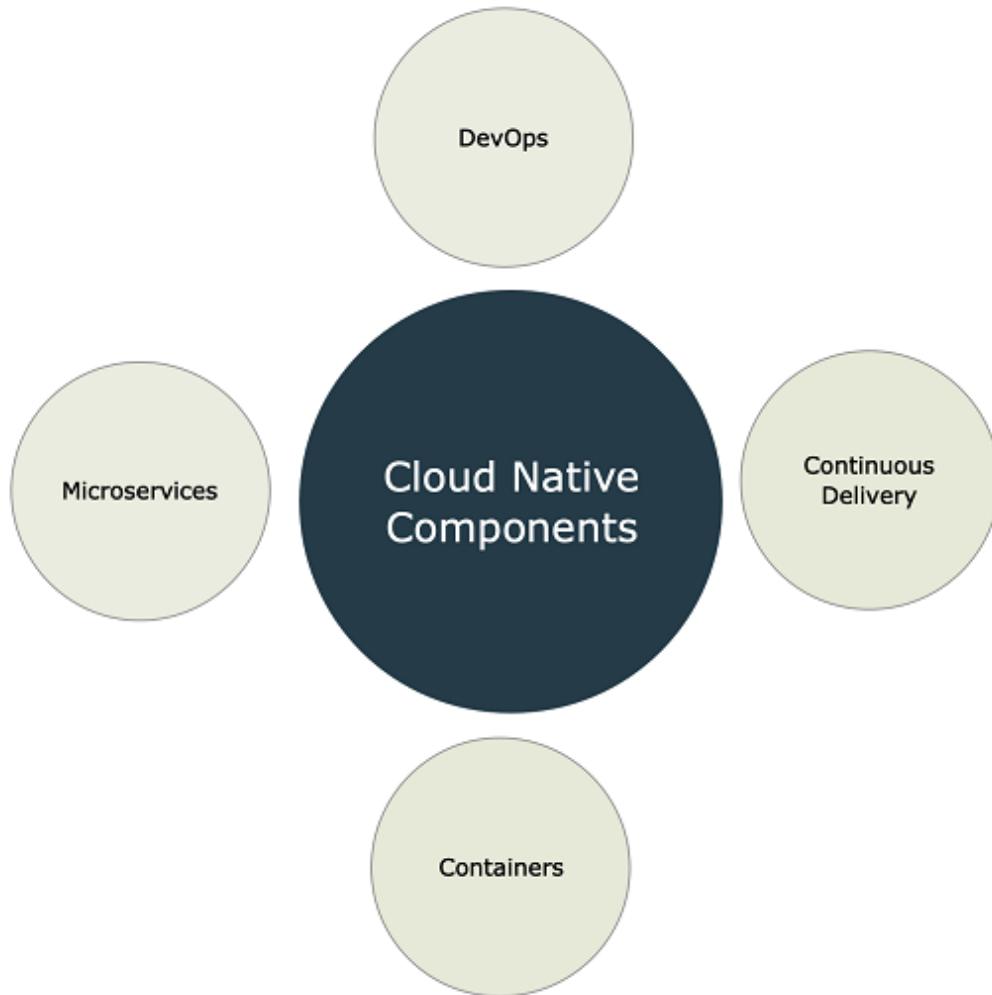


Figure 2.2 Cloud-native applications are built with scalable components like containers, deployed as microservices and managed on virtual infrastructures through DevOps processes with continuous delivery workflows.

The four principles of native cloud development are:

- **DevOps.** Is the acronym for development (Dev) and operations (Ops), which refers to a software development methodology that focuses on communication, collaboration and integration between software

developers and IT operations. The DevOps main goal is to automate the software delivery processes and infrastructure changes at lower costs.

- **Microservices.** A microservice is a small, loosely coupled, distributed service. Microservices allow you to take a large application and decompose it into easy-to-manage components with narrowly defined responsibilities. Microservices help combat the traditional problems of complexity in a large code base by decomposing the large code base down into small, well-defined pieces.
- **Continuous Delivery.** It's a software development practice in which the process of delivering software is automated to allow short-term deliveries into a production environment.
- **Containers.** Containers are a natural extension of deploying your microservices on a virtual machine image. Rather than deploying a service to a full virtual machine, many developers deploy their services as Docker containers (or similar container technology) to the cloud.

In this book, we are going to focus on creating microservices and from now on, we should remember that by definition they are cloud-native meaning that can be executed on multiple cloud providers and can obtain all of the benefits of the cloud services.

In order to face the challenges of creating cloud-native microservices, we will use the Heroku's best practice guide called twelve factor app in order to build high-quality microservices. The twelve factor app is a methodology to build and develop cloud native applications (microservices). We can see this as a collection of development and

designing practices that focus on dynamic scaling and the fundamental points while building a distributed service.

This document (<https://12factor.net/>) was created in 2002 by several developers in Heroku with the main goal to provide twelve best practices you should always keep in the back of your mind when building microservices. I chose the 12 factors document because it is one of the most complete guides to follow when we are creating cloud-native applications. This guides not only provide a common vocabulary about the most common problems observed in the development of modern applications but, also provides robust solutions to tackle these problems.

NOTE In this chapter I will briefly introduce each best practice because as you continue reading this book, you'll see how I intend to use the twelve-factor methodology throughout the book and apply them into examples with the Spring Cloud projects and other technologies.

Figure 2.3 shows the 12 best practices covered by the Twelve factor app Manifesto:

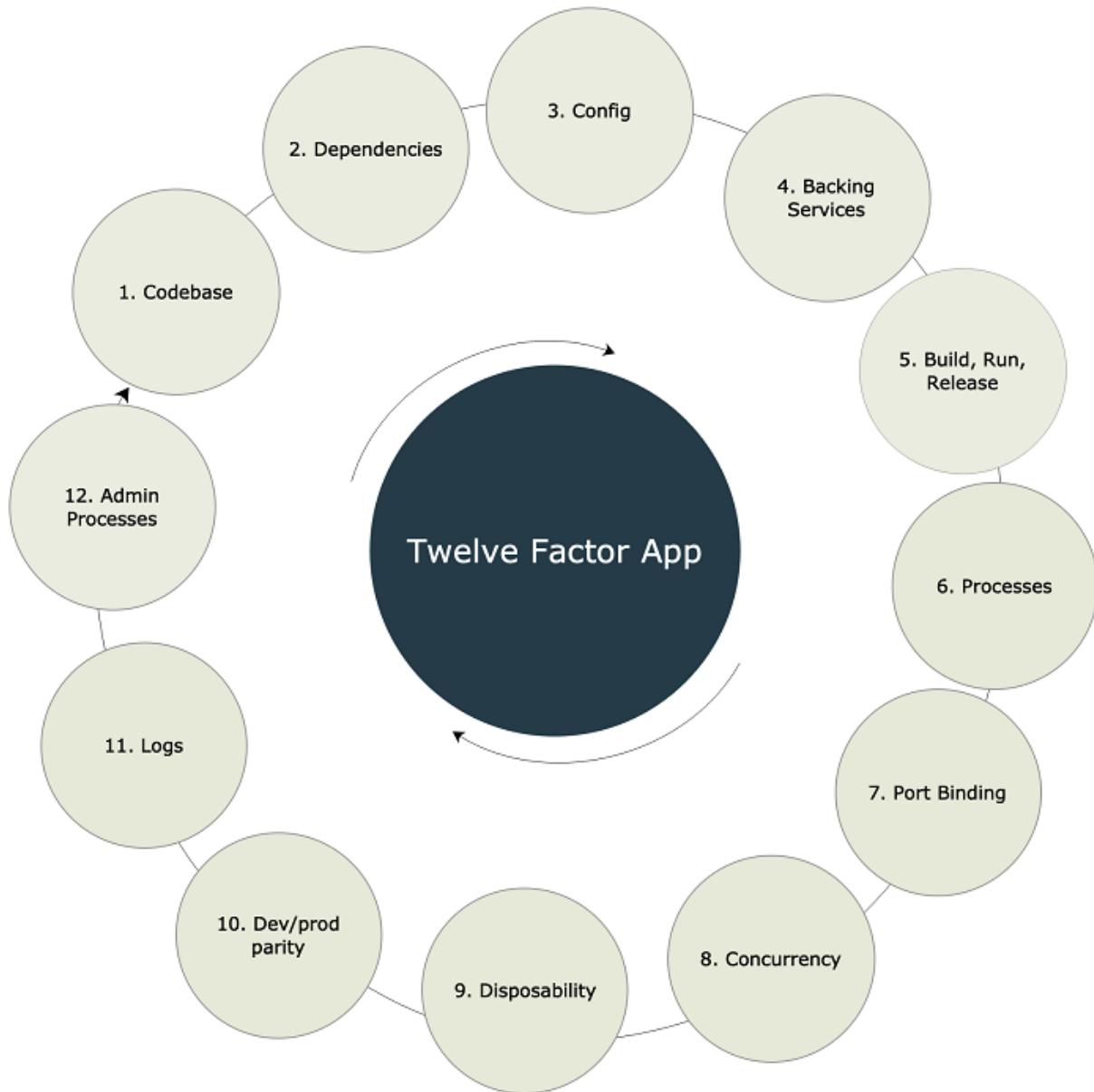


Figure 2.3 Twelve factor application best practices.

In the following sections, I will give you a high-level overview of each practice; the idea is that when you use each technology in this book, I will teach you all the details on each as needed.

2.3.1 Code Base

This practice explains that each microservice should have a single codebase, managed in source control. Also, it is essential to highlight that the server provisioning information should also be in version control. Remember, version control is the management of changes to a file or set of files. The code base can have multiple instances of deployment environments such as development, testing, staging, production, and more but is not shared with any other microservice. This is an important guideline because if we share the base for all the microservices we would end up producing a lot of immutable releases that belong to different environments. Figure 2.4 shows a single codebase with many deployments.

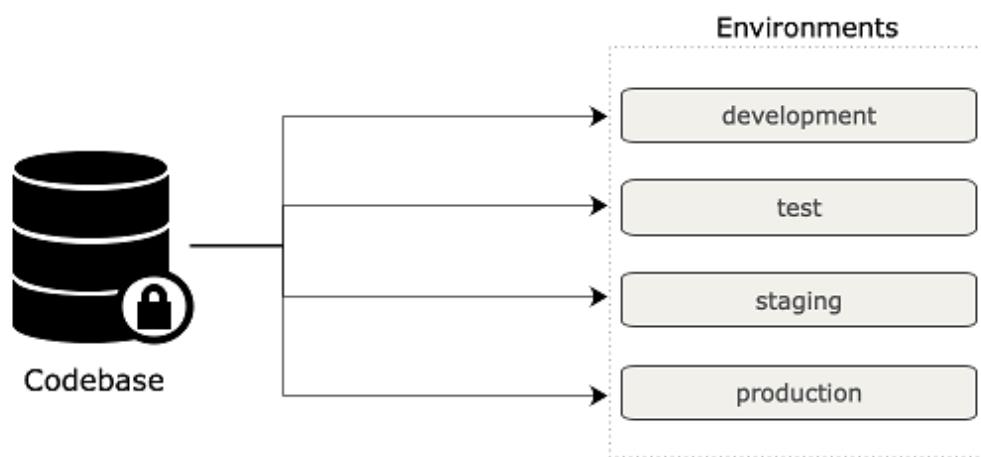


Figure 2.4 One single codebase with many deployments.

2.3.2 Dependencies

Explicitly declare the dependencies your application uses through build tools such as Maven, Gradle (Java). Third-party JAR dependence should be declared using their specific versions number. This allows your microservice to always be built using the same version of libraries.

If you are new to the build tools concept, the following figure can help you understand how a build tool work. First, maven reads the dependencies stored in the pom.xml file, then it searches them on the local repository. If they are not found over there, then it proceeds to download them from the maven central repository and inserts them into your local repository for future uses. Figure 2.5 shows how Maven works.

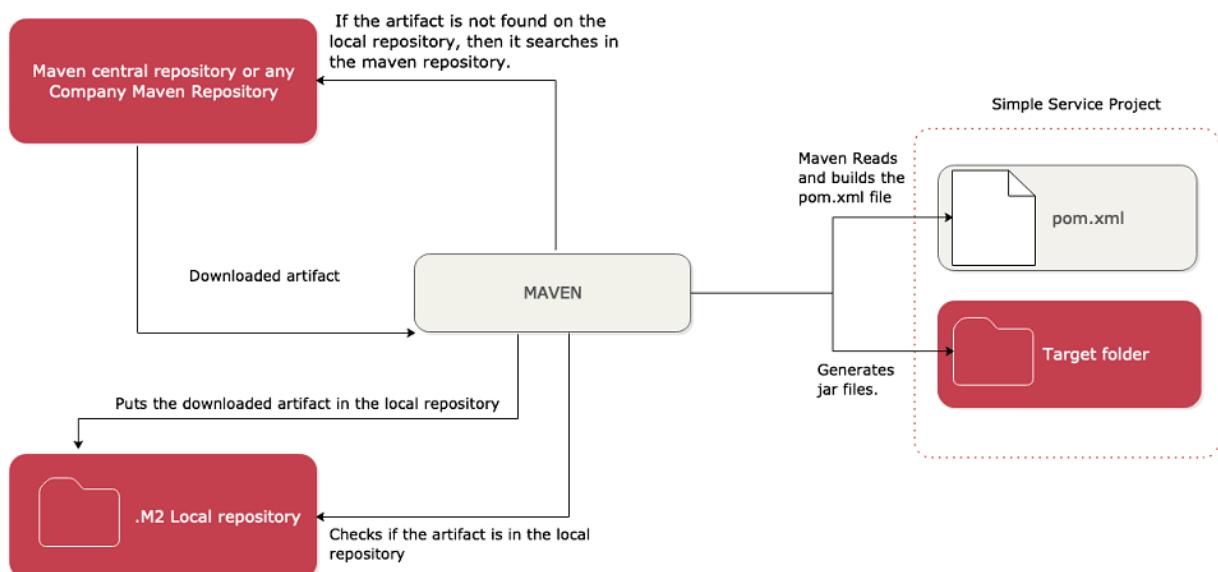


Figure 2.5 Maven reads the dependencies stored in the pom.xml file and then searches them on the local repository. If they are not found on the local repository, then Maven proceeds to download and it also downloads the dependencies of those dependencies from the maven repository and puts them into your local repository.

2.3.3 Config

Store your application configuration (specially your environment-specific configuration) independently from your code. Never add embedded configurations to your source code; instead, maintain your configuration completely separated from your deployable microservice. Imagine this scenario; we want to update a configuration for a specific microservice that has been replicated a hundred times in a server. If we keep the configuration packaged within the microservice, we'll need to redeploy each of the hundred instances to make the change. However, with this guideline microservices can load the external configuration at startup or they can use their service to reload the external configuration at runtime without having to restart the microservice. Figure 2.6 shows an example of how your environment should look like.

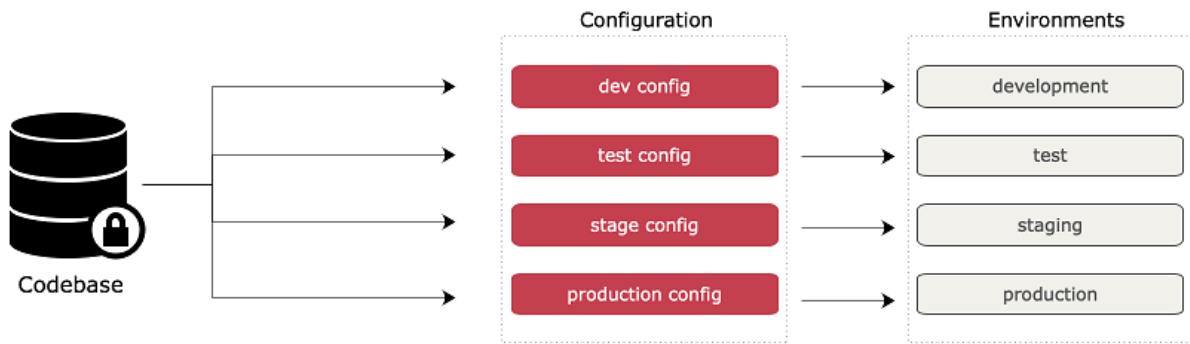


Figure 2.6 Externalize the environment-specific configurations in the environment.

2.3.4 Backing services

Your microservice will often communicate over a network to databases, API RESTful services, other servers or message systems. When it does, you should ensure that at any time, you can swap out your implementation from an in-house managed service to a third-party service without making changes to the microservice's code. This best practice indicates that a deploy should be able to swap between local connections to third party without any changes to the application code. In chapter 12, we are going to see how to move the microservices from a locally managed database to one managed by Amazon. Figure 2.7 shows an example of some backing services we might have on our application.

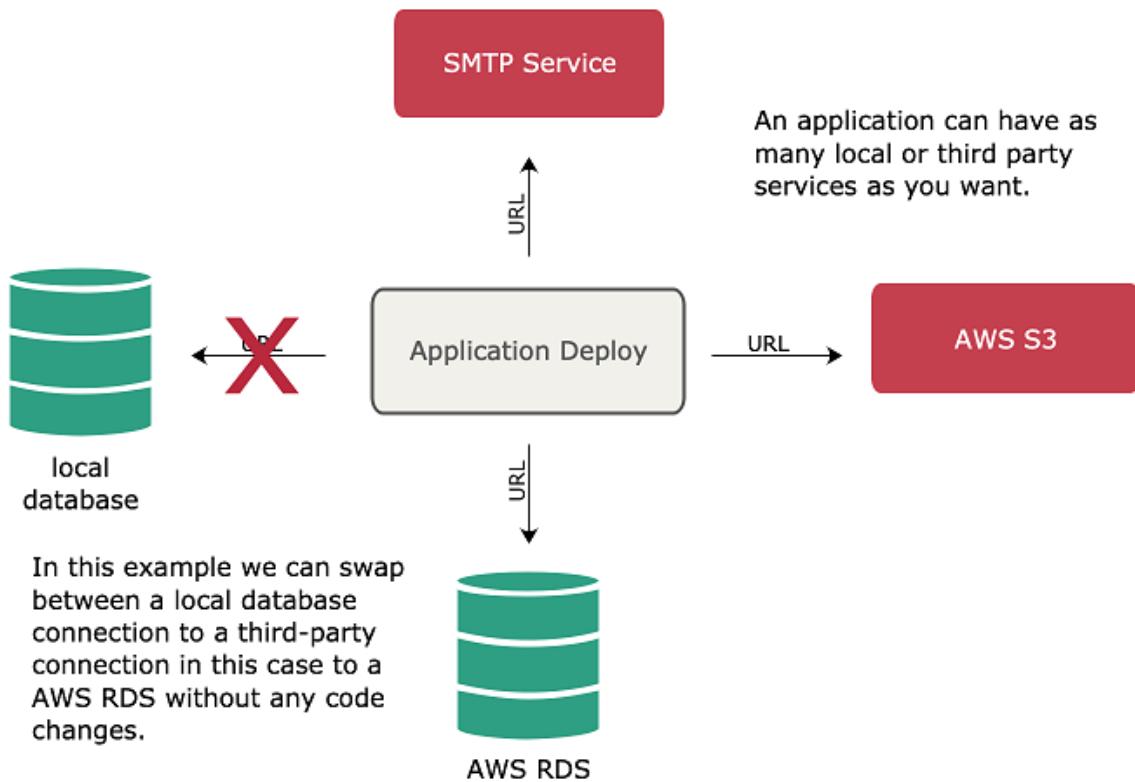


Figure 2.7 A backing service is considered any service the application consumes over the network. The application deploy should be able to swap a local connection to a third-party without any changes to the code.

2.3.5 Build, release, run

Keep your build, release, and run stages of deploying your application completely separated. We should be able to build microservices that are independent of the environment which they are running. Once code is built, the developer should never make changes to the code at runtime. Any changes need to go back to the build process and be

redeployed. A built service is immutable and cannot be changed. The release phase is in charge of combining the built service with the specific configuration for each target environment. When we do not separate the different stages, we can find problems and differences in the code that are not trackable. For example, if we modify a service already deployed in production, this change will not be able to be tracked in the repository, and two situations probably occur: Changes get lost with newer versions of the service, or we are forced to copy the changes to the new version of the service. Figure 2.8 shows a high-level architecture example.

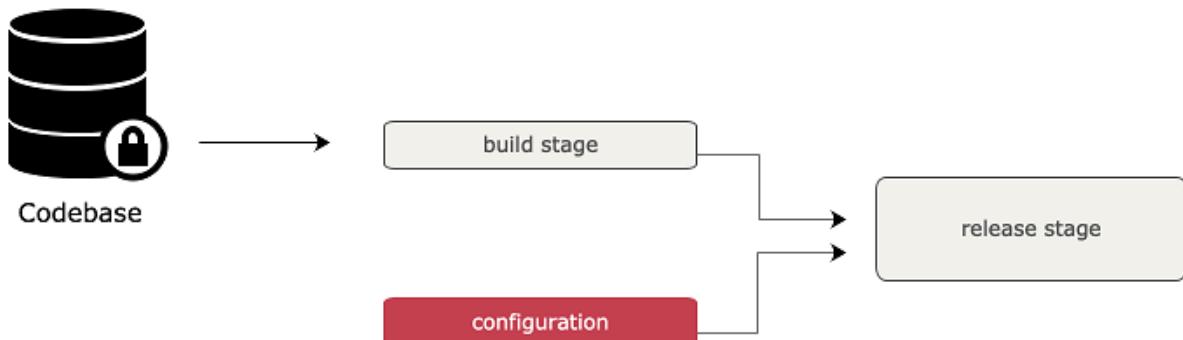


Figure 2.8 Strictly separate the build, release and run stages.

2.3.6 Processes

Your microservices should always be stateless and should only contain the necessary information to carry out the requested transaction. Microservices can be killed and replaced at any time without the fear that a loss of a service-instance will result in data loss. If there is a specific

requirement to store a state, it must be done through an in-memory cache such as Redis or a backing database. Figure 2.9 shows how stateless microservices works.

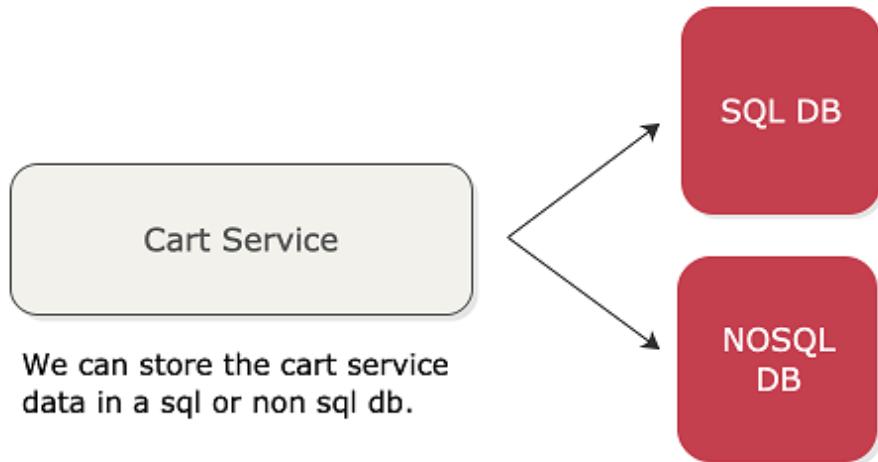


Figure 2.9 Stateless microservices don't store any session data (state) on the server. They use SQL or NON SQL DBs to store all the information.

2.3.7 Port binding

Port binding means publish services through a specific port. In a microservices architecture a microservice is completely self-contained with the runtime engine for the service packaged in the service executable. You should run the service without the need for a separated web or application server. The service should start by itself on the command line and be accessed immediately through an exposed HTTP port.

2.3.8 Concurrency

The concurrency best practice explains that cloud-native applications should scale out using the process model. What does this mean? Let's imagine rather than making a single significant process larger. We can create multiple processes and then distribute the service's load or application among the different processes.

Vertical scaling (Scale up) refers to increase the hardware infrastructure (CPU, RAM). Horizontal scaling (Scale out) refers to adding more instances of the application. When you need to scale, launch more microservice instances and scale out and not up. Figure 2.10 shows the difference between both types of scaling.

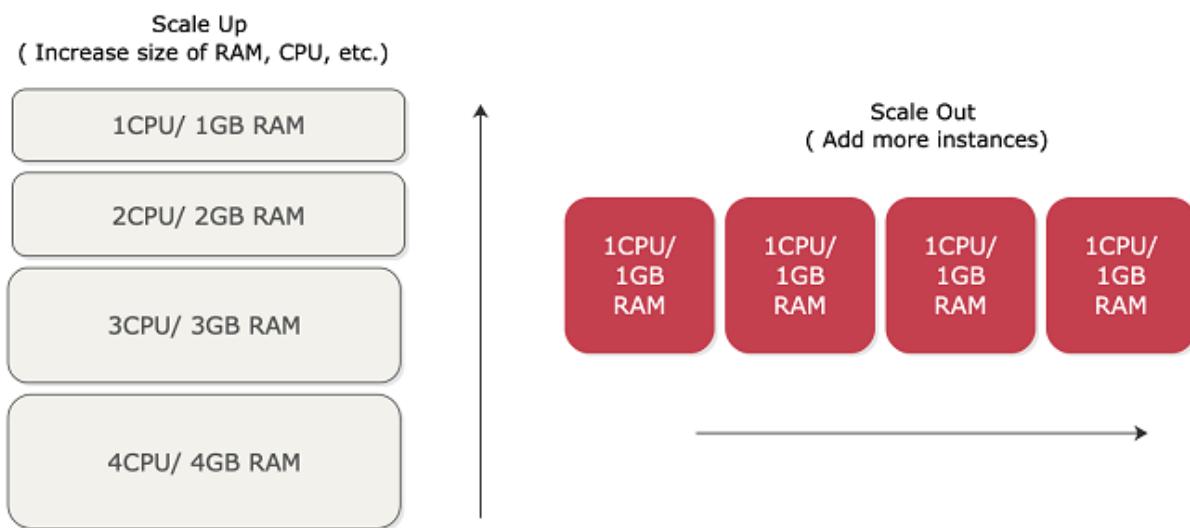


Figure 2.10 Differences between Scale Up and Scale Out.

2.3.9 Disposability

Microservices are disposable and can start and stop on demand in order to facilitate the elastic scaling and to quickly deploy application code and config changes. Ideally, these startups last a few seconds from the moment the launch command executes until the process is ready to receive requests. What I mean with disposable is that we can remove failing instances with other new instances without affecting any other services. For example, if one of the instances of the microservice is failing because of a failure in the underlying hardware, we can shut down that instance without affecting other microservices and start another one somewhere else if needed.

2.3.10 Dev/prod parity

This best practice refers to have different environments (for example, development, staging, production) as similar as possible. The environments should always contain similar versions of deployed code, as well as the infrastructure and services. This could be done with continuous deployment, that automates the deployment process as much as possible and it allows a microservice to be deployed between environments in short periods. As soon as a code is committed, it should be tested and then promoted as quickly as possible from development all the way to production. This guideline is essential if we want to avoid deployment errors. Having similar development and production environments allows us to control all the possible scenarios we might have while deploying and executing our application.

2.3.11 Logs

Logs are a stream of events. As logs are written out, they should be managed by tools, such as Logstash(<https://www.elastic.co/products/logstash>) or Fluentd (<http://fluentd.org>), that will collect the logs and write them to a central location. The microservice should never be concerned about the mechanisms of how this happens, they only need to focus on writing the log entries into the stdout. In chapter 11, I will demonstrate how to provide an auto-configuration for sending these logs to the ELK stack (Elasticsearch, Logstash and Kibana). Figure 2.11 shows how the logging works in a microservice architecture using the ELK stack.

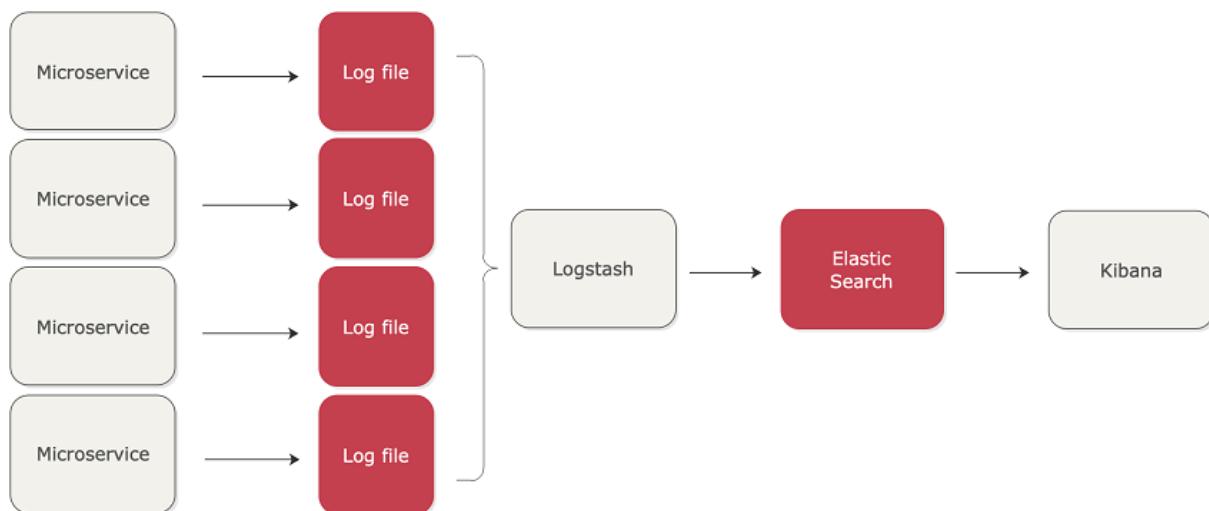


Figure 2.11 Manage the microservices logs with the ELK architecture.

2.3.12 Admin processes

Developers will often have to do administrative tasks against their services (Data migration or conversion). These tasks should never be ad hoc and instead should be done via scripts that are managed and maintained through source code repository. These scripts should be repeatable and non-changing (the script code isn't modified for each environment) across each environment they're run against. It's important to have defined the types of tasks we need to take into consideration while running our microservice, in case we have multiple microservices with these scripts we are able to execute all of the administrative tasks without having to do it manually.

NOTE If you're interested in reading more about the Heroku's Twelve-Factor Application manifesto, please visit the Twelve factor app website (<https://12factor.net/>).

In chapter 8, I will explain how to implement these features using the Spring Cloud API Gateway.

Now that we have seen what the best practices are, we can continue with the next section where we are going to start building our first microservice with Spring Boot and Spring Cloud.

2.4 Making sure our examples are relevant

I want to make sure this book provides examples that you can relate to as you go about your day-to-day job. To this end, I've structured the chapters in this book and the

corresponding code examples around a software product of a fictitious company called Optima Growth.

This company is a software development company whose core product, Optima Stock which we are going to refer as O-stock from now on, provides an enterprise-grade software asset management application. It provides coverage for all the critical elements: inventory, software delivery, license management, compliance, cost and resource management. Its primary goal is to enable organizations to gain an accurate point-in-time picture of its software assets.

The company is approximately 12 years old. While they've experienced solid revenue growth, internally they're debating whether they should be re-platforming their core product from a monolithic on-premise-based application or move their application to the cloud. The re-platforming involved with O-stock can be a "make or break" moment for the company.

The company is looking at rebuilding their core product O-stock on a new architecture. While much of the business logic for the application will remain in place, the application itself will be broken down from a monolithic to a much smaller microservice design whose pieces can be deployed independently to the cloud. The examples in this book won't build the entire Optima Growth application. Instead, you'll build specific microservices from the problem domain at hand and then create the infrastructure that will support these services using various Spring Cloud (and some non-Spring-Cloud) technologies.

The ability to successfully adopt a cloud-based microservice architecture will impact all parts of a technical organization, including the architecture, engineering (development), and operations teams. Input will be needed from each group and, in the end, they're probably going to need reorganization as the team reevaluates their responsibilities in this new environment. Let's start our journey with Optima Growth as you begin the fundamental work of identifying and building out several of the microservices used in O-stock and then building these services using Spring Boot.

NOTE I understand that the architecture of an asset management system is complex. Therefore, within this book, we will only use some of the basic concepts of it. It is essential to highlight that in this book we will only focus on creating a complete microservice architecture with a simple system as an example. Creating a complete software asset management application is beyond the scope of this book.

2.5 Building a microservice with Spring Boot and Java

In this section, we are going to build the skeleton of a microservice called licensing service for the Optima Growth company mentioned in the previous section. It is important to highlight that all of the microservices are going to be created using Spring Boot. Spring Boot, as I previously mentioned, is an abstraction layer over the Spring libraries that allows developers to build Groovy- and Java-based web applications quickly and microservices with significantly less ceremony and configuration than a full-blown Spring application.

For the licensing service example, we'll use Java as your core programming language and Apache maven as our build tool.

Over the next sections, you're going to

1. Build the basic skeleton of the microservice and a Maven script to build the application.
2. Implement a Spring bootstrap class that will start the Spring container for the microservice and initiate the kick-off of any initialization work for the class.

2.5.1 Setting up the environment

To start building our microservices you should have the following components:

- Java 11 (<https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html>).
- Maven 3.5.4 or later (<https://maven.apache.org/download.cgi>)
- Spring Tools 4 (<https://spring.io/tools>) or also you can download it within the selected Integrated Development Environment (IDE).
- IDEs such as Eclipse (<https://www.eclipse.org/downloads>), IntelliJ IDEA (<https://www.jetbrains.com/idea/download/>) or NetBeans (<https://netbeans.org/features/index.html>)

NOTE All of the code listings from now on will be created using Spring Framework 5 and Spring Boot 2. It is important to understand that I'm not going to explain all the features of Spring Boot I'm just going to highlight the essential ones in order to create the microservices. Another important fact is that I will be using Java 11 in this book, in order to reach out to reach the largest audience possible.

2.5.2 Getting started with the skeleton project

To begin, you'll create a skeleton project for the licensing service using the Spring Initializr. The Spring Initializr (<https://start.spring.io/>) is going to enable you to create a new Spring Boot project with the possibility to choose dependencies from an extensive list. Also, it will allow you to change specific configurations of the project you are about to create. Figure 2.12 and 2.13 shows how the Spring Initializr page should look like for the licensing service.

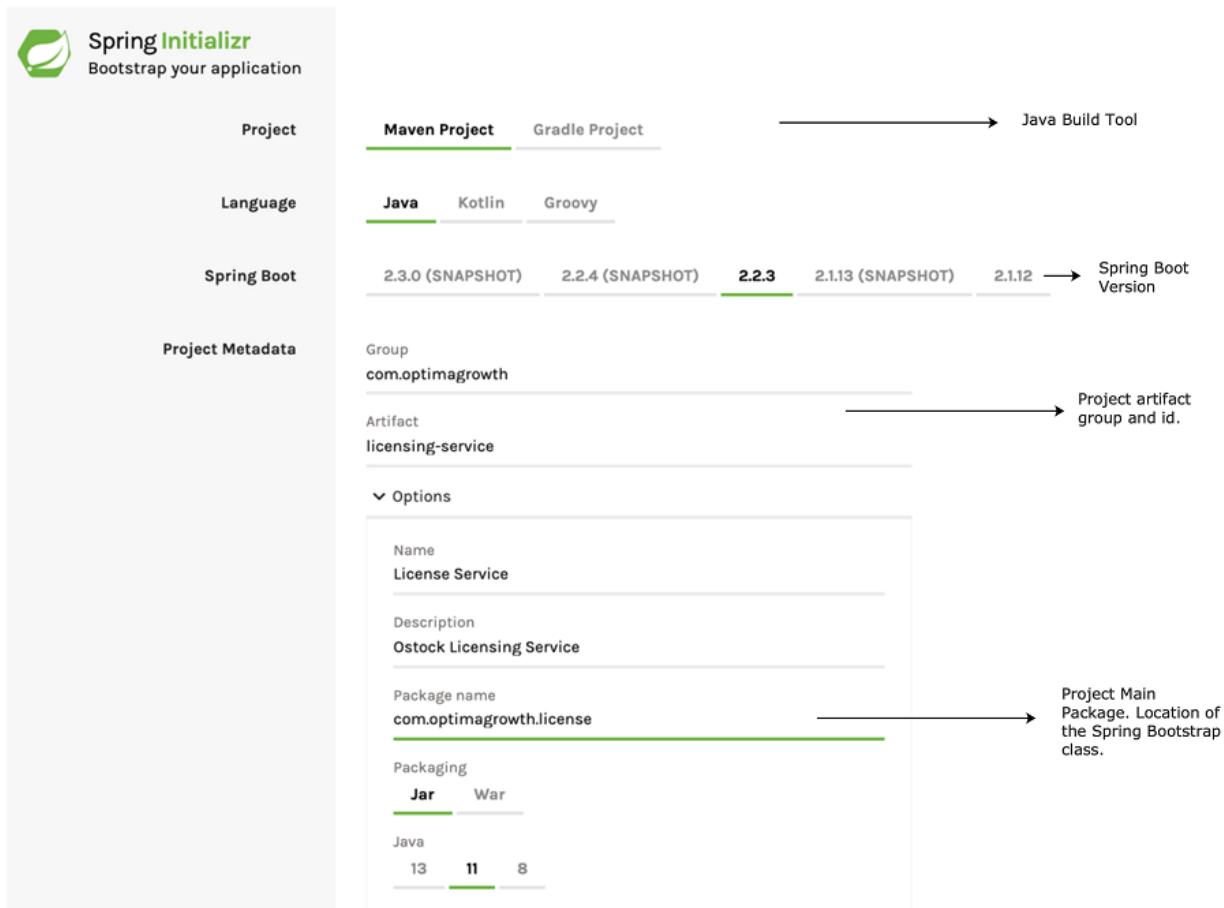


Figure 2.12 Spring Initializr configuration for the licensing service.

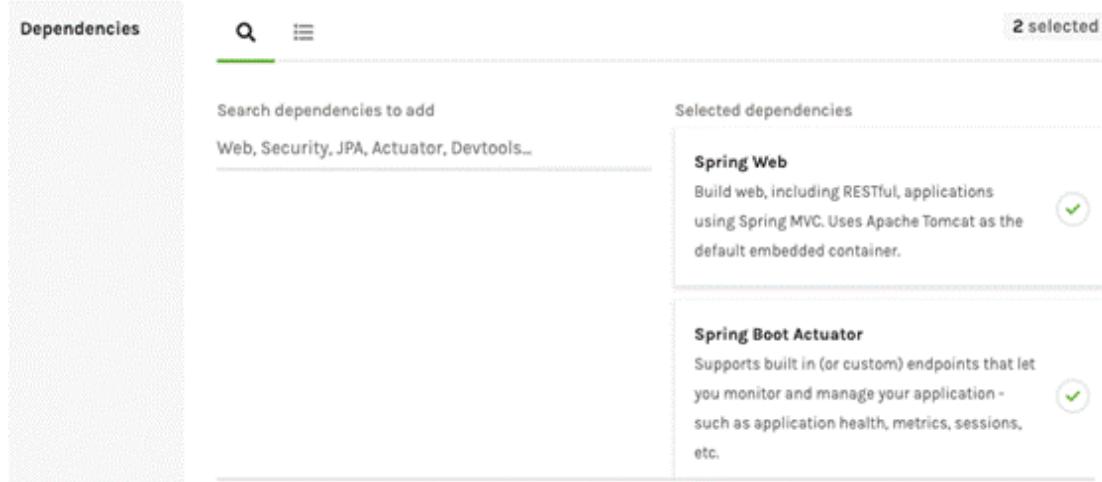


Figure 2.13 Spring Initializr dependencies Spring Web and Spring Boot Actuator dependencies for the licensing service.

NOTE You can also pull down the source code from the GitHub repository (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter2>)

Once you've created and imported the project as a Maven project into your preferred Integrated Development Environment (IDE). Let's create the following packages:

```
com.optimagrowth.license.controller  
com.optimagrowth.license.model  
com.optimagrowth.license.service
```

Figure 2.14 shows the initial project structure for the license service in the IDE

```
Licensing-service
- src/main/java
  - com.optimagrowth.license
  - com.optimagrowth.license.controller
  - com.optimagrowth.license.model
  - com.optimagrowth.license.service
- src/main/resources
  - static
  - templates
  - application.properties
- src/test/java
  - com.optimagrowth.license
- src
- target
- pom.xml
```

Figure 2.14 License project structure with the bootstrap class, application properties, tests and pom.xml

NOTE An in-depth discussion of how to test our microservices is outside of the scope this book, but if you are interested in diving into more detail on how to create unit, integration, and platform tests, I highly recommend Alex Soto Bueno, Andy Gumbrecht, and Jason Porter's book, Testing Java Microservices (Manning, 2018).

The code listing 2.2 shows how your pom.xml file should look like for your licensing service.

Listing 2.2 Maven pom file for the licensing service

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>
<parent>

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId> #A
<version>2.2.3.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.optimagrowth</groupId>
<artifactId>licensing-service</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>License Service</name>
<description>Ostock Licensing Service</description>
<properties>
<java.version>11</java.version> #B
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId> #C
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId> #D
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<scope>provided</scope>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId> #E
</plugin>
</plugins>
</build>
</project>

```

#A 1. Tells Maven to include the Spring Boot Starter Kit dependencies

#B 2. Java Version 11, by default, the pom adds Java 6. That's why we override it
with Java 11 in order to use Spring 5

#C 3. Tells Maven to include the Spring Actuator dependencies

#D 4. Tells Maven to include the Spring Boot Web dependencies

#E 5. Tells Maven to include Spring specific maven plugins for building and deploying Spring Boot applications.

NOTE A Spring Boot project doesn't need to set the individual Spring dependencies explicitly. These dependencies are automatically pulled from the Spring Boot core artifact defined in the pom. Spring Boot 2.x builds use Spring Framework 5.

We won't go through the entire file in detail but note a few key areas as we begin. Spring Boot is broken into many individual projects. The philosophy is that you shouldn't have to "pull down the world" if you aren't going to use different pieces of Spring Boot in your application. This also allows the various Spring Boot projects to release new versions of code independently of one another. To help simplify the life of the developers, the Spring Boot team has gathered related dependent projects into various "starter" kits. In part 1 of the Maven POM, you tell Maven that you need to pull down the version of the Spring Boot Framework. In the second part, you specify the version of java you're going to use, and in parts 3 and 4 of the Maven file, you identify that you're pulling down the Spring Web and Spring Actuator starter kits. It's essential to highlight that the Spring Actuator dependency is not required, but we will use several actuator endpoints in the next chapters, so that's why we are adding it at this point. These two projects are the heart of almost any Spring Boot REST-based service. You'll find that as you build more functionality into your services, the list of these dependent projects becomes longer.

Also, Spring has provided Maven plugins that simplify the build and deployment of the Spring Boot applications. In Part 4 tells your Maven build script to install the latest Spring

Boot Maven plugin. This plugin contains several add-on tasks (such as `spring-boot:run`) that simplify your interaction between Maven and Spring Boot.

In order to check the Spring dependencies pulled by Spring boot into our licensing service, we can use the maven goal `dependency:tree`. Figure 2.15 shows the dependency tree for the licensing service.

```
[INFO] --- maven-dependency-plugin:3.1.1:tree (default-cli) @ licensing-service ---
[INFO] com.optimagrowth:licensing-service:jar:0.0.1-SNAPSHOT
[INFO] +- org.springframework.boot:spring-boot-starter-hateoas:jar:2.2.3.RELEASE:compile
[INFO] | \- org.springframework.hateoas:spring-hateoas:jar:1.0.3.RELEASE:compile
[INFO] |   +- org.springframework:spring-aop:jar:5.2.3.RELEASE:compile
[INFO] |   +- org.springframework:spring-beans:jar:5.2.3.RELEASE:compile
[INFO] |   +- org.springframework:spring-context:jar:5.2.3.RELEASE:compile
[INFO] |   +- org.springframework.plugin:spring-plugin-core:jar:2.0.0.RELEASE:compile
[INFO] |   \- org.slf4j:slf4j-api:jar:1.7.30:compile
[INFO] +- org.springframework.boot:spring-boot-starter-actuator:jar:2.2.3.RELEASE:compile
[INFO] | +- org.springframework.boot:spring-boot-starter:jar:2.2.3.RELEASE:compile
[INFO] | | +- org.springframework.boot:spring-boot:jar:2.2.3.RELEASE:compile
[INFO] | | +- org.springframework.boot:spring-boot-autoconfigure:jar:2.2.3.RELEASE:compile
[INFO] | | +- org.springframework.boot:spring-boot-starter-logging:jar:2.2.3.RELEASE:compile
[INFO] | | | +- ch.qos.logback:logback-classic:jar:1.2.3:compile
[INFO] | | | | \- ch.qos.logback:logback-core:jar:1.2.3:compile
[INFO] | | | +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.12.1:compile
[INFO] | | | | \- org.apache.logging.log4j:log4j-api:jar:2.12.1:compile
[INFO] | | | \- org.slf4j:jul-to-slf4j:jar:1.7.30:compile
[INFO] | | +- jakarta.annotation:jakarta.annotation-api:jar:1.3.5:compile
[INFO] | | \- org.yaml:snakeyaml:jar:1.25:runtime
[INFO] | +- org.springframework.boot:spring-boot-actuator-autoconfigure:jar:2.2.3.RELEASE:compile
[INFO] | | +- org.springframework.boot:spring-boot-actuator:jar:2.2.3.RELEASE:compile
[INFO] | | +- com.fasterxml.jackson.core:jackson-databind:jar:2.10.2:compile
[INFO] | | | +- com.fasterxml.jackson.core:jackson-annotations:jar:2.10.2:compile
[INFO] | | | | \- com.fasterxml.jackson.core:jackson-core:jar:2.10.2:compile
[INFO] | | | \- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar:2.10.2:compile
[INFO] | | \- io.micrometer:micrometer-core:jar:1.3.2:compile
[INFO] | |   +- org.hdrhistogram:HdrHistogram:jar:2.1.11:compile
[INFO] | |   | \- org.latencyutils:LatencyUtils:jar:2.0.3:compile
[INFO] | | +- org.springframework.boot:spring-boot-starter-web:jar:2.2.3.RELEASE:compile
[INFO] | | | +- org.springframework.boot:spring-boot-starter-json:jar:2.2.3.RELEASE:compile
[INFO] | | | | +- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:jar:2.10.2:compile
[INFO] | | | | \- com.fasterxml.jackson.module:jackson-module-parameter-names:jar:2.10.2:compile
[INFO] | | | +- org.springframework.boot:spring-boot-starter-tomcat:jar:2.2.3.RELEASE:compile
[INFO] | | | | +- org.apache.tomcat.embed:tomcat-embed-core:jar:9.0.30:compile
[INFO] | | | | \- org.apache.tomcat.embed:tomcat-embed-el:jar:9.0.30:compile
[INFO] | | | \- org.apache.tomcat.embed:tomcat-embed-websocket:jar:9.0.30:compile
[INFO] | | +- org.springframework.boot:spring-boot-starter-validation:jar:2.2.3.RELEASE:compile
[INFO] | | | +- jakarta.validation:jakarta.validation-api:jar:2.0.2:compile
[INFO] | | | \- org.hibernate.validator:hibernate-validator:jar:6.0.18.Final:compile
[INFO] | | |   +- org.jboss.logging:jboss-logging:jar:3.4.1.Final:compile
[INFO] | | |   | \- com.fasterxml:classmate:jar:1.5.1:compile
[INFO] | | | \- org.springframework:spring-web:jar:5.2.3.RELEASE:compile
[INFO] | | \- org.springframework:spring-webmvc:jar:5.2.3.RELEASE:compile
[INFO] | |   \- org.springframework:spring-expression:jar:5.2.3.RELEASE:compile
[INFO] +- org.springframework.boot:spring-boot-starter-test:jar:2.2.3.RELEASE:test
[INFO] | +- org.springframework.boot:spring-boot-test:jar:2.2.3.RELEASE:test
[INFO] | | +- org.springframework.boot:spring-boot-test-autoconfigure:jar:2.2.3.RELEASE:test
[INFO] | | \- org.springframework.boot:spring-boot-test:jar:2.2.3.RELEASE:test
```

Figure 2.15 Dependency tree for the licensing service. The dependency tree is going to display all of the dependencies declared and used in the service.

2.5.3 Booting your Spring Boot application: Writing the Bootstrap class

Our goal is to get a simple microservice up and running in Spring Boot and then iterate on it to deliver functionality. To this end, you need to create two classes in your licensing service microservice:

- A Spring Bootstrap class that will be used by Spring Boot to start up and initialize the application.
- A Spring Controller class that will expose the HTTP endpoints that can be invoked on the microservice.

As you'll see shortly, Spring Boot uses annotations to simplify setting up and configuring the service. This becomes evident as you look at the bootstrap class in the code listing 2.3. This bootstrap class is in the LicenseServiceApplication.java file located in src/main/java/com/optimagrowth/license.

Listing 2.3 Introducing the @SpringBootApplication annotation

```
package com.optimagrowth.license;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication #A
public class LicenseServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(LicenseServiceApplication.class, args); #B
    }
}
```

#A @SpringBootApplication tells the Spring Boot framework that this is the bootstrap class for the project.

```
#B Call to start the entire Spring Boot service
```

The first thing to note in this code is the use of the `@SpringBootApplication` annotation. Spring Boot uses this annotation to tell the Spring container that this class is the source of bean definitions for use in Spring. In a Spring Boot application, you can define Spring Beans by

NOTE A Spring Bean is an object that the Spring Framework manages at runtime by the Inversion of Control (IoC) container. They are created and added to a “repository of objects” from where you can get them later.

1. Annotating a Java class with a `@Component`, `@Service`, or `@Repository` annotation tag.
2. Annotating a class with a `@Configuration` tag and then defining a factory method for each Spring Bean you want to build with a `@Bean` tag.

Under the covers, the `@SpringBootApplication` annotation marks the `Application` class in listing 2.3, as a configuration class, then begins auto-scanning all the classes on the Java classpath for other Spring beans.

The second thing to note is the `LicenseServiceApplication` class's `main()` method. In the `main()` method, the `SpringApplication.run(LicenseServiceApplication.class, args)`, the call starts the Spring container and returns a Spring `ApplicationContext` object. (We aren't doing anything with the `ApplicationContext`, so it isn't shown in the code).

The easiest thing to remember about the `@SpringBootApplication` annotation and the corresponding `LicenseServiceApplication` class is that it's the bootstrap class for the entire microservice. Core initialization logic for the service should be placed in this class.

Now that we know how to create the skeleton and the bootstrap class of our microservices, let's continue with the next chapter. In the next chapter, I will explain some of the critical roles we must consider while we are building a microservice and how those roles are involved in the creation of the O-stock scenario. Also, I will explain some additional technologies to make our microservices more flexible and robust.

2.6 Summary

- Spring Cloud is a collection of open source technologies from companies such as Netflix and HashiCorp that have been “wrapped” with Spring annotations to significantly simplify the setup and configuration of these services.
- Cloud-native applications are built with scalable components like containers, deployed as microservices and managed on virtual infrastructures through DevOps processes with continuous delivery workflows.
- DevOps is the acronym for development (Dev) and operations (Ops), which refers to a software development methodology that focuses on communication, collaboration and integration between software developers and IT operations with the primary goal of automating the process of software delivery and infrastructure changes at lower costs.
- The twelve factor application manifesto document, framed by Heroku, provides twelve best practices you should always keep in the back of your mind when building cloud-native microservices.
- The best practices of the twelve factor application manifesto are: Codebase, dependencies, config, backing services, build release run, processes, port binding, concurrency, disposability, dev/prod parity, logs, admin processes.

- The Spring Initializr allows you to create a new Spring Boot project with the possibility to choose dependencies from an extensive list.
- Spring Boot is the ideal framework for building microservices because it lets you build a REST-based JSON service with a few simple annotations.

3 Building microservices with Spring Boot

This chapter covers

- Understanding how microservices fit into a cloud architecture
- Decomposing a business domain into a set of microservices
- Understanding the perspectives for building microservice-based applications
- Learning when not to use microservices
- Implementing a microservice using Spring Boot, Spring Actuator, Spring Hypermedia as the Engine of Application State (HATEOAS), and Internationalization

To successfully design and build microservices, you need to approach microservices as if you're a police detective interviewing witnesses to a crime. Even though every witness saw the same events take place, their interpretation of the crime is shaped by their background, what was important to them (for example, what motivates them), and what environmental pressures were brought to bear at the moment they witnessed the event. Participants each have their own perspectives (and biases) of what they consider essential.

Like successful police detectives trying to get the truth, the journey to build a successful microservice architecture involves incorporating the perspectives of multiple individuals within your software development organization. Although it takes more than technical people to deliver an entire application, I believe that the foundation for successful microservice development starts with the perspectives of three critical roles:

- **The architect**— The architect's job is to see the big picture and decompose an application into individual microservices and see how the microservices will interact to deliver a solution.
- **The software developer**— The software developer writes the code and understands in detail how the language and development frameworks for the language will be used to deliver a microservice.
- **The DevOps engineer**— The DevOps engineer brings intelligence to how the services are deployed and managed throughout not only production but also all the non-production environments. The watchwords for the DevOps engineer are consistency and repeatability in every environment.

In this chapter I will demonstrate how to design and build a set of microservices from the perspective of each of these roles. This chapter will give you the foundation you need to identify potential microservices within your business application, and then understand the operational attributes that need to be in place for a microservice to be deployed. By the time the chapter concludes, you'll have a service that can be packaged and deployed to the cloud using the skeleton project we created in the previous chapter.

3.1 The architect's story: designing the microservice architecture

An architect's role in a software project is to provide a working model of the problem that needs to be solved. The job of the architect is to provide the scaffolding against which developers will build their code so that all the pieces of the application fit together.

When building a microservices architecture, a project's architect focuses on three key tasks:

1. Decomposing the business problem
2. Establishing service granularity
3. Defining the service interfaces

3.1.1 Decomposing the business problem

In the face of complexity, most people try to break the problem on which they're working into manageable chunks. They do this, so they don't have to try to fit all the details of the problem in their heads. Instead, they break the problem down abstractly into a few essential parts and then look for the relationships that exist between these parts.

In a microservices architecture, the architect breaks the business problem into chunks that represent discrete domains of activity. These chunks encapsulate the business

rules and the data logic associated with a particular part of the business domain.

For example, an architect might look at a business flow that's to be carried out by code and realize that they need both customer and product information. The presence of two discrete data domains is a good indication that multiple microservices are at play. How the two different parts of the business transaction interact usually becomes the service interface for the microservices.

Breaking apart a business domain is an art form rather than a black-and-white science. Use the following guidelines for identifying and decomposing a business problem into microservices candidates:

- 1. Describe the business problem and listen to the nouns you're using to describe the problem.** Using the same nouns over and over in describing the problem is usually an indication of a core business domain and an opportunity for a microservice. Examples of target nouns for the O-stock application will be something like **contract, licenses, and assets**.
- 2. Pay attention to the verbs.** Verbs highlight actions and often represent the natural contours of a problem domain. If you find yourself saying "transaction X needs to get data from thing A and thing B", that usually indicates that multiple services are at play. If you apply this approach of watching for verbs to the O-stock application we're going to create, you might look for statements such as, "When Mike from desktop services is setting up a new PC, he looks up the number of licenses available for software X and, if licenses are available, installs the software. He then updates the number of licenses used in his tracking spreadsheet." The key verbs here are looks and updates.

3. Look for data cohesion. As you break apart your business problem into discrete pieces, look for pieces of data that are highly related to one another. If suddenly, during the course of your conversation, you're reading or updating data that is radically different from what you've been discussing so far, you potentially have another service candidate. **Microservices must completely own their data.**

Let's take these guidelines and apply them to a real-world problem, such as the one I previously described of the O-stock software that is used for managing software assets.

O-stock is a traditional monolithic web application that is deployed to a J2EE application server residing within a customer's data center. Our goal is to tease apart the existing monolithic application into a set of services. To achieve this, we are going to start by interviewing all the users and some of the business stakeholders of the O-stock application and discussing with them how they interact and use the application. Figure 3.1 captures a summary and highlights a number of nouns and verbs of the conversations with different business customers.

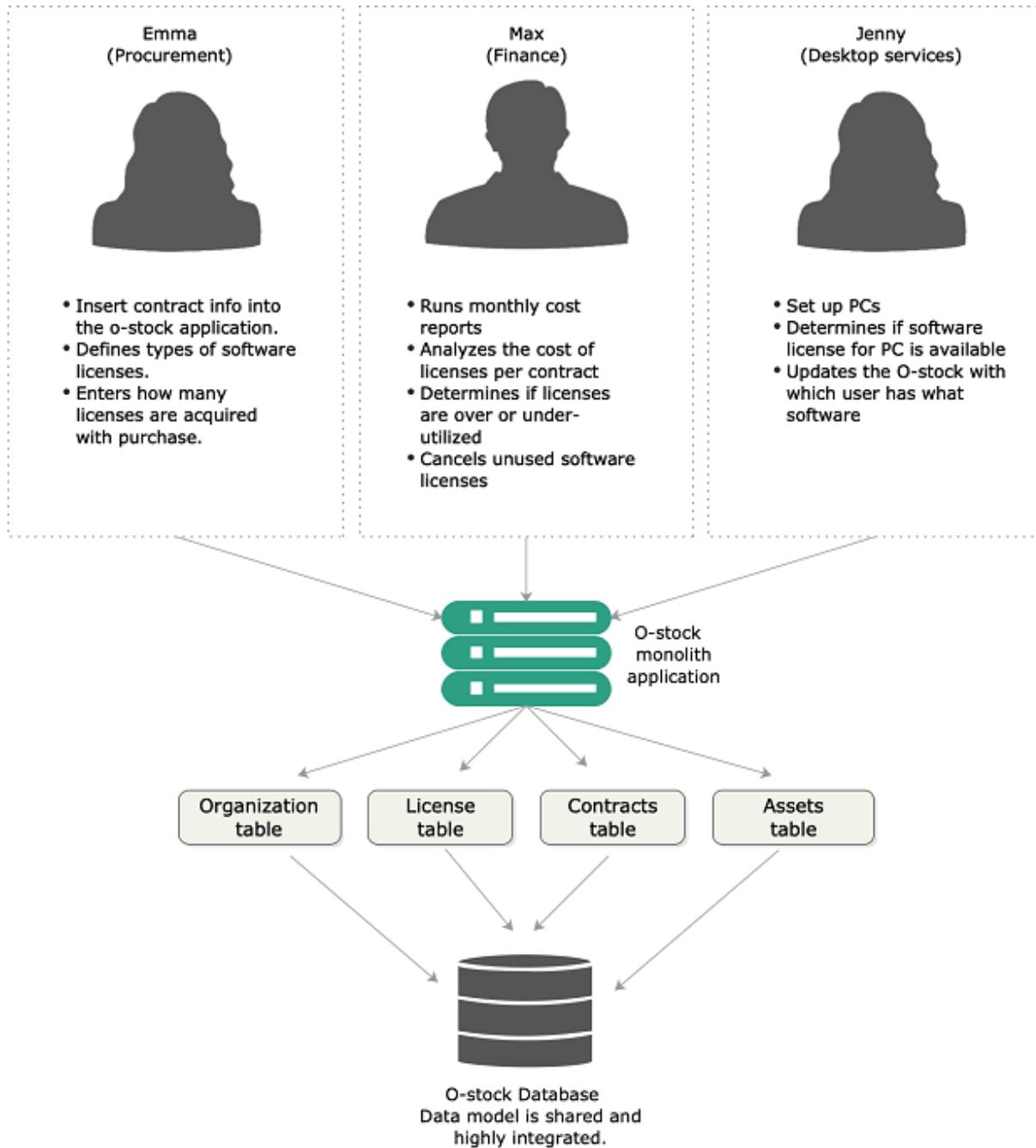


Figure 3.1 Interview the O-stock users and understand how they do their day-to-day work.

By looking at how the users of O-stock interact with the application and answering the following questions. We can

identify the data model for the application, and we can decompose the O-stock problem domain into the following microservice candidates.

- Where are we going to store the contract info managed by Emma?
- Where are we going to store the license information? We need to store, and manage the cost, the type of license, the owner of the license and the contract of the license.
- Jenny works by setting up the licenses on the PC's. Are we going to store the PCs or in this case, we call it Assets information?
- Taking into consideration all the previously mentioned concepts, we can see that the license belongs to an organization that has several assets, right? So, where are we going to store the organization information?

Figure 3.2 shows a simplified data model based on the conversations with the customers. Based on the business interviews and the data model, the microservice candidates are **organization**, **license**, **contract** and **asset**.

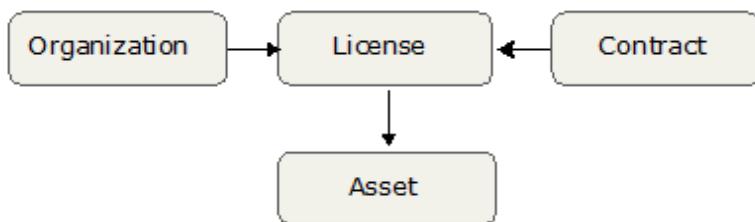


Figure 3.2 A simplified O-stock data model. An organization can have many licenses, the license can be applied to one or several assets, and each license has a contract.

3.1.2 Establishing service granularity

Once we have a simplified data model, we can begin the process of defining what microservices we are going to need in the application. Based on the data model in figure 3.2, we can see the potential four microservices based on the following elements.

- Assets
- License
- Contract
- Organization

The goal is to take these major pieces of functionality and extract them into entirely self-contained units that can be built and deployed independently of each other, they can optionally share or have individual databases. However, extracting services from the data model involves more than repackaging code into separate projects. It also involves teasing out the actual database tables the services are accessing and only allowing each service to access the tables in its specific domain. Figure 3.3 shows how the application code and the data model become “chunked” into individual pieces.

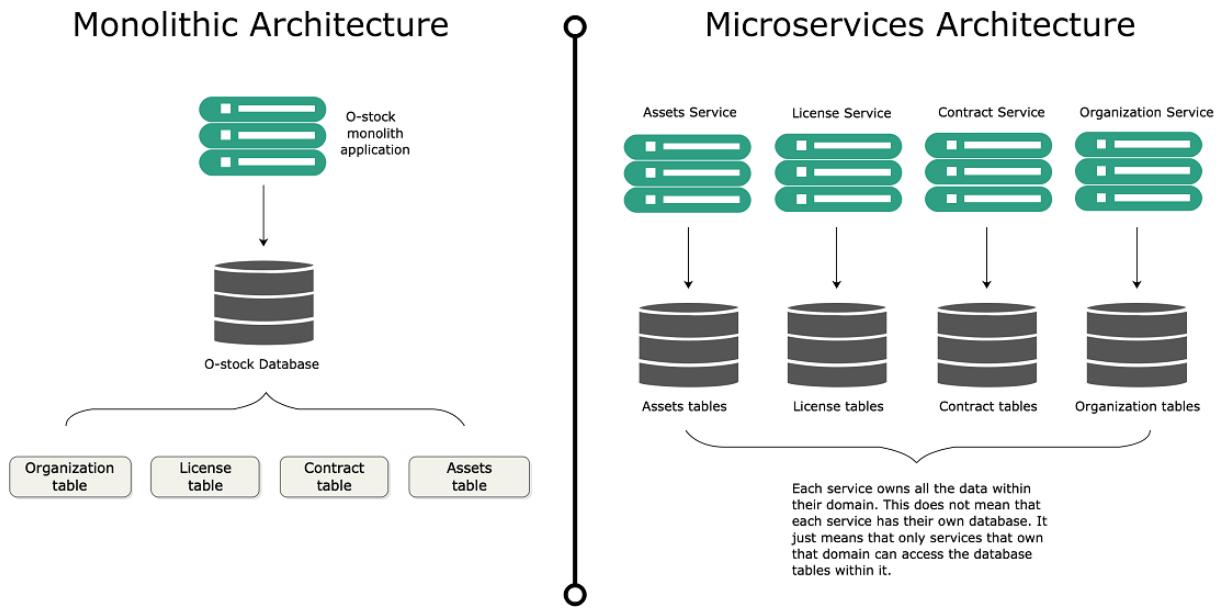


Figure 3.3 The O-stock application is broken down from a monolithic application into smaller individual services that are deployed independently of one another.

NOTE In the previous example I created individual databases for each service, but also you can share databases between the services.

After we have broken a problem domain down into discrete pieces, we will often find ourselves struggling to determine whether we've achieved the right level of granularity for our services. A microservice that is too coarse or fine-grained will have several telltale attributes that we'll discuss shortly.

When we are building a microservice architecture, the question of do we have the right granularity is essential; this is why I want to explain the following concepts to determine the correct answer to that question:

- 1. It's better to start broad with our microservice and refactor to smaller services.** It is easy to go overboard when you begin your microservice journey and make everything a microservice. But decomposing the problem domain into small services often leads to premature complexity because microservices devolve into nothing more than fine-grained data services.
- 2. Focus first on how our services will interact with one another.** This will help establish the coarse-grained interfaces of your problem domain. It is easier to refactor from being too coarse-grained than too fine-grained.
- 3. Service responsibilities will change over time as our understanding of the problem domain grows.** Often, a microservice gains responsibilities as new application functionalities are requested. What starts as a single microservice might grow into multiple services, with the original microservice acting as an orchestration layer for these new services and encapsulating their functionality from other parts of the application.

The smells of a bad microservice

How do you know whether your microservices are the right size? If a microservice is too coarse-grained, you'll likely see the following:

A service with too many responsibilities. The general flow of the business logic in the service is complicated and seems to be enforcing an overly diverse array of business rules.

The service is managing data across a large number of tables. A microservice is the system of record for the data it manages. If you find yourself persisting data to multiple tables or reaching out to tables outside of the service database, this is a clue the service is too big. I like to use the guideline that a microservice should own no more than three to five tables. Any more, and your service is likely to have too much responsibility.

Too many test cases. Services can grow in size and responsibility over time. If you have a service that started with a small number of test cases and ends up with hundreds of unit and integration test cases, you might need to refactor. What about a microservice that's too fine-grained?

The microservices in one part of the problem domain breed like rabbits. If everything becomes a microservice, composing business logic out of the services becomes complex and difficult because the number of services needed

to get a piece of work done grows tremendously. A common smell is when you have dozens of microservices in an application and each service interacts with only a single database table.

Your microservices are heavily interdependent on one another. You find that the microservices in one part of the problem domain keep calling back and forth between each other to complete a single user request.

Your microservices become a collection of simple CRUD (Create,

Replace, Update, Delete) services. Microservices are an expression of business logic and not an abstraction layer over your data sources. If your microservices do nothing but CRUD-related logic, they're probably too fine-grained.

A microservices architecture should be developed with an evolutionary thought process where you know that you aren't going to get the design right the first time. That is why it's better to start with your first set of services being more coarse-grained than fine-grained. It is also essential not to be dogmatic with your design. You may run into physical constraints on your services where you'll need to make an aggregation service that joins data together because two separate services will be too chatty, or where no clear boundaries exist between the domain lines of a service.

In the end, take a pragmatic approach and deliver, rather than waste time trying to get the design perfect and then have nothing to show for your effort.

3.1.3 Defining the service interfaces

The last part of the architect's input is about defining how the microservices in your application are going to talk with one another. When building business logic with

microservices, the interfaces for the services should be intuitive, and developers should get a rhythm of how all the services work in the application by learning one or two of the services in the application.

In general, the following guidelines can be used for thinking about service interface design:

1. **Embrace the REST philosophy.** This is one of the best practices I describe in Appendix A with the Richardson Level of maturity. The REST approach to service is at heart the embracing of HTTP as the invocation protocol for the services and the use of standard HTTP verbs (GET, PUT, POST, and DELETE). Model your basic behaviors around these HTTP verbs.
2. **Use URI's to communicate intent.** The URI you use as endpoints for the service should describe the different resources in your problem domain and provide a basic mechanism for relationships of resources within your problem domain.
3. **Use JSON for your requests and responses.** JavaScript Object Notation (in other words, JSON) is an extremely lightweight data serialization protocol and is much easier to consume than XML.
4. **Use HTTP status code to communicate results.** The HTTP protocol has a rich body of standard response codes to indicate the success or failure of a service. Learn these status codes and most importantly use them consistently across all your services.

All the basic guidelines drive to one thing, making your service interfaces easy to understand and consumable. You want a developer to sit down and look at the service interfaces and start using them. If a microservice isn't easy to consume, developers will go out of their way to work around and subvert the intention of the architecture.

3.2 When not to use microservices

We've spent this chapter talking about why microservices are a powerful architectural pattern for building applications. But I haven't touched on when you shouldn't use microservices to build your applications. Let's walk through them:

1. Complexity building distributed systems
2. Virtual server / container sprawl
3. Application type
4. Data transactions and consistency

3.2.1 Complexity of building distributed systems

Because microservices are distributed and fine-grained (small), they introduce a level of complexity into your application that wouldn't be there in more monolithic applications. Microservice architectures require a high degree of operational maturity. Don't consider using microservices unless your organization is willing to invest in the automation and operational work (monitoring, scaling, and more) that a highly distributed application needs to be successful.

3.2.2 Server sprawl

One of the most common deployment models for microservices is to have one microservice instance deployed

on one container. In a large microservices-based application, you might end up with 50 to 100 servers or containers (usually virtual) that have to be built and maintained in production alone. Even with the lower cost of running these services in the cloud, the operational complexity of having to manage and monitor these services can be tremendous.

NOTE The flexibility of microservices has to be weighed against the cost of running all of these servers. You can also have different alternatives such as considering functional developments such as lambdas or adding more microservices instances on the same server.

3.2.3 Type of application

Microservices are geared toward reusability and are extremely useful for building large applications that need to be highly resilient and scalable. This is one of the reasons why so many cloud-based companies have adopted microservices. If you're building small, departmental-level applications or applications with a small user base, the complexity associated with building on a distributed model such as microservice might generate more expenses than it's worth.

3.2.4 Data transactions and consistency

As you begin looking at microservices, you need to think through the data usage patterns of your services and how service consumers are going to use them. A microservice wraps around and abstracts away a small number of tables and works well as a mechanism for performing “operational”

tasks such as creating, adding, and performing simple (non-complex) queries against a store.

If your applications need to do complex data aggregation or transformation across multiple sources of data, distributed nature of microservices will make this work difficult. Your microservices will invariably take on too much responsibility and can also become vulnerable to performance problems.

3.3 The developer's tale: building a microservice with Spring Boot and Java

In this section, we'll explore the developer's priorities in building out the licensing microservice from the o-stock domain model.

NOTE Remember, we created the skeleton of the licensing service in the previous chapter. In case you didn't follow the previous code listings. You can download the source code from the following link.
(<https://github.com/ihuaylupo/manning-smia/tree/master/chapter2>).

Over the next sections, you're going to

1. Implement a Spring Boot controller class for mapping an endpoint to expose the endpoints of the licensing service.
2. Implement Internationalization so that the messages can be adapted to different languages.
3. Implement Spring HATEOAS to provide enough information to the user to interact with the server.

3.3.1 Building the doorway into the microservice: The Spring Boot controller

Now that we've gotten the build script out of the way and implemented a simple Spring Boot Bootstrap class, you can begin writing your first code that will do something.

This code will be your Controller class. In a Spring Boot application, a Controller class exposes the services endpoints and maps the data from an incoming HTTP request to a Java method that will process the request.

Give it a REST

All the microservices in this book follow the levels of the Richardson Level of maturity explained in the previous chapter. All the services you build will have the following characteristics:

Use HTTP/HTTPS as the invocation protocol for the service—The service will be exposed via HTTP endpoint and will use the HTTP protocol to carry data to and from the services.

Map the behavior of the service to standard HTTP verbs—REST emphasizes having services map their behavior to the HTTP verbs of POST, GET, PUT, and DELETE verbs. These verbs map to the CRUD functions found in most services. Use JSON as the serialization format for all data going to and from the service—This isn't a hard-and-fast principle for REST-based microservices, but JSON has become lingua franca for serializing data that's going to be submitted and returned by a microservice. XML can be used, but many REST-based applications make heavy use of JavaScript and JSON (JavaScript Object Notation). JSON is the native format for serializing and deserializing data being consumed by JavaScript-based web front-ends and services.

Use HTTP status codes to communicate the status of a service call—The HTTP protocol has developed a rich set of status codes to indicate the success or failure of a service. REST-based services take advantage of these HTTP status codes and other web-based infrastructure, such as reverse proxies and caches, which can be integrated with your microservices with relative ease.

HTTP is the language of the web, and using HTTP as the philosophical framework for building your service is a key to building services in the cloud.

Your first controller class should be located in `src/main/java/com/optimagrowth/license/controller/LicenseController.java`. This class will expose four HTTP endpoints that will map to the POST, GET, PUT, DELETE verbs.

Let's walk through the controller class and look at how Spring Boot provides a set of annotations that keeps the effort needed to expose your service endpoints to a minimum and allows you to focus on building the business logic for the service. We'll start by looking at the basic controller class definition without any class methods in it yet. Code listing 3.1 shows the controller class for the licensing service.

Listing 3.1 Marking the LicenseServiceController as a Spring RestController

```
package com.optimagrowth.license.controller;
import java.util.Random;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.optimagrowth.license.model.License;
@RestController #A
@RequestMapping(value="v1/organization/{organizationId}/license") #B
public class LicenseController { }
```

#A @RestController tells Spring Boot; this is a REST-based service and will automatically serialize/deserialize service request/response JSON.

#B Exposes all the HTTP endpoints in this class with a prefix of /v1/organizations/{organizationId}/license

We'll begin our exploration by looking at the @RestController annotation. The @RestController is a class-level Java annotation that tells the Spring Container that this java class is going to be used for a REST-based service. This annotation automatically handles the serialization of data passed into the services as JSON or XML (by default, the @RestController class will serialize returned data into JSON). Unlike the traditional Spring @Controller annotation, the @RestController annotation doesn't require you as a developer to return a ResponseBody class from your method in the controller class. This is all handled by the presence of the @RestController annotation, which includes the @ResponseBody annotation.

Why JSON for microservices?

Multiple protocols can be used to send data back and forth between HTTP-based microservices. But, JSON has emerged as the de facto standard for several reasons.

First, compared to other protocols such as the XML-based SOAP (Simple Object Access Protocol), it's extremely lightweight in that you can express your data without having much textual overhead.

Second, it's easily read and consumed by a human being. This is an underrated quality for choosing a serialization protocol. When a problem arises, it's critical for developers to look at a chunk of JSON and quickly, visually process what's in it. The simplicity of the protocol makes this incredibly easy to do.

Third, JSON is the default serialization protocol used in JavaScript. Since the dramatic rise of JavaScript as a programming language and the equally dramatic rise of Single Page Internet Applications (SPIA) that rely heavily on JavaScript, JSON has become a natural fit for building REST-based applications because it's what the front-end web clients use to call services.

Other mechanisms and protocols are more efficient than JSON for communicating between services. The Apache Thrift (<http://thrift.apache.org>) framework allows you to build multi-language services that can communicate with one another using a binary protocol. The Apache Avro protocol (<http://avro.apache.org>) is a data serialization protocol that converts data back and forth to a binary format between client and server calls.

If you need to minimize the size of the data you're sending across the wire, I recommend you look at these protocols. But it has been my experience that using straight-up JSON in your microservices works effectively and doesn't interpose another layer of communication to debug between your service consumers and service clients.

The second annotation shown in listing 3.1 is the `@RequestMapping` annotation. You can use the `@RequestMapping` annotation as a class-level and method-level annotation and is used to tell the Spring container the HTTP endpoint that the service is going to expose to the user. When you use the class-level annotation, you're establishing the root of the URL for all the other endpoints exposed by the controller.

In listing 3.1, the `@RequestMapping(value="v1/organization/{organizationId}/license")` uses the `value` attribute to establish the root of the URL for all endpoints exposed in the controller class. All service endpoints exposed in this controller will start with `v1/organization/{organizationId}/license` as the root of their endpoint. The `{organizationId}` is a placeholder that indicates how you expect the URL to be parametrized with an `organizationId` passed in every call. The use of `organizationId` in the URL allows you to differentiate between the different customers who might use your service.

Before we add the first method to the controller, let's explore the model and the service class that we are going to be using in the services we're about to create.

Code Listing 3.2 shows the POJO class that encapsulates the license data.

NOTE Remember, encapsulation is one of the main principles of object-oriented programming, and in order to achieve encapsulation in Java, we must declare the variables of a class as private and then provide public getters and setters to read and write the values of those variables.

Listing 3.2 Exploring the License Model

```
package com.optimagrowth.license.model;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
@Getter @Setter @ToString
public class License { #A
    private int id;
    private String licenseId;
    private String description;
    private String organizationId;
    private String productName;
    private String licenseType;
}
```

#A Java Plain Old Java Object (POJO) that contains the license Info.

Lombok

Lombok is a small library that allows us to reduce the amount of boilerplate Java code written in the java classes of our project. Lombok generates code such as getters, setters, to string methods, constructors, and more.

In this book, I will be using Lombok throughout the code examples in order to keep the code more readable, but I will not get into details of how to use it. So, if you are interested in knowing more about this tool, I highly recommend you look at the following Baeldung.com articles. (<https://www.baeldung.com/intro-to-project-lombok>) and (<https://www.baeldung.com/lombok-ide>).

In case you want to install Lombok on the Spring Tool Suite 4, you must download and execute the Lombok and link it to the IDE.

Code Listing 3.3 shows the service class that we are going to use to develop the logic of the different services we are going to create on the Controller class.

Listing 3.3 Exploring the license service class

```
package com.optimagrowth.license.service;
import java.util.Random;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import com.optimagrowth.license.model.License;
@Service
public class LicenseService {
    public License getLicense(String licenseId, String organizationId){
        License license = new License();
        license.setId(new Random().nextInt(1000));
        license.setLicenseId(licenseId);
        license.setOrganizationId(organizationId);
        license.setDescription("Software product");
        license.setProductName("Ostock");
        license.setLicenseType("full");
        return license;
    }
    public String createLicense(License license, String organizationId){
        String responseMessage = null;
        if(license != null) {
            license.setOrganizationId(organizationId);
            responseMessage = String.format("This is the post and the object is: %s",
                license.toString());
        }
        return responseMessage;
    }
    public String updateLicense(License license, String organizationId){
        String responseMessage = null;
        if(!StringUtils.isEmpty(license)) {
            license.setOrganizationId(organizationId);
            responseMessage = String.format("This is the put and the object is: %s",
                license.toString());
        }
        return responseMessage;
    }
    public String deleteLicense(String licenseId, String organizationId){
        String responseMessage = null;
        responseMessage = String.format("Deleting license with id %s for the organization
        %s", licenseId, organizationId);
        return responseMessage;
    }
}
```

This service class contains a set of dummy service returning, some hard-coded data to give you an idea of how the skeleton a microservice should look. As you continue reading the book, you'll continue working on this service and delve further into how to structure it.

Now, let's add the first method to the controller. This method will implement the GET verb used in a REST call and return a single License class instance, as shown in the 3.4 code listing.

Listing 3.4 Exposing an individual GET HTTP endpoint

```
package com.optimagrowth.license.controller;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.optimagrowth.license.model.License;
import com.optimagrowth.license.service.LicenseService;
@RestController
@RequestMapping(value="v1/organization/{organizationId}/license")
public class LicenseController {
@.Autowired
private LicenseService licenseService;
@GetMapping(value="/{licenseId}") #A
public ResponseEntity<License> getLicense( @PathVariable("organizationId") String
organizationId, @PathVariable("licenseId") String licenseId) { #B
License license = licenseService.getLicense(licenseId, organizationId);
return ResponseEntity.ok(license); #C
}
}
```

#A Get method to retrieve a license data
#B Maps two parameters from the URL (organizationId and licenseld) to method parameters
#C Response Entity to represent the entire HTTP Response

The first thing we've done in this listing is annotate the `getLicense()` method with a method level `@RequestMapping`

annotation, passing in two parameters to the annotation. Value and method. With a method-level @RequestMapping annotation, you're building the following endpoint v1/organization/{organizationId}/license/{licenseId} for the getLicense() method. Why? If we go back and take a look, there is a root-level annotation specified at the top of the class to match all HTTP requests coming to the controller, so first, we add the root-level annotation value and then the method-level value. The second parameter of the annotation, method, specifies the HTTP verb that the method will be matched on. In the getLicense() method, we are matching on the GET method as presented by the RequestMethod.GET enumeration.

The second thing to note about listing 3.4 is that we use the @PathVariable annotation in the parameter body of the getLicense() method. The @PathVariable annotation is used to map the parameter values passed in the incoming URL (as denoted by the {parameterName} syntax) to the parameters of your method. In the code listing 3.4 for the GET service, we are mapping two parameters from the URL, organizationId, and licenseId, to two parameter-level variables in the method:

```
@PathVariable("organizationId") String organizationId,  
@PathVariable("licenseId") String licenseId
```

The third thing to note about the 3.4 code listing is the ResponseEntity return object. The ResponseEntity represents the entire HTTP Response, including the status code, the headers, and the body. In the previous example, it allowed us

to return the License object as the body and the 200(OK) status code as the HTTP Response of the service.

Now that you understood how to create an endpoint using the HTTP verb let's continue by adding POST, PUT, and DELETE methods to create, update, and delete License class instances.

Listing 3.5 Exposing individual HTTP Verbs endpoints

```
package com.optimagrowth.license.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.optimagrowth.license.model.License;
import com.optimagrowth.license.service.LicenseService;
@RestController
@RequestMapping(value="v1/organization/{organizationId}/license")
public class LicenseController {
    @Autowired
    private LicenseService licenseService;
    @RequestMapping(value="/{licenseId}",method = RequestMethod.GET)
    public ResponseEntity<License> getLicense( @PathVariable("organizationId") String organizationId,
                                                @PathVariable("licenseId") String licenseId) {
        License license = licenseService.getLicense(licenseId, organizationId);
        return ResponseEntity.ok(license);
    }
    @PutMapping #A
    public ResponseEntity<String> updateLicense(@PathVariable("organizationId") String organizationId, @RequestBody License request) { #B
        return ResponseEntity.ok(licenseService.updateLicense(request, organizationId));
    }
    @PostMapping #C
    public ResponseEntity<String> createLicense(@PathVariable("organizationId") String organizationId, @RequestBody License request) {
        return ResponseEntity.ok(licenseService.createLicense(request, organizationId));
    }
    @DeleteMapping(value="/{licenseId}") #D
    public ResponseEntity<String> deleteLicense(@PathVariable("organizationId") String organizationId, @PathVariable("licenseId") String licenseId) {
        return ResponseEntity.ok(licenseService.deleteLicense(licenseId, organizationId));
    }
}
```

```
#A Put method to update a license  
#B Maps the HTTP Request body to a License object  
#C Post method to insert a license  
#D Delete method to delete a license
```

The first thing we've done in listing 3.5 is annotate the `updateLicense()` method with a method level `@PutMapping` annotation. The `@PutMapping` annotation is a method-level annotation that acts as a shortcut for the `@RequestMapping(method = RequestMethod.PUT)` annotation.

The second thing to note about the code listing 3.5 is that we use the `@PathVariable` annotation and the `@RequestBody` annotation in the parameter body of the `updateLicense()` method. The `@RequestBody` annotation is used to map the `HTTPRequest` body to a transfer object, in this case, the `License` object. In the `updateLicense()` we are mapping two parameters, one from the URL and the other of the `HTTPRequest` body, to two parameter-level variables in the method:

```
@PathVariable("organizationId") String organizationId,  
@RequestBody License request
```

The third and fourth thing to note about the code listing 3.5 is that we use the `@PostMapping` and the `@DeleteMapping` annotations. The `@PostMapping` annotation is a method-level annotation that acts as a shortcut for the `@RequestMapping(method = RequestMethod.POST)` and the `@DeleteMapping(value="/{licenseId}")` is also a method-level annotation and acts as a shortcut for the

`@RequestMapping(value="/{licenseId}",method = RequestMethod.DELETE)` annotation.

Endpoint names matter

Before you get too far down the path of writing microservices, make sure that you (and potentially other teams in your organization) establish standards for the endpoints that will be exposed via your services. The URLs (Uniform Resource Locator) for the microservice should be used to clearly communicate the intent of the service, the resources the service manages, and the relationships that exist between the resources managed within the service. I've found the following guidelines useful for naming service endpoints:

1. Use clear URL names that establish what resource the service represents—Having a canonical format for defining URLs will help your API feel more intuitive and easier to use. Be consistent in your naming conventions.
2. Use the URL to establish relationships between resources—Oftentimes you'll have a parent-child relationship between resources within your microservices where the child doesn't exist outside the context of the parent (hence you might not have a separate microservice for the child). Use the URLs to express these relationships. But if you find that your URLs tend to be excessively long and nested, your microservice may be trying to do too much.
3. Establish a versioning scheme for URLs early—The URL and its corresponding endpoints represent a contract between the service owner and consumer of the service. One common pattern is to prepend all endpoints with a version number. Establish your versioning scheme early and stick to it. It's extremely difficult to retrofit versioning to URLs after you already have several consumers using them. For example using the /v1/ in the URL mappings.

At this point, you have several services. From a command-line window, go to your project root directory where the pom.xml is and execute the following Maven command:

```
mvn spring-boot:run
```

Figure 3.4 shows the expected output of the mvn spring-boot:run command

```
c.o.license.LicenseServiceApplication      : No active profile set, falling back to default profiles: default
o.s.b.w.embedded.tomcat.TomcatWebServer   : Tomcat initialized with port(s): 8080 (http)
o.apache.catalina.core.StandardService     : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/9.0.30]
o.a.c.c.C.[Tomcat].[localhost].[/]         : Initializing Spring embedded WebApplicationContext
o.s.web.context.ContextLoader             : Root WebApplicationContext: initialization completed in 662 ms
o.s.s.concurrent.ThreadPoolTaskExecutor   : Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.a.e.web.EndpointLinksResolver       : Exposing 1 endpoint(s) beneath base path '/v1'
o.s.b.w.embedded.tomcat.TomcatWebServer   : Tomcat started on port(s): 8080 (http) with context path ''
c.o.license.LicenseServiceApplication      : Started LicenseServiceApplication in 1.732 seconds (JVM running for 1.969)
```

Figure 3.4 The licensing service starting successfully.

Once the service is started, you can directly hit the exposed endpoints. I highly recommend using a chrome-based tool like POSTMAN or CURL for calling the service. Figure 3.5 shows how to call the GET and DELETE services on the <http://localhost:8080/v1/organization/optimaGrowth/license/0235431845> endpoint.

The image shows two screenshots of the POSTMAN application interface, demonstrating the use of the `http://localhost:8080/v1/organization/optimaGrowth/license/0235431845` endpoint.

GET Request:

- Method:** GET
- URL:** `http://localhost:8080/v1/organization/optimaGrowth/license/0235431845`
- Headers:** (7)
- Body:** (Pretty, Raw, Preview, Visualize BETA, JSON, Copy icon)
- Response:**

```

1 {
2   "id": 600,
3   "licenseId": "0235431845",
4   "description": "Software product",
5   "organizationId": "optimaGrowth",
6   "productName": "Ostock",
7   "licenseType": "full"
8 }
```

A callout box points from the response body to the following note: "When the GET endpoint is called, a JSON payload containing licensing data is returned."

DELETE Request:

- Method:** DELETE
- URL:** `http://localhost:8080/v1/organization/optimaGrowth/license/0235431845`
- Headers:** (8)
- Body:** (Pretty, Raw, Preview, Visualize BETA, Text, Copy icon)
- Response:**

```
1 Deleting license with id 0235431845 for the organization optimaGrowth
```

A callout box points from the response body to the following note: "When the DELETE endpoint is called, a string is returned."

Figure 3.5 Your licensing GET and DELETE service being called with POSTMAN.

Figure 3.6 shows how to call the POST and PUT services using the `http://localhost:8080/v1/organization/optimaGrowth/license` endpoint.

POST <http://localhost:8080/v1/organization/optimaGrowth/license> Send

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL BETA JSON

```

1 {
2   "licenseId": "0235431845",
3   "organizationId": "Optima Growth",
4   "description": "Software product",
5   "productName": "Ostock",
6   "licenseType": "complete"
7 }
  
```

Body Cookies Headers (3) Test Results Status: 200 OK Time: 6ms Size: 289 B Save

Pretty Raw Preview Visualize BETA Text

1 This is the post and the object is: License(id=0, licenseId=0235431845, description=Software product, organizationId=optimaGrowth, productName=Ostock, licenseType=complete)

PUT <http://localhost:8080/v1/organization/optimaGrowth/license> Send

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL BETA JSON

```

1 {
2   "licenseId": "0235431845",
3   "organizationId": "Optima Growth",
4   "description": "Software product",
5   "productName": "Ostock",
6   "licenseType": "complete"
7 }
  
```

Body Cookies Headers (3) Test Results Status: 200 OK Time: 32ms Size: 288 B Save

Pretty Raw Preview Visualize BETA Text

1 This is the put and the object is: License(id=0, licenseId=0235431845, description=Software product, organizationId=optimaGrowth, productName=Ostock, licenseType=complete)

Figure 3.6 Your licensing GET and DELETE service being called with POSTMAN.

Once we have already implemented the methods for the Put, Delete, Post and Get HTTP verbs, we can move on with the

internalization section.

3.3.2 Applying internationalization into the licensing service

Internationalization is an essential requirement often asked for the applications that consist of enabling the application to adapt to different languages. The main goal with this is to develop applications that offer content in multiple formats and languages. In this section, I will explain how to add internalization to the licensing service we previously created.

First, we are going to update the Bootstrap Class LicenseServiceApplication.java to create a LocaleResolver and a ResourceBundleMessageSource to our License service. The following code listing 3.6 will show you how the bootstrap class should look.

Listing 3.6 Creating Beans for LocaleResolver and ResourceBundleMessageSource

```
package com.optimagrowth.license;
import java.util.Locale;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
@SpringBootApplication
public class LicenseServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(LicenseServiceApplication.class, args);
    }
    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver localeResolver = new SessionLocaleResolver();
        localeResolver.setDefaultLocale(Locale.US); #A
        return localeResolver;
    }
    @Bean
```

```
public ResourceBundleMessageSource messageSource() {  
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();  
    messageSource.setUseCodeAsDefaultMessage(true); #B  
    messageSource.setBasenames("messages"); #C  
    return messageSource;  
}  
}
```

#A Set US as a default locale
#B Don't throw an error if a message is not found, instead return the message code
#C Set the base name of the languages properties files

The first thing to note on the code listing 3.6 is that we are setting as a default locale the Locale.US. If we don't set a Locale when we are retrieving the message, the messageSource will use the default locale set as the LocaleResolver.

The second thing to note is the messageSource.setUseCodeAsDefaultMessage(true)call. With this option, when a message is not found, it will return the message code 'license.create.message' instead of an error like this one "No message found under code 'license.create.message' for locale 'es'.

The third thing to note in the code listing 3.6 is the messageSource.setBasenames("messages") call. This call will set messages as the base name of the message source files. For example, if we were in Italy, we would use the Locale.IT and we would have a file called messages_it.properties. In case we don't find a message in a specific language, the message source is going to search on the default message file called messages.properties.

Now, let's configure the messages; for this example, I will be using English and Spanish messages. To achieve this, we have to create the following files under the /src/main/resources source folder.

```
messages_en.properties  
messages_es.properties  
messages.properties
```

The following code listing 3.7 and 3.8 will show how the messages_en.properties and the messages_es.properties should look

Listing 3.7 Exploring the messages_en.properties file

```
license.create.message = License created %s  
license.update.message = License %s updated  
license.delete.message = Deleting license with id %s for the organization %s
```

Listing 3.8 Exploring the messages_es.properties file

```
license.create.message = Licencia creada %s  
license.update.message = Licencia %s creada  
license.delete.message = Eliminando licencia con id %s para la organization %s  
license
```

Now that we've implemented the messages and the @Beans we can update the code in our controller and/or service to call the message resource. The following code listing 3.9 will show you how.

Listing 3.9 Updating the service to look for the messages on the Message Source

```
@Autowired
MessageSource messages;
public String createLicense(License license, String organizationId,
Locale locale){ #A
String responseMessage = null;
if(!StringUtils.isEmpty(license)) {
license.setOrganizationId(organizationId);
responseMessage =
String.format(messages.getMessage("license.create.message",null,locale), #B
license.toString());
}
return responseMessage;
}
public String updateLicense(License license, String organizationId){
String responseMessage = null;
if(!StringUtils.isEmpty(license)) {
license.setOrganizationId(organizationId);
responseMessage = String.format(messages.getMessage("license.update.message",
null, null), #C license.toString());
}
return responseMessage;
}
```

#A Receive the Locale as a method parameter

#B Set the received locale to retrieve the specific message

#C Send a null locale to retrieve the specific message

There are three important things to highlight from the previous code. The first one is that we can receive the Locale from the Controller itself. The second one is that we can call the

`messages.getMessage("license.create.message",null,locale)` using the locale we received by parameters, and the third one is that we can call the

`messages.getMessage("license.update.message", null, null)` without sending any locale. In this particular scenario, the application will use the default locale we previously defined in the bootstrap class.

Let's update our `createLicense()` method on the controller to receive the language from the request `Accept-Language` header.

```
@PostMapping  
public ResponseEntity<String> createLicense(  
    @PathVariable("organizationId") String organizationId,  
    @RequestBody License request,  
    @RequestHeader(value = "Accept-Language", required = false) Locale locale){  
    return ResponseEntity.ok(licenseService.  
        createLicense(request, organizationId, locale));  
}
```

A few things to note from this previous code is that we are using the `@RequestHeader` annotation. The `@RequestHeader` annotation is used to map method parameters with request header values. In the `createLicense()` method, we are retrieving the `Locale` from the request header `Accept-Language`. This service parameter is not required, so if it's not specified, we are going to use the default `Locale`.

Figure 3.7 will show you how to send the `Accept-Language` request header from POSTMAN.

Figure 3.7 Set the Accept-language header in the POST create license service.

NOTE For the locale, there isn't a well-defined rule on how to use it; my recommendation is: analyze your architecture and select the option that is more suitable for you. For example, if the frontend application is the one handling the locale, then receiving the locale as a parameter in the controller method is the best option. But if you are managing the locale in the backend, you can use a default locale.

3.3.3 Implementing Spring HATEOAS to display related links

As I previously explained in the previous chapter, HATEOAS stands for Hypermedia as the Engine of Application State). Spring HATEOAS is a small project of Spring that allows us to create APIs that follow the HATEOAS principle of displaying the related links for a given resource. The HATEOAS principle implies that an API should provide a guide to the client by returning possible information about the next steps with

each service response. This project isn't a core feature or a must-have feature but If you want to have a complete guide of all of the API services of a giving resource is an excellent option. With this project, you can quickly create model classes for links, resource representation models. It also provides a link builder API to create specific links that point to Spring MVC controller methods. The following code snippet shows an example of how HATEOAS should look like for the License service.

```
"_links": {  
  "self" : {  
    "href" : "http://localhost:8080/v1/organization/optimaGrowth/license/0235431845"  
  },  
  "createLicense" : {  
    "href" : "http://localhost:8080/v1/organization/optimaGrowth/license"  
  },  
  "updateLicense" : {  
    "href" : "http://localhost:8080/v1/organization/optimaGrowth/license"  
  },  
  "deleteLicense" : {  
    "href" : "http://localhost:8080/v1/organization/optimaGrowth/license/0235431845"  
  }  
}
```

In this section, I will show you how to implement Spring HATEOAS in the license service. The first thing we must do to send the links related to a resource in the response is adding the HATEOAS dependency into the pom.xml file. Please add the following dependency to your pom.xml

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-hateoas</artifactId>  
</dependency>
```

Once we have the dependency now, we have to update the License class in order to extend from

RepresentationModel<License> as shown in code Listing 3.10.

Listing 3.10 Extending from RepresentationModel

```
package com.optimagrowth.license.model;
import org.springframework.hateoas.RepresentationModel;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
@Getter @Setter @ToString
public class License extends RepresentationModel<License> {
private int id;
private String licenseId;
private String description;
private String organizationId;
private String productName;
private String licenseType;
}
```

The RepresentationModel<License> gives the ability to add links to the License model class.

Now that we have everything set up, let's create the links to retrieve the links of the LicenseController class. The code listing 3.11 will show you how. (For this example, I'm only going to change the getLicense() method in the LicenseController class.

Listing 3.11 Adding links to the LicenseController

```
@RequestMapping(value="/{licenseId}",method = RequestMethod.GET)
public ResponseEntity<License> getLicense(
@PathVariable("organizationId") String organizationId,
@PathVariable("licenseId") String licenseId) {
License license = licenseService.getLicense(licenseId, organizationId);
license.add(
linkTo(methodOn(LicenseController.class).getLicense(organizationId,
license.getLicenseId())).withSelfRel(),
linkTo(methodOn(LicenseController.class).createLicense(organizationId, license,
null)).withRel("createLicense"),
linkTo(methodOn(LicenseController.class).updateLicense(organizationId,
license)).withRel("updateLicense"),
```

```
linkTo(methodOn(LicenseController.class).deleteLicense(organizationId,  
license.getLicenseId())).withRel("deleteLicense"));  
return ResponseEntity.ok(license);  
}
```

The method add() is a method of the RepresentationModel. The linkTo method inspects the License controller class and obtains the root mapping, and the methodOn obtains the method mapping by doing a dummy invocation of the target method. Both methods are static methods of the org.springframework.hateoas.server.mvc.WebMvcLinkBuilder . The WebMvcLinkBuilder is a utility class for creating links on the controller classes.

Figure 3.8 will show you the links on the response body of the getLicense() service, to do you must call the GET http method.

The screenshot shows a Postman request for a GET operation. The URL is `http://localhost:8080/v1/organization/optimaGrowth/license/0235431845`. The response body is a JSON object:

```
1 {
2     "id": 595,
3     "licenseId": "0235431845",
4     "description": "Software product",
5     "organizationId": "optimaGrowth",
6     "productName": "Ostock",
7     "licenseType": "full",
8     "_links": {
9         "self": {
10             "href": "http://localhost:8080/v1/organization/optimaGrowth/license/0235431845"
11         },
12         "createLicense": {
13             "href": "http://localhost:8080/v1/organization/optimaGrowth/license"
14         },
15         "updateLicense": {
16             "href": "http://localhost:8080/v1/organization/optimaGrowth/license"
17         },
18         "deleteLicense": {
19             "href": "http://localhost:8080/v1/organization/optimaGrowth/license/0235431845"
20         }
21     }
22 }
```

Figure 3.8 HATEOAS links on the response body of the HTTP GET license service.

At this point, you have a running skeleton of a service. But from a development perspective, this service isn't complete. A good microservice design doesn't eschew segregating the service into well-defined business logic and data access layers. As you progress in later chapters, you'll continue to iterate on this service and delve further into how to structure it.

Let's switch to the final perspective: exploring how a DevOps engineer would operationalize the service and package it for deployment to the cloud.

3.4 The DevOps story: building for the rigors of runtime

For the DevOps engineer, the design of the microservice is all about managing the service after it goes into production. Writing the code is often the easy part. Keeping it running is the hard part.

While DevOps is a rich and emerging IT field, you'll start your microservice development effort with four principles and build on these principles later in the book.

These principles are

1. A microservice should be **self-contained** and **independently deployable** with multiple instances of the service being started up and torn down with a single software artifact.
2. A microservice should be **configurable**. When a service instance starts up, it should read the data it needs to configure itself from a central location or have its configuration information passed on as environment variables. No human intervention should be required to configure the service.
3. A microservice instance needs to be **transparent** to the client. The client should never know the exact location of a service. Instead, a microservice client should talk to a service discovery agent that will allow the application to

locate an instance of a microservice without having to know its physical location.

4. A microservice should **communicate** its health. This is a critical part of your cloud architecture. Microservice instances will fail, and discovery agents need to route around bad services instances. In this book we are going to use Spring Boot Actuator to display the health of each microservice.

These four principles expose the paradox that can exist with microservice development. Microservices are smaller in size and scope, but their use introduces more moving parts in an application, especially because microservices are distributed and running independently of each other in their own distributed containers. This introduces a high degree of coordination and more opportunities for failure points in the application.

From a DevOps perspective, you must address the operational needs of a microservice upfront and translate these four principles into a standard set of lifecycle events that occur every time a microservice is built and deployed to an environment. The four principles can be mapped to the following operational lifecycle steps:

- **Service assembly.** How do you package and deploy your service to guarantee repeatability and consistency so that the same service code and runtime is deployed exactly the same way?
- **Service bootstrapping.** How do you separate your application and environment- specific configuration code from the runtime code so you can start and deploy a microservice instance quickly in any environment without human intervention to configure the microservice?

- **Service registration/discovery.** When a new microservice instance is deployed, how do you make the new service instance discoverable by other application clients?
- **Service monitoring.** In a microservices environment it's extremely common for multiple instances of the same service to be running due to high availability needs. From a DevOps perspective, you need to monitor microservice instances and ensure that any faults in your microservice are routed around and that failing service instances are taken down.

Figure 3.9 shows how these four steps fit together.

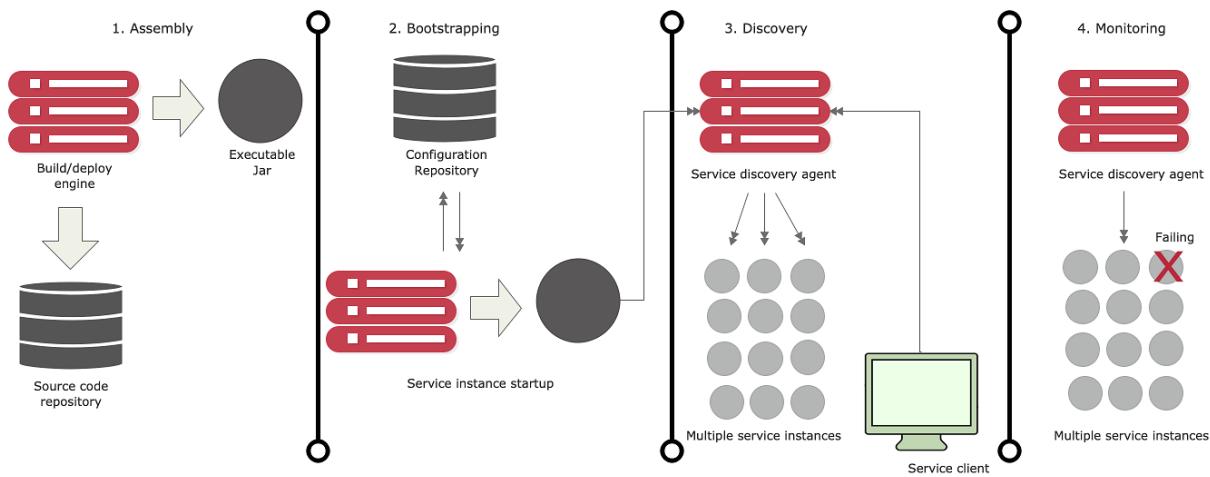


Figure 3.9 When a microservice starts up, it goes through multiple steps in its lifecycle.

3.4.1 Service assembly: packaging and deploying your microservices

From DevOps perspective, one of the key concepts behind a microservice architecture is that multiple instances of a microservice can be deployed quickly in response to a change application environment (for example, a sudden influx of user requests, problems within the infrastructure, and so on).

To accomplish this, a microservice needs to be packaged and installable as a single artifact with all of its dependencies defined within it. These dependencies will also include the runtime engine (for example, an HTTP server or application container) that will host the microservice.

This process of consistently building, packaging and deploying is the service assembly (step 1 in figure 3.9). Figure 3.10 shows additional details about the service assembly step.

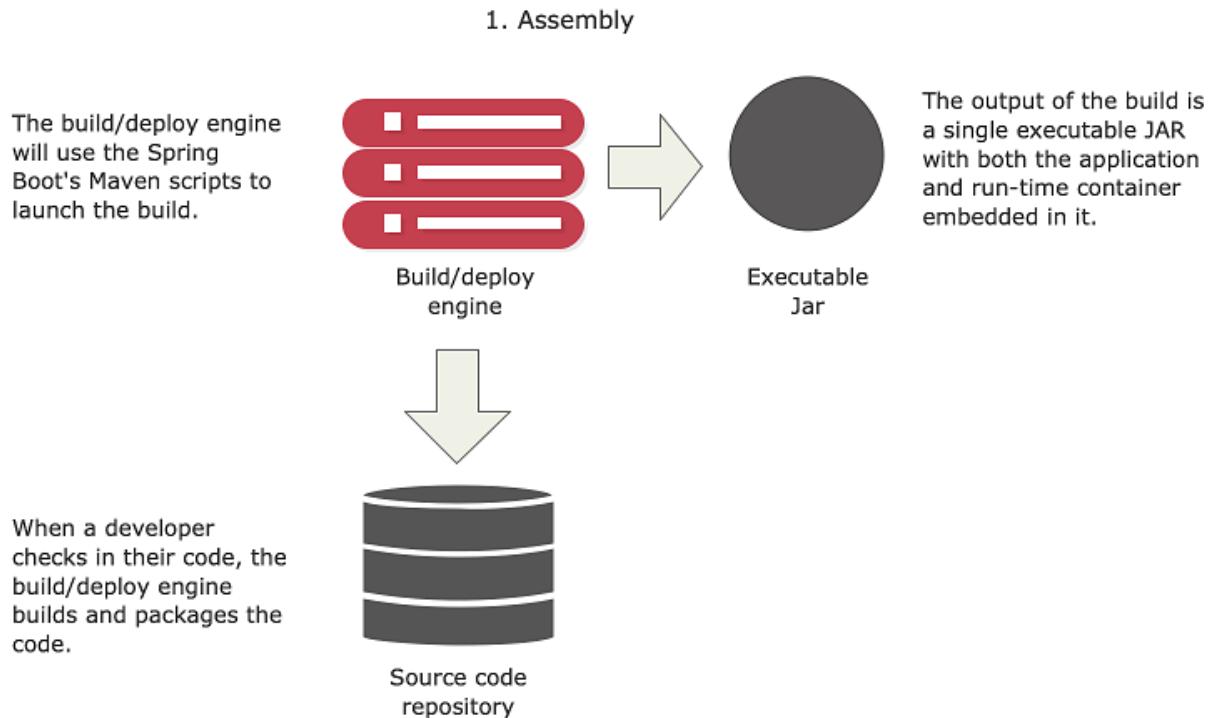


Figure 3.10 In the Service Assembly step, source code is compiled and packaged with its runtime engine.

Fortunately, almost all java microservice frameworks will include a runtime engine that can be packaged and deployed with the code. For instance, in the Spring Boot example in figure 3.10, you use Maven and Spring Boot to build an executable Java jar file that has an embedded Tomcat engine built right into the JAR. In the following command-line example, you're building the licensing service as an executable JAR and then starting the JAR file from the command-line:

```
mvn clean package && java -jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

For certain operation teams, the concept of embedding a runtime environment right in the JAR file is major shift in the way they think about deploying applications. In a traditional Java-web Application, the application is deployed to an application server. This model implies that the application server is an entity in and of itself and would often be managed by a team of system administrators who managed the configuration of the servers independently of the applications being deployed to them.

This separation of the application server configuration from the application introduces failure points in the deployment process, because in many organizations the configuration of the application servers isn't kept under source control and is managed through a combination of the user interface and home-grown management scripts. It's too easy for configuration drift to creep into the application server environment and suddenly cause what, on the surface, appear to be random outages.

3.4.2 Service bootstrapping: managing configuration of your microservices

Service bootstrapping (step 2 of figure 3.9) occurs when the microservice is first starting up and needs to load its application configuration information. Figure 3.11 provides more context for the bootstrapping processing.

As any application developer knows, there will be times when you need to make the runtime behavior of the

application configurable. Usually this involves reading your application configuration data from a property file deployed with the application or reading the data out of a data store such as a relational database.

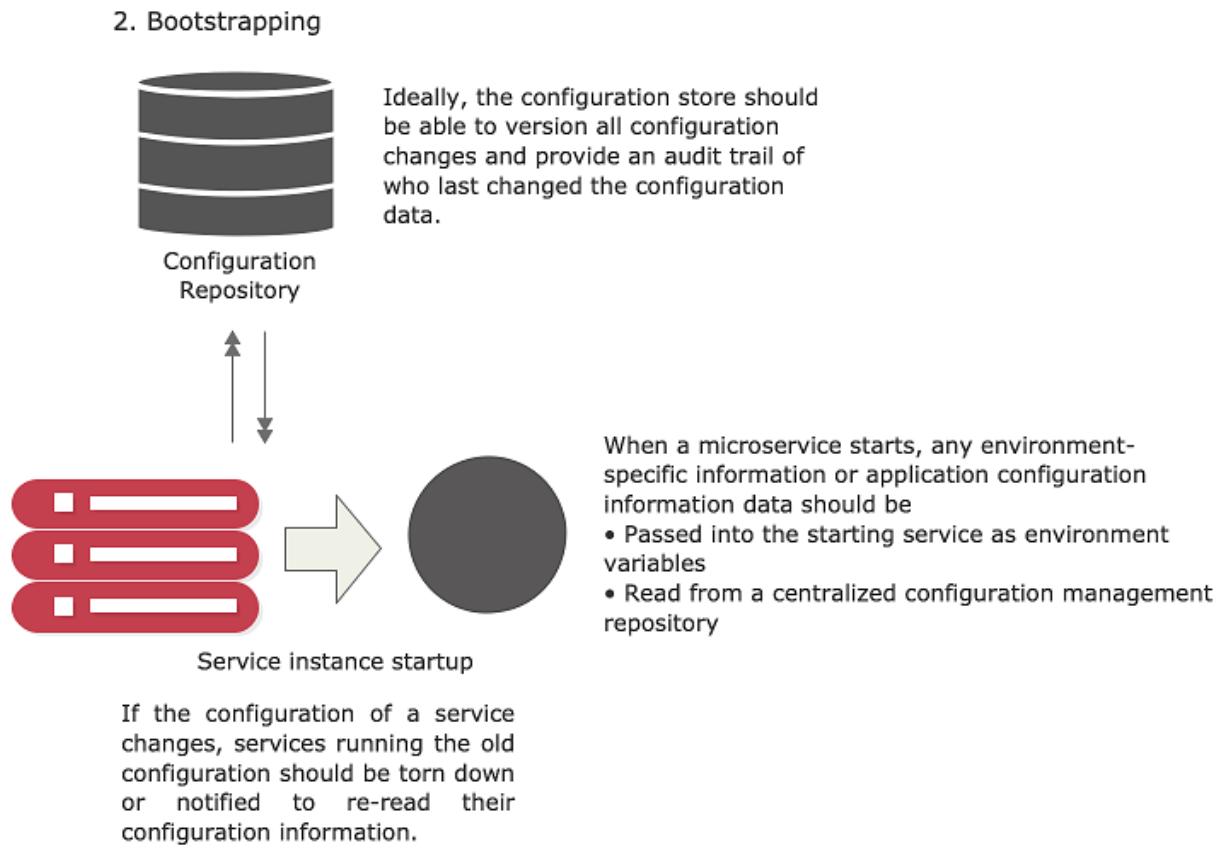


Figure 3.11 As a service starts (bootstraps), it reads its configuration from a central repository.

Microservices often run into the same type of configuration requirements. The difference is that in microservice application running in the cloud, you might have hundreds or even thousands of microservice instances running. Further complicating this is that the services might be spread across

the globe. With a high number of geographically dispersed services, it becomes unfeasible to redeploy your services to pick up new configuration data.

Storing the data in a data store external to the service solves this problem, but microservices in the cloud offer a set of unique challenges:

1. Configuration data tends to be simple in structure and is usually read frequently and written infrequently.
Relational databases are overkill in this situation because they're designed to manage much more complicated data models than a simple set of key-value pairs.
2. Because the data is accessed on a regular basis but changes infrequently, the data must be readable with a low level of latency.
3. The data store has to be highly available and close to the services reading the data. A configuration data store can't go down completely, because it would become a single point of failure for your application.

In chapter 5, I show how to manage your microservice application configuration data using things like a simple key-value data store.

3.4.3 Service registration and discovery: how clients communicate with your microservices

From a microservice consumer perspective, a microservice should be location transparent, because in a cloud-based environment, servers are ephemeral. Ephemeral means the

servers that a service is hosted on usually have shorter lives than a service running in a corporate data center. Cloud-based services can be started and torn down quickly with an entirely new IP address assigned to the server on which the services are running.

By insisting that services are treated as short-lived disposable objects, microservice architectures can achieve a high degree of scalability and availability by having multiple instances of a service running. Service demand and resiliency can be managed as quickly as the situation warrants. Each service has a unique and non-permanent IP address assigned to it. The downside to ephemeral services is that with services constantly coming up and down, managing a large pool of ephemeral services manually or by hand is an invitation to an outage.

A microservice instance needs to register itself with the third-party agent. This registration process is called service discovery (see step 3, service discovery, in figure 3.9; see figure 3.12 for details on this process). When a microservice instance registers with a service discovery agent, it will tell the discovery agent two things: the physical IP address or domain address of the service instance, and a logical name that an application can use to look up a service. Certain service discovery agents will also require a URL back to the registering service that can be used by the service discovery agent to perform health checks.

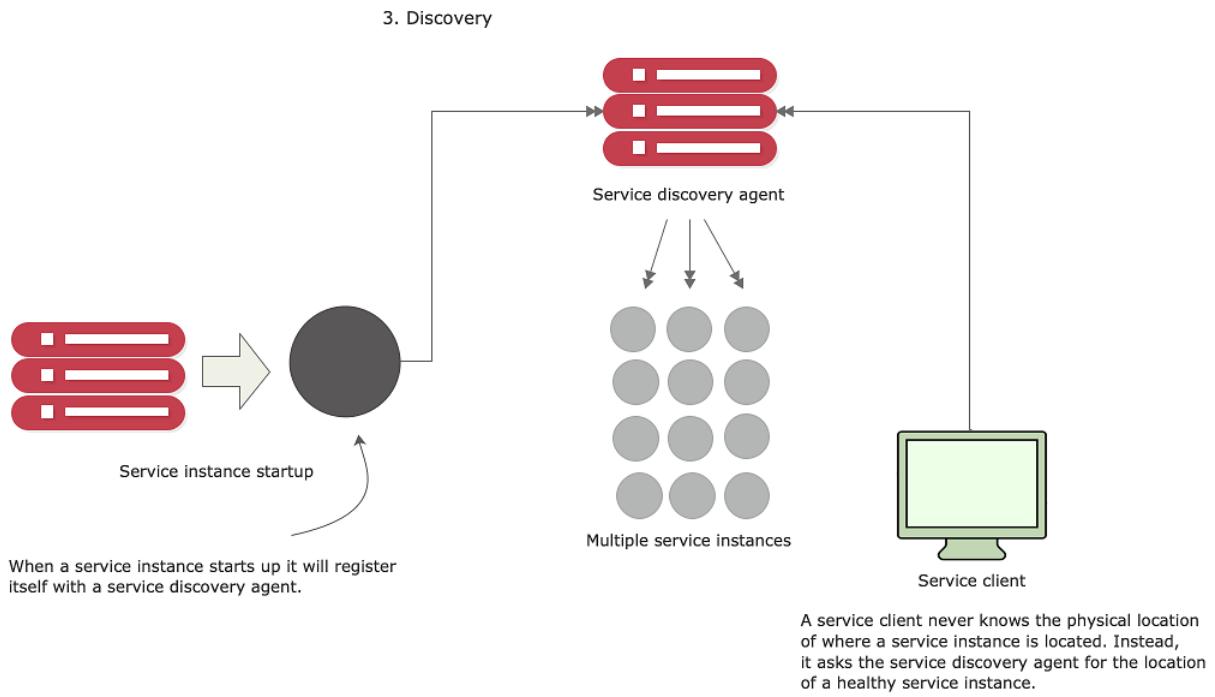


Figure 3.12 A service discovery agent abstracts away the physical location of a service.

The service client then communicates with the discovery agent to look up the service's location.

3.4.4 Communicating a microservice's health

A service discovery agent doesn't act only as a traffic cop that guides the client to the location of the service. In a cloud-based microservice application, you'll often have multiple instances of a service running. Sooner or later, one of those service instances will fail. The service discovery agent monitors the health of each service instance registered with it and removes any service instances from its

routing tables to ensure that clients aren't sent a service instance that has failed.

After a microservice has come up, the service discovery agent will continue to monitor and ping the health check interface to ensure that that service is available. This is step 4 in figure 3.9. Figure 3.13 provides context for this step.

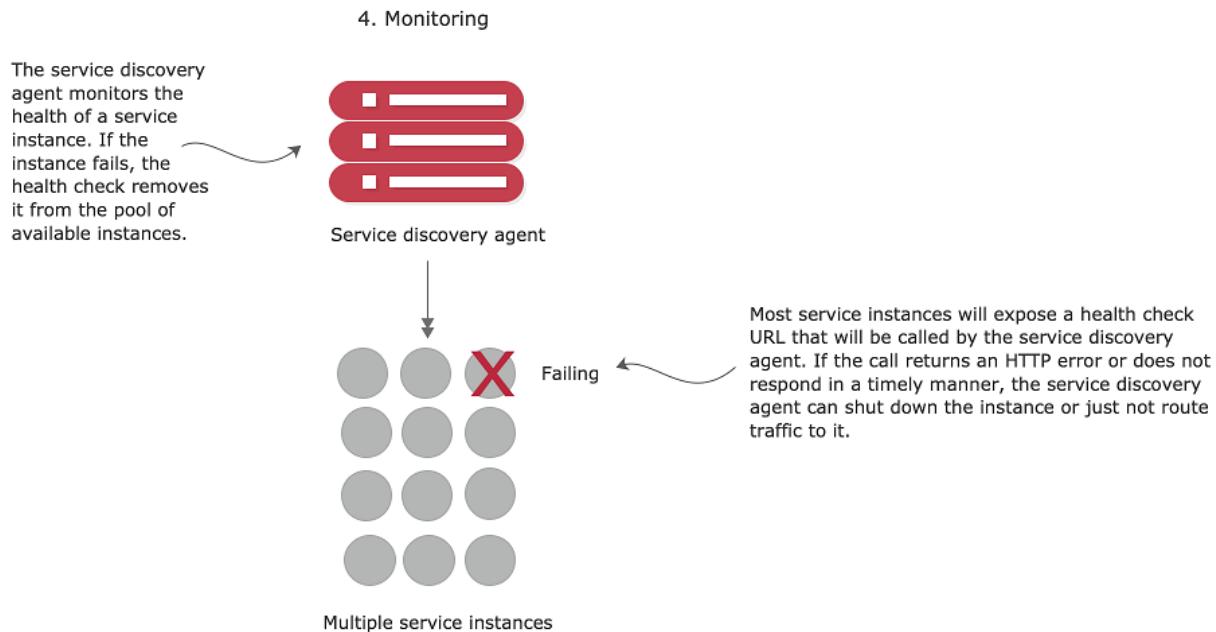


Figure 3.13 The service discovery agent uses the exposes health URL to check the microservice health.

By building a consistent health check interface, you can use cloud-based monitoring tools to detect problems and respond to them appropriately.

If the service discovery agent discovers a problem with a service instance, it can take corrective action such as

shutting down the ailing instance or bringing additional service instances up.

In a microservices environment that uses REST, the simplest way to build a health check interface is to expose an HTTP endpoint that can return a JSON payload and HTTP status code. In a non-Spring-Boot-based microservice, it's often the developer's responsibility to write an endpoint that will return the health of the service.

In Spring Boot, exposing an endpoint is trivial and involves nothing more than modifying your Maven build file to include the Spring Actuator module. Spring Actuator provides out-of-the-box operational endpoints that will help you understand and manage the health of your service. To use Spring Actuator, you need to make sure you include the following dependencies in your Maven build file:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

If you hit the `http://localhost:8080/actuator/health` endpoint on the licensing service, you should see health data returned. Figure 3.14 provides an example of the data returned.

GET http://localhost:8080/health

Params Authorization Headers (7) Body Pre-request Script Tests

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize BETA JSON ↻

```
1 {
2   "status": "UP",
3   "components": {
4     "diskSpace": {
5       "status": "UP",
6       "details": {
7         "total": 250685575168,
8         "free": 60396019712,
9         "threshold": 10485760
10      }
11    },
12    "ping": {
13      "status": "UP"
14    }
15  }
16 }
```

The "out of the box" Spring Boot health check will return whether the service is up and some basic information like how much disk space is left on the server.

Figure 3.14 A health check on each service instance allows monitoring tools to determine if the service instance is running.

As you can see in figure 3.14, the health check can be more than an indicator of what's up and down. It also can give information about the state of the server on which the microservice instance is running. Spring actuator allows you

to change the default configuration via the application properties file. For example:

```
management.endpoints.web.base-path=/  
management.endpoints.enabled-by-default=false  
management.endpoint.health.enabled=true  
management.endpoint.health.show-details=always  
management.health.db.enabled=false  
management.health.diskspace.enabled=true
```

The first line will allow you to set the base-path for all the actuator services, for example the health endpoint will now be exposed in the `http://localhost:8080/health` URL the other lines allow you to disable the default services and enable the services you want to use.

NOTE If you are interested in knowing all of the services exposed by Spring Actuator, I highly recommend you read the following Spring Document (<https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html>)

3.5 Pulling the perspectives together

Microservices in the cloud seem deceptively simple. But to be successful with them, you need to have an integrated view that pulls the perspective of the architect, the developer, and DevOps engineer together into a cohesive vision. The key takeaways for each of these perspectives are

1. **Architect.** Focus on the natural contours of your business problem. Describe your business problem domain and listen to the story you're telling. Target microservice candidates will emerge. Remember, too,

that it's better to start with a "coarse-grained" microservice and refactor back to smaller services than to start with a large group of small services. Microservice architectures, like most good architectures, are emergent and not preplanned to-the-minute.

2. **Software engineer.** The fact that the service is small doesn't mean good design principles get thrown out the window. Focus on building a layered service where each layer in the service has discrete responsibilities. Avoid the temptation to build frameworks in your code and try to make each microservice completely independent. Premature framework design and adoption can have massive maintenance costs later in the lifecycle of the application.

3. **DevOps engineer.** Services don't exist in a vacuum. Establish the lifecycle of your services early. The DevOps perspective needs to focus not only on how to automate the building and deployment of a service, but also on how to monitor the health of the service and react when something goes wrong. Operationalizing a service often takes more work and forethought than writing business logic. In Appendix C I explain how to achieve this with Prometheus and Grafana.

3.6 Summary

- To be successful with microservices, you need to integrate the architect's, software developer's, and DevOps' perspectives.
- Microservices, while a powerful architectural paradigm, have their benefits and tradeoffs. Not all applications should be microservice applications.
- From an architect's perspective, microservices are small, self-contained, and distributed. Microservices should have narrow boundaries and manage a small set of data.

- From a developer's perspective, microservices are typically built using a REST-style of design, with JSON as the payload for sending and receiving data from the service.
- The main goal of internationalization is to develop applications that offer content in multiple formats and languages.
- HATEOAS stands for Hypermedia as the Engine of Application State). Spring HATEOAS is a small project of Spring that allows us to create APIs that follow the HATEOAS principle of displaying the related links for a given resource.
- From a DevOp's perspective, how a microservice is packaged, deployed, and monitored are of critical importance.
- Out of the box, Spring Boot allows you to deliver a service as a single executable JAR file. An embedded Tomcat server in the producer JAR file hosts the service.
- Spring Actuator, which is included with the Spring Boot framework, exposes information about the operational health of the service along with information about the runtime of the service.

4 Welcome to Docker

This chapter covers

- Understanding the importance of containers and how they fit into the microservices architecture
- Understanding the differences between a virtual machine and a container
- Using Docker and its main components
- Integrating Docker with microservices

To continue successfully building our microservices, we need to address the portability issue: how are we going to execute our microservices in different technology locations?

Portability is the ability to use or move software to different environments. In recent years, the concept of containers has gained more and more strength, going from a "nice to have" to a "must-have" in software architecture.

The use of containers is an agile and useful way to migrate and execute any software development from one platform to another, for example, from the developer's machine to a physical or virtual enterprise server. We can replace the traditional models of web servers with smaller and much more adaptable virtualized software containers that offer advantages such as speed, portability, and scalability to our microservices.

This chapter provides a brief introduction to containers using Docker, a technology I selected because it can be used in all the major cloud providers. I will explain what Docker is, how to use its core components, and how to integrate Docker with our microservices. By the end of the chapter, you will be able to run Docker, create your own images with Maven, and execute your microservices within a container. Also, you'll notice you no longer have to worry about installing all the technology prerequisites your microservices need to run; the only requirement is that you have an environment with Docker installed.

NOTE This chapter contains only a brief introduction to Docker, and I will only explain what we are going to use throughout the book. If you are interested in knowing more about Docker, I highly recommend you check out the excellent book *Docker in Action*, Second Edition (Manning Publications, Oct 2019). Authors Jeff Nickoloff, Stephen Kuenzli, and Bret Fisher give an exhaustive overview of what Docker is and how it works.

4.1 Containers or virtual machines?

In many companies, virtual machines are still the de facto standard for software deployment. Let's look at the main differences between virtual machines and containers.

A virtual machine (VM) is a software that allows us to emulate the operation of a computer within another computer. They are based on a hypervisor that emulates the complete physical machine allocating the desired amount of system memory, processor cores, disk storage, and other

technological resources such as networks, PCI add-ons, and more.

On the other hand, containers are another operating system virtualization method that allows you to run an application with its dependent elements in an isolated and independent environment.

Both technologies have similarities, like the existence of a hypervisor or container engine that allows the execution of both technologies, but the way they implement it makes them very different. In figure 4.1, you can see the main differences between VMs and containers.

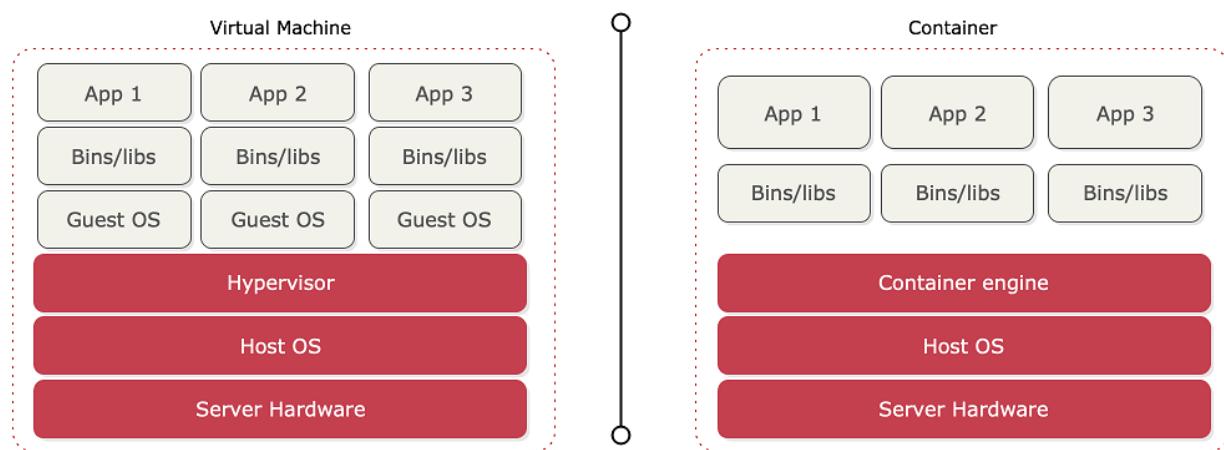


Figure 4.1 Main differences between virtual machines and containers. Containers don't need the Guest Os nor the hypervisor to assign resources; instead, they use the container engine.

If we analyze figure 4.1, at first glance, we can see that there isn't that much difference; after all, only the layer of the guest operating system has disappeared in the container,

and the hypervisor is replaced by the container's engine. However, the differences between virtual machines and containers are enormous.

In a virtual machine, we must set in advance how many physical resources we need, for example, how many virtual processors are we going to use, how many GBs of RAM and disk space are we going to use. Defining those values can be a tricky task, and we must be careful, and take into consideration the following: Processors can be shared between different virtual machines, the disk space can be set only to use what it needs so it will grow in the function of their needs but with the memory is a different story. The reservation with the memory is total and can't be shared between VMs.

In containers, we can also set the memory and the CPU that we need by using Kubernetes, but it isn't required. In case you don't specify these values, the container engine is responsible for assigning the necessary resources for the container to function correctly. The fact that containers don't need a complete operating system but that they can reuse the underlying one reduces the load the physical machine must support as well as the storage space used, and the time needed to launch the applications. Therefore, the containers are much lighter than virtual machines.

In the end, both technologies have their benefits and drawbacks, and the ultimate decision will depend on your specific needs. For example, if you want to handle a variety of operating systems, manage multiple applications on a single server and execute an application that requires functionalities of an operating system virtual machines are a

better solution for you. In this book I've chosen to use containers due to the cloud architecture we are building, so instead of virtualizing the hardware as with the virtual machines approach, we will use containers to only virtualize the operating system level creating a much lighter and faster alternative that will be able to run on-premises and on every major cloud providers.

Nowadays, the performance and portability are critical concepts for decision making in a company, so it's really important to highlight and known the benefits of the technologies we are going to use. In this case with the use of containers with our microservices we will be having:

- **Increased portability.** Containers can run everywhere, which facilitates the development and implementation.
- **Consistent environments.** Ability to create predictable environments that are entirely isolated from other applications.
- Containers can be started and stopped faster than VMs, reason that makes them cloud-native feasible. Each of these containers are scalable and can be actively scheduled and managed to optimize the resource utilization to increase the performance and the maintainability of the application running within.
- Maximum number of applications on the minimum number of servers.

Now that we understand the difference between virtual machines and containers, let's take a look at Docker.

4.2 What is Docker?

Docker is a popular open-source container engine based on Linux containers, created by Solomon Hykes, founder and CEO of dotCloud on March 15, 2013. Docker started as a nice to have technology that was responsible for launching and managing containers with our applications. This technology allowed us to share the resources of a physical machine with different containers instead of exposing different hardware resources like virtual machines. The support that big companies such as IBM, Microsoft, and Google gave Docker with that technological breakthrough allowed them to convert a new technology into a fundamental tool for software developers.

Nowadays, Docker continues to grow and currently represents one of the most used tools to deploy software on any server through containers.

NOTE Remember, a container represents a logical packaging mechanism where applications have everything they need to run.

To understand better how docker works is essential to highlight that the docker engine is the core part of the entire docker system. So, what is the docker engine? It is an application that follows the client-server pattern architecture. This application is installed in the host machine and contains the following three critical components: Server, Rest API, and Command Line Interface (CLI). These components and other components are described in Figure 4.2.

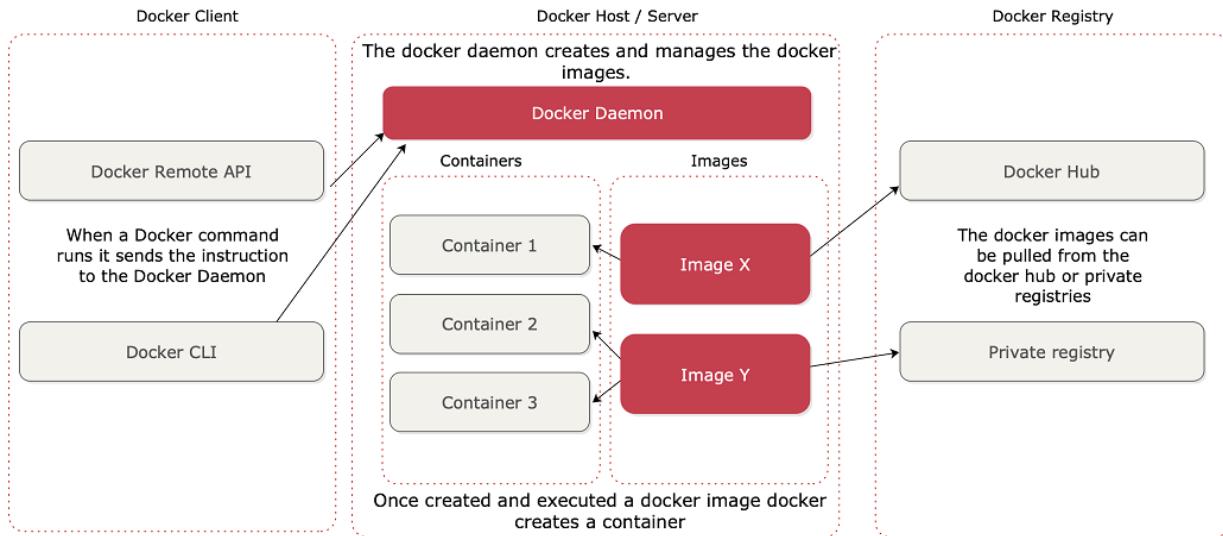


Figure 4.2 Docker architecture composed by the Docker Client, Host and Registry.

- **Docker daemon.** The server is the docker daemon called dockerd that allows us to create and manage the docker images. The Rest API is used to send instructions to the daemon, and the CLI is a client to enter docker commands.
- **Docker client.** Docker users can interact with Docker using a client. When a Docker command runs, the client is in charge of sending the instruction to the daemon.
- **Docker registries.** The registries are the location where Docker images are stored. These registries can be either public or private. It's important to highlight that the Docker Hub is the default place for the public registries, but you can also create your own private registry.
- **Docker images.** The docker images are read-only templates with several instruction commands to create a docker container. The docker images can be pulled from the docker hub, and you can use it as it is, or you can modify it by adding the additional instructions you need. Also, you can create new images by using a Dockerfile.

Later on, in this chapter, I will explain how to use dockerfiles.

- **Docker containers.** Once we've created and executed by using the run command a Docker image creates a container. The application and their environment will run inside this container. In order to start, stop, and delete a docker container, you can use the Docker API or the CLI.
- **Docker volumes.** Docker volumes are the preferred mechanism to store data generated by Docker and used by the Docker containers. They can be managed using the Docker API and the CLI.
- **Docker networks.** The docker networks allow us to attach the containers to as many networks as we like. We can see the networks as a passage to communicate with the isolated containers. Docker contains the following five network drivers: Bridge, Host, Overlay, None, and macvlan.

Figure 4.3 shows a diagram of how Docker works. It's important to highlight that the Docker daemon is responsible for all the container's actions. As shown in Figure 4.3, we can see that the daemon receives the commands from the Docker client; these commands can be sent by using the CLI or the REST APIs. In the diagram, we can see how the images found in the registries create the containers.

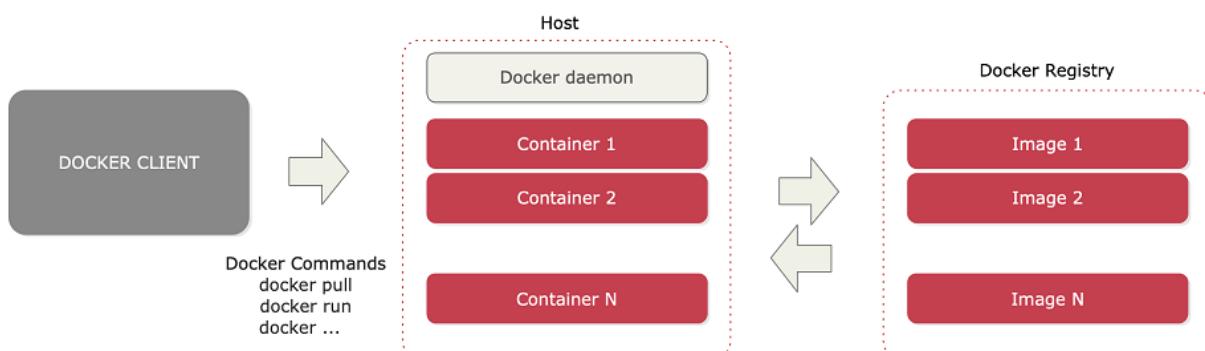


Figure 4.3 The docker client sends the docker commands to the docker daemon and the docker daemon creates the containers based on the docker images.

NOTE In this book, I won't teach you how to install Docker, so if you don't have Docker installed on your computer, I highly recommend you look at the documentation of following link <https://docs.docker.com/install/> that contains all the steps to install and configure Docker in Windows, macOS, or Linux.

In the next sections, I will explain in detail how those components work and how to integrate them with the license microservice. In case you didn't follow the example in chapter 1 and 3, you can apply what I'm about to explain to any Java Maven project you have.

4.3 Dockerfiles

A Dockerfile is a simple text file that contains a list of instructions and commands that the docker client calls to create and prepare an image. Docker created this file to automate the image creation process. The commands used in the Dockerfile are similar to the Linux commands what makes it easier to understand. The following code snippet shows a brief example of how a Dockerfile should look like. In section 4.5.1, I will show you how to create our custom Dockerfiles for our microservices.

```
FROM openjdk:11-slim
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Figure 4.4 shows how the docker image creation workflow should look like.



Figure 4.4 Once the dockerfile is created, you should run the docker build command which will build a docker image and then once the docker image is ready use the run command to create the containers.

In the next section I will be explaining some of the most common commands we are going to use in our Dockerfile.

4.3.1 Dockerfile commands

The most common commands in a Dockerfile are the following:

- **FROM.** This command defines a base image to use and to start the build process. In other words, the FROM command specifies the docker image that you're going to use in your Docker run-time.
- **LABEL.** This command allows us to add metadata to an image. This is a key-value pair.
- **ARG.** This command or instruction allows us to define variables that the user can pass at the build-time to builder using the docker build command.
- **COPY.** This command allows us to copy new files, directories, or remote file URLs from a source and adds

them to the filesystem of the image we are creating at the specified destination path.

- **VOLUME.** This command allows us to create a mount point in our container. So, when we create a new container using the same image, we will create a new volume that will be isolated from the previous one.
- **RUN.** This command takes the command and its arguments to run it from the image. Often used for installing software packages.
- **CMD.** The main objective of this command is to provide arguments to the ENTRYPOINT. This command is similar to the run command with the difference that this one gets executed only after the container is instantiated.
- **ADD.** This command allows you to copy the files from a source to a destination within the container.
- **ENTRYPOINT.** This command allows you to configure a container that is going to run as an executable.
- **ENV.** It allows you to set environment variables.

4.4 Docker compose

Docker Compose is a tool that simplifies the use of Docker by allowing us to create scripts that facilitate the design and the construction of services. With Docker compose, you can run multiple containers as a single service, and you can create different containers at the same time.

To use Docker Compose, you should follow the next steps:

1. Install the Docker Compose Tool, if you don't have it.
(<https://docs.docker.com/compose/install/>)
2. Create an yml file to configure the application services.
The file should be named docker-compose.yml

3. Check the validity of the file using the following command: docker-compose config
4. Start the services by using the following command:
docker-compose up

A docker-compose.yml should look like the code listing 4.1, later on this chapter I will explain how to create our docker-compose.yml file.

Listing 4.1 docker-compose.yml example

```
version: <docker-compose-version>
services:
database:
image: <database-docker-image-name>
ports:
- "<databasePort>:<databasePort>"
environment:
POSTGRES_USER: <databaseUser>
POSTGRES_PASSWORD: <databasePassword>
POSTGRES_DB:<databaseName>
<service-name>:
image: <service-docker-image-name>
ports:
- "<applicationPort>:<applicationPort>"
environment:
PROFILE: <profile-name>
DATABASESERVER_PORT: "<databasePort>"
container_name: <container_name>
networks:
backend:
aliases:
- "alias"
networks:
backend:
driver: bridge
```

The instructions we are going to use from the docker-compose.yml are the following:

- **Version.** Specifies the version of the Docker compose tool.
- **Services.** Specifies to the Docker compose what the services to deploy are. The services name will become

the DNS entry for the Docker instance when it's started and is how other services can access it.

- **Image.** Specifies the tool to run a container using the specified image.
- **Ports.** This entry defines the port numbers on the started Docker container that will be exposed to the outside world. It maps the internal and the external port.
- **Environment.** The environment tag is used to pass along environment variables to the starting Docker image.
- **Networks.** Specifies a custom network allowing us to create complex topologies. The default type is bridge, so if we don't specify another type of network (host, overlay, macvlan, none), we will be creating a bridge one. The bridge network will allow us to maintain the containers connect to the same network. It is essential to highlight that the bridge network will only apply to the containers running on the same Docker daemon host.
- **Aliases.** Specifies an alternative hostname for the service on the network.

4.4.1 Docker-compose commands

The docker-compose commands we will be using throughout the book are the following:

- docker-compose up -d. Builds the images for your application and start the services you defined. This command will download all the necessary images and then deploy them and start the containers. The -d parameter indicates to run in the background.
- docker-compose logs. With this command, you can see all the information about your last deployment.
- docker-compose logs <service_id>. For example, docker-compose logs licenseService allows you to see the logs for the license service deployment.

- docker-compose ps. Will output the list of all the containers you have deployed in your system.
- docker-compose stop. Stop your services once you have finished with them; this will also stop the containers.
- docker-compose down. Shutdowns everything and removes all containers.

4.5 Integrating docker with our microservices

Now that we understand the main components of Docker, let's integrate Docker with our licensing microservice to create a more portable, scalable and manageable microservice. To achieve this integration, let's start by adding the dockerfile-maven-plugin to our licensing-service created in the previous chapters. In case you didn't follow the code listings, you can download the code created in chapter 3 from the following link:

<https://github.com/ihuaylupo/manning-smia/tree/master/chapter3>.

4.5.1 Building the Docker Image

Let's start; the first step to build a Docker image is to add that specific dockerfile-maven-plugin to the pom.xml of our licensing service. This plugin will allow us to manage the docker images and containers from our maven pom.xml file. The following code listing 4.2 shows how your pom file should look like.

Listing 4.2 Add dockerfile-maven-plugin to pom.xml

```

<build> #A
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
<!-- This plugin is used to create a docker image and publish it to docker hub-->
<plugin> #B
<groupId>com.spotify</groupId>
<artifactId>dockerfile-maven-plugin</artifactId>
<version>1.4.13</version>
<configuration>
<repository>${docker.image.prefix}/${project.artifactId}</repository> #C
<tag>${project.version}</tag> #D
<buildArgs>
<JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE> #E
</buildArgs>
</configuration>
<executions>
<execution>
<id>default</id>
<phase>install</phase>
<goals>
<goal>build</goal>
<goal>push</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>

```

- #A 1. Build section of the pom.xml
- #B 2. Starts the dockerfile-maven-plugin
- #C 3. Set the remote repository name. In this scenario, we are going to use a predefined variable called docker.image.prefix and the project.artifactId.
- #D 4. Sets the repository tag with the project version.
- #E 5. Sets the JAR file location by using <buildArgs>, this value will be used in the dockerfile.

Now that we have the plugin in our pom.xml, we can continue the process by creating the variable docker.image.prefix that I mentioned in the previous code listing. This variable will indicate what prefix we are going to use for our image. The following code listing 4.3 shows how to add the variable into the pom.xml.

Listing 4.3 Add docker.image.prefix variable

```
<properties>
<java.version>11</java.version>
<docker.image.prefix>ostock</docker.image.prefix> #A
</properties>
```

#A 1. Sets the value for the docker.image.prefix variable. There are several ways to define its value, this is one option the other one is directly sending the value using the option -d in the maven JVM arguments.

NOTE Remember, if you don't create the docker.image.prefix variable in the properties section of the pom.xml when you execute the command to package and create the Docker image; it will throw the following error.
Failed to execute goal com.spotify:dockerfile-maven-plugin:1.4.0:build (default-cli) on project licensing-service: Execution default-cli of goal com.spotify:dockerfile-maven-plugin:1.4.0:build failed: The template variable 'docker.image.prefix' has no value

Now that we've imported the plugin into the pom.xml, let's continue by creating the Dockerfile into our project. In the next sections, I will show you how to create a basic Dockerfile and a Multi-stage build Dockerfile. It's important to highlight that you can use both Dockerfiles because both files will allow you to execute your microservice. The main difference between them is that with the basic Dockerfile, you are going to copy the entire JAR file of your Spring Boot Microservice, and with the multi-stage build, you are going to copy only what's essential to the application to run. In this book, I chose to use the multi-stage build to optimize the docker image we are going to create, but feel free to use the option that best suits your needs.

BASIC DOCKERFILE

In this Dockerfile we will copy the Spring Boot jar file into the image and then we will execute the application jar. The

following Listing 4.4 will show you how to achieve that in a few simple steps.

Listing 4.4 Basic Dockerfile

```
#Start with a base image containing Java runtime #A
FROM openjdk:11-slim
# Add Maintainer Info
LABEL maintainer="Hillary Huaylupo <illaryhs@gmail.com>" # The application's jar file
ARG JAR_FILE #B
# Add the application's jar to the container
COPY ${JAR_FILE} app.jar #C
#execute the application
ENTRYPOINT ["java","-jar","/app.jar"] #D
```

#A The FROM command specifies the docker image that we are going to use in your Docker run-time. In this case, we are going to use openjdk:11-slim
#B The ARG command allows us to define the JAR_FILE variable. The dockerfile-maven-plugin sets this variable.
#C The COPY command allows us to copy the jar file to the filesystem of the image by the name of app.jar
#D The ENTRYPOINT command allows us to target the licensing service application in the image when the container is created.

MULTI-STAGE BUILD

In this Dockerfile I will be using a Multi-Stage build. Why Multi-Stage? This will allow us to discard anything that isn't essential to the execution of the application. For example, with Spring Boot, instead of copying the entire target directory to the docker image, we will only copy what's necessary to run the Spring Boot application. This practice will optimize the Docker image we are creating. Listing 4.5 shows how your Dockerfile should look like

Listing 4.5 Add Dockerfile

```

#stage 1
#Start with a base image containing Java runtime
FROM openjdk:11-slim as build
# Add Maintainer Info
LABEL maintainer="Illary Huaylupo <illaryhs@gmail.com>"
# The application's jar file
ARG JAR_FILE
# Add the application's jar to the container
COPY ${JAR_FILE} app.jar
#unpackage jar file
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf /app.jar) #A
#stage 2 #B
#Same Java runtime
FROM openjdk:11-slim
#Add volume pointing to /tmp
VOLUME /tmp
#Copy unpackaged application to new container #C
ARG DEPENDENCY=/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
#execute the application #D
ENTRYPOINT ["java","-
cp","app:app/lib/*","com.optimagrowth.license.LicenseServiceApplication"]

```

#A Unpacks the app.jar copied on the previous step into the filesystem of the build image
#B New image that contains the different layers of a Spring Boot Application instead of having the complete jar file.
#C Copies the different layers from the first image named build.
#D The ENTRYPOINT command allows us to target the licensing service application in the image when the container is created.

We won't go through the entire file in detail but note a few key areas as we begin. In the first step, the Dockerfile creates an image called build using the openJDK image that is optimized for the Java applications. This image will be in charge of creating and unpacking the JAR application file.

NOTE The image we are going to use already has the Java 11 JDK installed on it.

Next, the Dockerfile will obtain the value for the JAR_FILE variable we set in the <configuration> < buildArgs> section of the pom.xml. In the third and fourth steps, the JAR file is

copied into the image filesystem as app.jar and unpacked the JAR file to expose the different layers that a Spring Boot application has.

Once the different layers are exposed, the Dockerfile creates the other image that will only contain the layers instead of the complete application jar.

Next, the Dockerfile will copy the different layers into the new image; it's essential to highlight that if we don't change the dependencies of the project, the BOOT-INF/lib doesn't change. The BOOT-INF/lib folder contains all of the internal and external dependencies needed to run the application.

On the final step, the ENTRYPOINT command will allow us to target the licensing service application in the new image when the container is created.

NOTE To understand more the multi-stage build process, you can take a look inside the Spring Boot application fat jar by executing the following command on the target folder of your microservice. jar tf jar-file. For example, for the licensing-service the command should look like the following: jar tf licensing-service-0.0.1-SNAPSHOT.jar. In case you don't have the Jar file in the target folder execute the following maven command on the root of the pom.xml file of your project. mvn clean package

Now that we have our maven environment set up, let's build our Docker image. To achieve that you will need to execute the following command:

```
mvn package dockerfile:build
```

NOTE Please verify you have at least the 18.06.0 or greater version of the docker engine on your local machine in order to guarantee that all the

Docker code samples are going to run successfully. To find out your docker version just execute the following command: docker version

Once the docker image is built, you should see something like figure 4.5.

```

--- dockerfile-maven-plugin:1.4.13:build (default-cli) @ licensing-service ---
dockerfile: null
contextDirectory: /Users/illary.huaylupo/Documents/Personal/Manning/code/manning-smia/chapter4/licensing-service
Building Docker context /Users/illary.huaylupo/Documents/Personal/Manning/code/manning-smia/chapter4/licensing-
Path(dockerfile): null
Path(contextDirectory): /Users/illary.huaylupo/Documents/Personal/Manning/code/manning-smia/chapter4/licensing-
Image will be built as ostock/licensing-service:0.0.1-SNAPSHOT

Step 1/12 : FROM openjdk:11-slim as build
Pulling from library/openjdk
Digest: sha256:225e03d0955b1cd6da39003db94f0e655b112b76bd65a29e06e8dd98e9030bf5
Status: Image is up to date for openjdk:11-slim
--> 6085fd745c24
Step 2/12 : LABEL maintainer="Ilary Huaylupo <illaryhs@gmail.com>"
--> Running in 371380f9e53e
Removing intermediate container 371380f9e53e
--> 57a44e0e985b
Step 3/12 : ARG JAR_FILE
--> Running in 4655b1862836
Removing intermediate container 4655b1862836
--> 71e6835e40fc
Step 4/12 : COPY ${JAR_FILE} app.jar
--> dee44fe0de6c
Step 5/12 : RUN mkdir -p target/dependency && (cd target/dependency; jar -xf /app.jar)
--> Running in 9c23dabae835
Removing intermediate container 9c23dabae835
--> f0aa2fff9fec
Step 6/12 : FROM openjdk:11-slim
Pulling from library/openjdk
Digest: sha256:225e03d0955b1cd6da39003db94f0e655b112b76bd65a29e06e8dd98e9030bf5
Status: Image is up to date for openjdk:11-slim
--> 6085fd745c24
Step 7/12 : VOLUME /tmp
--> Using cache
--> bf23ae387bbd
Step 8/12 : ARG DEPENDENCY=/target/dependency
--> Running in 9b6a06b1b495
Removing intermediate container 9b6a06b1b495
--> 5dab66558903
Step 9/12 : COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
--> bb8ecadb4040
Step 10/12 : COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
--> 06ddb7fc4d1e
Step 11/12 : COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
--> 5ae10ed09222
Step 12/12 : ENTRYPOINT ["java","-cp","app:app/lib/*","com.optimagrowth.license.LicenseServiceApplication"]
--> Running in 464f8b4b4a45
Removing intermediate container 464f8b4b4a45
--> 52cef232a505
Successfully built 52cef232a505
Successfully tagged ostock/licensing-service:0.0.1-SNAPSHOT

Detected build of image with id 52cef232a505
Building jar: /Users/illary.huaylupo/Documents/Personal/Manning/code/manning-smia/chapter4/licensing-service/
target/licensing-service-0.0.1-SNAPSHOT-docker-info.jar
Successfully built ostock/licensing-service:0.0.1-SNAPSHOT

BUILD SUCCESS
Total time: 21.505 s
Finished at: 2019-12-30T12:18:45-06:00

```

Dockerfile steps

Image id

Image name

Figure 4.5 Docker image build with the maven plugin by executing the mvn package dockerfile:build command.

Now that we have the docker image now, we can see it in the list of all the docker images on our system. To list all of the Docker images, we need to execute the following command:

```
docker images
```

If everything ran correctly you should see something like this:

```
REPOSITORY TAG IMAGE ID CREATED SIZE
ostock/licensing-service 0.0.1-SNAPSHOT 231fc4a87903 1 minute ago 149MB
```

Once we have the docker image, we can run it by using the following docker run command:

```
docker run ostock/licensing-service:0.0.1-SNAPSHOT
```

NOTE You can also use the -d option in the docker run command to run the container in the background. docker run -d ostock/licensing-service:0.0.1-SNAPSHOT

The above command starts the container, so in order to see all the running containers in your system, you can execute the following command:

```
docker ps
```

The docker ps command will open all the running containers with their correspondent id, image, command, creation date,

status, ports, and names. In case you need to stop the container, you can execute the following command with the correspondent container id.

```
docker stop <container_id>
```

4.5.2 Creating Docker images with Spring Boot

In this section, I will give a brief detail on how to create Docker images using the new features released in Spring Boot 2.3. It's important to highlight that the prerequisites for this new feature are:

1. Have Docker and Docker-compose installed.
2. Have a Spring Boot Application with an equal or greater version of 2.3 in your microservice.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.4.0</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
```

These new features help improve the Buildpacks support and layered jars techniques.

BUILDPACKS

Buildpacks are tools that provide application and framework dependencies, transforming our source code into a runnable

application image. In other words, Buildpacks detect and obtain everything the application needs to run.

Spring Boot 2.3.0 introduces the support for building Docker images using Cloud Native Buildpacks. This support has been added to the following plugins using the `spring-boot:build-image` goal for Maven and the `bootBuildImage` task for Gradle.

- Spring Boot Maven Plugin (<https://docs.spring.io/spring-boot/docs/2.3.0.M1/maven-plugin/html/>)
- Spring Boot Gradle Plugin (<https://docs.spring.io/spring-boot/docs/2.3.0.M1/gradle-plugin/reference/html/>)

For purposes of this book, I will only explain how to use the Maven scenario. To build the image using this new feature, just execute the following command from the root directory of your spring boot microservice.

```
./mvnw spring-boot:build-image
```

Once the command is executed you should be able to see a similar output to the one shown in the next code snippet.

```
[INFO] [creator] Setting default process type 'web'  
[INFO] [creator] *** Images (045116f040d2):  
[INFO] [creator] docker.io/library/licensing-service:0.0.1-SNAPSHOT  
[INFO]  
[INFO] Successfully built image ' docker.io/library/licensing-service:0.0.1-SNAPSHOT'
```

If you want to customize the name of the image created you can just add the following plugin to your `pom.xml` file and

define the name under the configuration section, as shown in the following code snippet.

```
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<image>
<name>${docker.image.prefix}/${project.artifactId}:latest</name>
</image>
</configuration>
</plugin>
```

Once the image was built successfully you can execute the following command to start the container via Docker.

```
docker run -it -p8080:8080 docker.io/library/licensing-service:0.0.1-SNAPSHOT
```

LAYERED JARS

Spring Boot introduced a new jar layout called LAYERED_JAR. In this new format, the lib and classes folders have been split up and categorized into layers. This layering was created to separate code based on how likely it is to change between the builds and leave the necessary information for the build. This is another excellent option to use in case you don't want to use the buildpacks.

To extract the layers of our microservice let's execute the following steps.

1. Add the layer configuration into the pom.xml file.
2. Package the application.

3. Execute the jarmode system property with the layertools jar mode.
4. Create the Dockerfile.
5. Build and run the image.

The first step is to add the configuration of the layers into the spring-boot-maven-plugin in the pom.xml.

```
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<layers>
<enabled>true</enabled>
</layers>
</configuration>
</plugin>
```

Once we have the configuration set up in our pom.xml file, let's execute the following command to rebuild our spring boot jar.

```
mvn clean package
```

Once the jar file is created, let's execute the following command in the root directory of our application to display the layers and the order that they should be added in our Dockerfile.

```
java -Djarmode=layertools -jar target/licensing-service-0.0.1-SNAPSHOT.jar list
```

Once executed, you should see a similar output to the one shown in the following code snippet.

```
dependencies
spring-boot-loader
snapshot-dependencies
application
```

Now that we have the layer information, let's continue with the fourth step that is creating our Dockerfile. The following code listing 4.6 shows you how.

Listing 4.6 Dockerfile Layered JARS

```
FROM openjdk:11-slim as build
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract
FROM openjdk:11-slim
WORKDIR application
COPY --from=build application/dependencies/ ./ #A
COPY --from=build application/spring-boot-loader/ ./ #A
COPY --from=build application/snapshot-dependencies/ ./ #A
COPY --from=build application/application/ ./ #A
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"] #B
```

#A Copies each layer displayed in the result of the jarmode command.

#B Indicates that is going to use the
org.springframework.boot.loader.JarLauncher to execute the application.

Finally, let's execute the build and run Docker commands in the root directory of our microservice.

```
docker build . --tag licensing-service
docker run -it -p8080:8080 licensing-service:latest
```

4.5.3 Launching the services with Docker Compose

Docker Compose is installed as part of the Docker installation process. It's a service orchestration tool that allows you to define services as a group and then launch together as a single unit. Docker Compose includes capabilities for also defining environment variables with each service.

Docker Compose uses a YAML file for defining the services that are going to be launched. Each chapter in this book has a file called “<<chapter>>/docker-compose.yml”. This file contains the service definitions used to launch the services in the chapter. Let's take a look to our first docker-compose.yml file. Listing 4.7 shows how your docker-compose.yml should look like.

Listing 4.7 docker-compose.yml

```
version: '3.7'
services:
  licensingservice: #A
    image: ostock/licensing-service:0.0.1-SNAPSHOT #B
    ports:
      - "8080:8080" #C
    environment:
      - "SPRING_PROFILES_ACTIVE=dev" #D
    networks:
      backend: #E
    aliases:
      - "licenseeservice" #F
    networks:
      backend: #G
    driver: bridge
```

#A 1. Each service being launched has a label applied to it. This will become the DNS entry for the Docker instance when it's started and is how other services can access it.

#B 2. Docker Compose will first try to find the target image to be started in the local Docker repository. If it can't find it, it will check the central Docker hub (<http://hub.docker.com>)

#C 3. This entry defines the port numbers on the started Docker container that will be exposed to the outside world.

#D 4. The environment tag is used to pass along environment variables to the starting Docker image. In this case, the SPRING_PROFILES_ACTIVE environment variable will be set on the starting Docker image.

#E 5. Specifies the network name where the service belongs.

#F 6. Specifies the alternative hostname for the service on the network.

#G 7. Creates a custom network named backend with the default type bridge

Now that we have the docker-compose.yml we can start our services by executing the docker-compose up command in the directory where the docker-compose.yml file is. Once it is executed, you should see a result similar to figure 4.6

NOTE If you are yet not familiarized with the SPRING_PROFILES_ACTIVE variable, don't worry about it, I'll cover this in the next chapter to manage different profiles in our microservice.



```
Starting licensing-service_licensingservice_1 ... done
Attaching to licensing-service_licensingservice_1
[...]
main| c.o.license.LicenseServiceApplication : Starting LicenseServiceApplication on 62099d3ea36 with PID 1 [/app started by root in /]
main| c.o.license.LicenseServiceApplication : The following profiles are active: dev
main| o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 [http]
main| o.apache.catalina.core.StandardService : Starting service [Tomcat]
main| org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.26]
main| o.a.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
main| o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1917 ms
main| o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
main| o.s.e.a.e.WebEndpointLinksResolver : Exposing 1 endpoint(s) beneath base path '/v1'
main| o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 [http] with context path ''
main| c.o.license.LicenseServiceApplication : Started LicenseServiceApplication in 3,869 seconds (JVM running for 4,539)
```

Figure 4.6 docker-compose console log, where it shows that the licensing service is up and running with the SPRING_PROFILES_ACTIVE variable specified in the docker-compose.yml.

The above command starts the container, so you can execute the docker ps command to see all the running containers.

NOTE All the Docker containers used in this book are ephemeral—they won't retain their state when they're started and stopped. Keep this in mind

if you start playing with code, and you see your data disappear after you restart your containers. If you are interested on how to save the state of a container take a look at the docker commit command.

Now that we know what containers are and how to integrate docker with our microservices, let's continue with the next chapter in order to create our Spring Cloud configuration server.

4.6 Summary

- The containers allow us to execute successfully the software we are developing in any technology environment from the developer's machine to a physical or virtual enterprise server.
- A virtual machine (VM) is a software that allows us to emulate the operation of a computer within another computer. They are based on a hypervisor that emulates the complete physical machine allocating the desired amount of system memory, processor cores, disk storage, and other technological resources such as networks, PCI add-ons, and more.
- Containers are another operating system virtualization method that allows you to run an application with its dependent elements in an isolated and independent environment.
- **The use of containers reduces general costs by creating lightweight virtual machinery that speeds up the process and therefore reduces the costs of each project.**
- Docker is a popular open-source container engine based on Linux containers, created by Solomon Hykes, founder and CEO of dotCloud on March 15, 2013.
- Docker is composed of the following components: Docker engines, clients, registries, images, containers, volumes, and networks.

- A dockerfile is a simple text file that contains a list of instructions and commands that the docker client calls to create and prepare an image. Docker created this file to automate the image creation process. The commands used in the dockerfile are similar to the Linux commands what makes it easier to understand.
- Docker Compose is a service orchestration tool that allows you to define services as a group and then launch together as a single unit.
- Docker Compose is installed as part of the Docker installation process.
- Dockerfile-maven-plugin integrates Maven with Docker.

5 Controlling your configuration with Spring Cloud Configuration Server

This chapter covers

- Separating the service configuration from the service code
- Configuring a Spring Cloud configuration server
- Integrating a Spring Boot microservice with a configuration server
- Encrypting sensitive properties
- Integrating the Spring Cloud configuration server with Hashicorp Vault

Software developers have always heard about the importance of keeping the application configuration separate from the code. In most scenarios, this means not using hard-coded values in the code. Forgetting this principle can make changing an application more complicated, because every time a change to the configuration is made, the application has to be recompiled and/or redeployed. Completely separating the configuration information from the application code allows developers and ops to make changes to configuration without going through a recompile process. But

it also introduces complexity, because now developers have another artifact to manage and deploy with the application.

Many developers turn to property files (or YAML, JSON, or XML) to store their configuration information. Configuring your application in these files becomes an easy task, so easy that most developers never do more operationalization than placing their configuration file under source control (if that) and deploying it as part of their application.

This approach might work with a small number of applications, but it quickly falls apart when dealing with cloud-based applications that may contain hundreds of microservices, where each microservice, in turn, might have multiple service instances running. Suddenly, an easy and straightforward process becomes a big deal, and an entire team has to wrestle with all of the configuration files. For example, let's imagine we have hundreds of microservices, and each microservice contains different configurations for three environments. If we don't manage those files outside of the application every time there is a change, we must search for the file in the code repository, follow the integration process (if there is one), and restart the application.

To avoid that catastrophic scenario, best practices for cloud-based microservices development indicate developers should:

1. Completely separate the configuration of an application from the actual code being deployed
2. Build immutable application images that never change as they are promoted through your environments

3. Inject any application configuration information at startup time of the server through either environment variables or a centralized repository the microservices read on startup

This chapter will introduce you to the core principles and patterns needed to manage application configuration data in a cloud-based microservice application.

5.1 On managing configuration (and complexity)

Managing application configuration is critical for microservices running in the cloud because microservice instances need to be launched quickly with minimal human intervention. Every time a human being needs to manually configure or touch a service to get it deployed is an opportunity for configuration drift, an unexpected outage and a lag-time in responding to scalability challenges with the application.

Let's begin our discussion about application configuration management by establishing four principles we want to follow:

1. **Segregate.** We want to completely separate the services configuration information from the actual physical deployment of a service. Application configuration shouldn't be deployed with the service instance. Instead, configuration information should either be passed to the starting service as environment

variables or read from a centralized repository when the service starts.

2. **Abstract.** Abstract the access of the configuration data behind a service interface. Instead of writing a code that directly accesses the service repository (read the data out of a file or a database using JDBC) have the application use a REST-based JSON service to retrieve the configuration data.
3. **Centralize.** Because a cloud-based application might literally have hundreds of services, it's critical to minimize the number of different repositories used to hold configuration information. Centralize your application configuration into as few repositories as possible.
4. **Harden.** Because your application configuration information is going to be completely segregated from your deployed service and centralized, it's critical that whatever solution you utilize can be implemented to be highly available and redundant.

One of the key things to remember is that when you separate your configuration information outside of your actual code, you're creating an external dependency that will need to be managed and version controlled. I can't emphasize enough that the application configuration data needs to be tracked and version-controlled because mismanaged application configuration is a fertile breeding ground for difficult-to-detect bugs and unplanned outages.

5.1.1 Your configuration management architecture

As you'll remember from the previous chapters, the loading of configuration management for a microservice occurs during the bootstrapping phase of the microservice. As a reminder, figure 5.1, shows the microservice lifecycle.

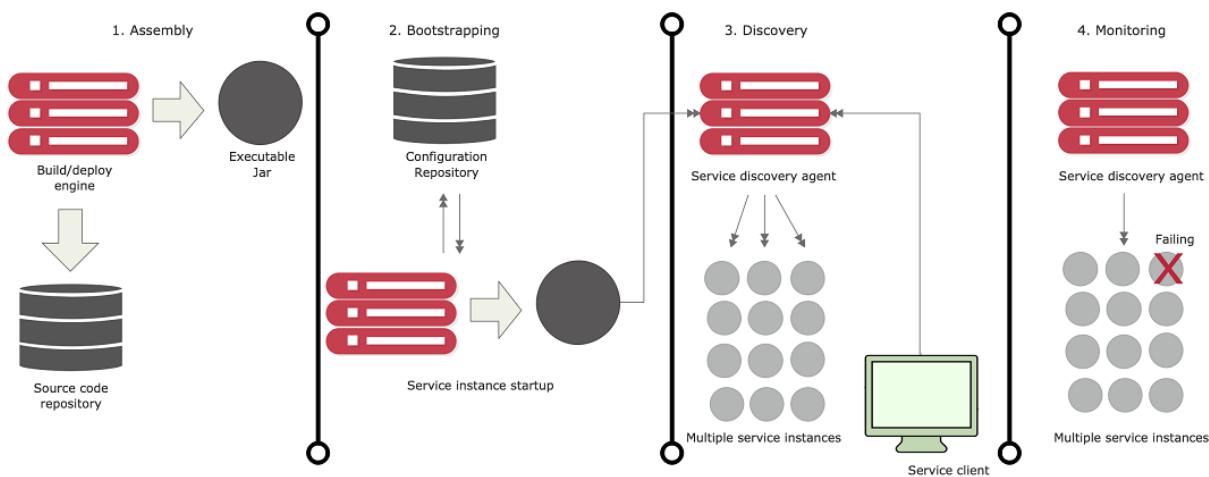


Figure 5.1 When a microservice starts up, it goes through multiple steps in its lifecycle and the application configuration data is read during the service bootstrapping phase.

Let's take the four principles we laid out earlier in section 5.1 (segregate, abstract, centralize, and harden) and see how these four principles apply when the service is bootstrapping. Figure 5.2 explores the bootstrapping process in more detail and shows how a configuration service plays a critical role in this step.

In figure 5.2, you see several activities taking place:

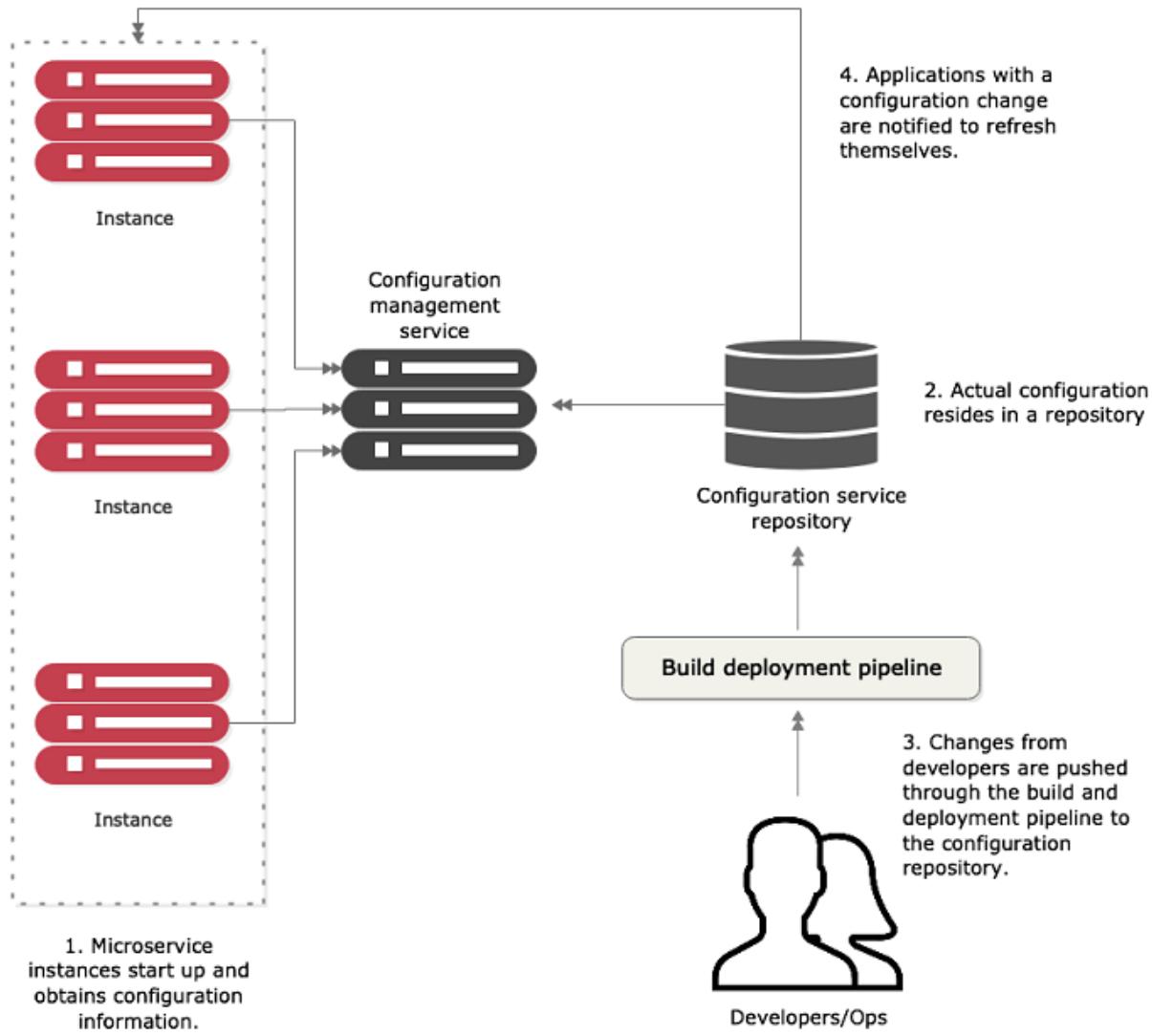


Figure 5.2 Configuration management conceptual architecture

1. When a microservice instance comes up, it calls a service endpoint to read its configuration information that's specific to the environment it's operating in. The connection information for the configuration

management (connection credentials, service endpoint, and so on) will be passed into the microservice when it starts up.

2. The actual configuration will reside in a repository. Based on the implementation of your configuration repository, you can choose to use different implementations to hold your configuration data. The implementation choices can include files under source control, relational database, key-value data store.
3. The actual management of the application configuration data occurs independently of how the application is deployed. Changes to configuration management are typically handled through the build and deployment pipeline where changes of the configuration can be tagged with version information and deployed through the different environments (development, staging, production or more).
4. When a configuration management change is made, the services that use that application configuration data must be notified of the change and refresh their copy of the application data.

At this point we've worked through the conceptual architecture that illustrates the different pieces of a configuration management pattern and how these pieces fit together. Now we are going to move on to look at the different solutions to achieve the configuration management and then see a concrete implementation.

5.1.2 Implementation choices

Fortunately, you can choose among a large number of battle-test open source projects to implement a configuration management solution. Let's look at several of the different choices available and compare them. Table 5.1 lays out these choices.

Table 5.1 Open source projects for implementing a configuration management system

Project name	Description	Characteristics
Etcd	Open source project written in Go. Used for service discovery and key-value management. Uses the raft (https://raft.github.io/) protocol for its distributed computing model.	Very fast and scalable Distributable Command-line driven Easy to use and setup
Eureka	Written by Netflix. Extremely battle-tested. Used for both service discovery and key-value management.	Distribute key-value store. Flexible; takes effort to set up Offers dynamic client refresh out of the box
Consul	Written by Hashicorp. Similar to Etcd and Eureka in features but	Fast Offers native service discovery with the

	<p>uses a different algorithm for its distributed computing model.</p>	<p>option to integrate directly with DNS Doesn't offer client dynamic refresh right out of the box</p>
ZooKeeper	<p>An Apache project that offers distributed locking capabilities. Often used as a configuration management solution for accessing key-value data.</p>	<p>Oldest, most battle-tested of the solutions The most complex to use Can be used for configuration management, but should be considered only if you're already using ZooKeeper in other pieces of your architecture</p>
Spring Cloud configuration server	<p>An open source project that offers a general configuration management solution with different back ends. It can integrate with Git, Eureka, and Consul as a back end.</p>	<p>Non-distributed key/value store Offers tight integration for Spring and non-Spring services Can use multiple back ends for storing configuration data including a shared filesystem, Eureka, Consul, and Git</p>

All the solution in table 5.1 can easily be used to build a configuration management solution. For the examples in this chapter and throughout the rest of the book, we'll use Spring Cloud configuration server that adapts perfectly to our Spring microservices architecture. I chose this solution because

1. Spring Cloud configuration server is easy to set up and use.
2. Spring Cloud configuration integrates tightly with Spring Boot. You can literally read all your application's configuration data with a few simple-to-use annotations.
3. Spring Cloud configuration server offers multiple back ends for storing configuration data.
4. Of all of the solutions in table 5.1 Spring Cloud configuration server can integrate directly with the Git source control platform and with the Hashicorp Vault I will explain these topics later on this chapter.

For the rest of this chapter, you're going to

1. Set up a Spring Cloud configuration server and demonstrate three different mechanisms for serving application configuration data—one using the filesystem another using a Git repository and another using the Hashicorp Vault.
2. Continue to build out the licensing service to retrieve data from a database
3. Hook the Spring Cloud configuration service into your licensing service to serve up application configuration

data

5.2 Building our Spring Cloud configuration server

The Spring Cloud configuration server is a REST-based application that is built on top of Spring Boot. It is essential to highlight that the Spring Cloud configuration server doesn't come as a standalone server. Instead, you can choose to either embed it in an existing Spring Boot application or start a new Spring Boot project with the server embedded in it. The best practice is to keep things separated.

The first thing we need is to create a Spring Boot project with the Spring Initializr (<https://start.spring.io/>), to achieve that let's do the following steps:

1. Select maven as the project type.
2. Select Java as the language
3. Select the latest or more stable spring version.
4. Write com.optimagrowth as group and configserver as artifact.
5. Expand the options list and write Configuration Server as name, Configuration server as description and com.optimagrowth.configserver as package name.
6. Select the JAR packaging.
7. Select Java 11 as the java version.
8. Add the Config Server and Spring Boot Actuator dependencies.

Once filled your spring Initializr form should look like Figure 5.3 and 5.4

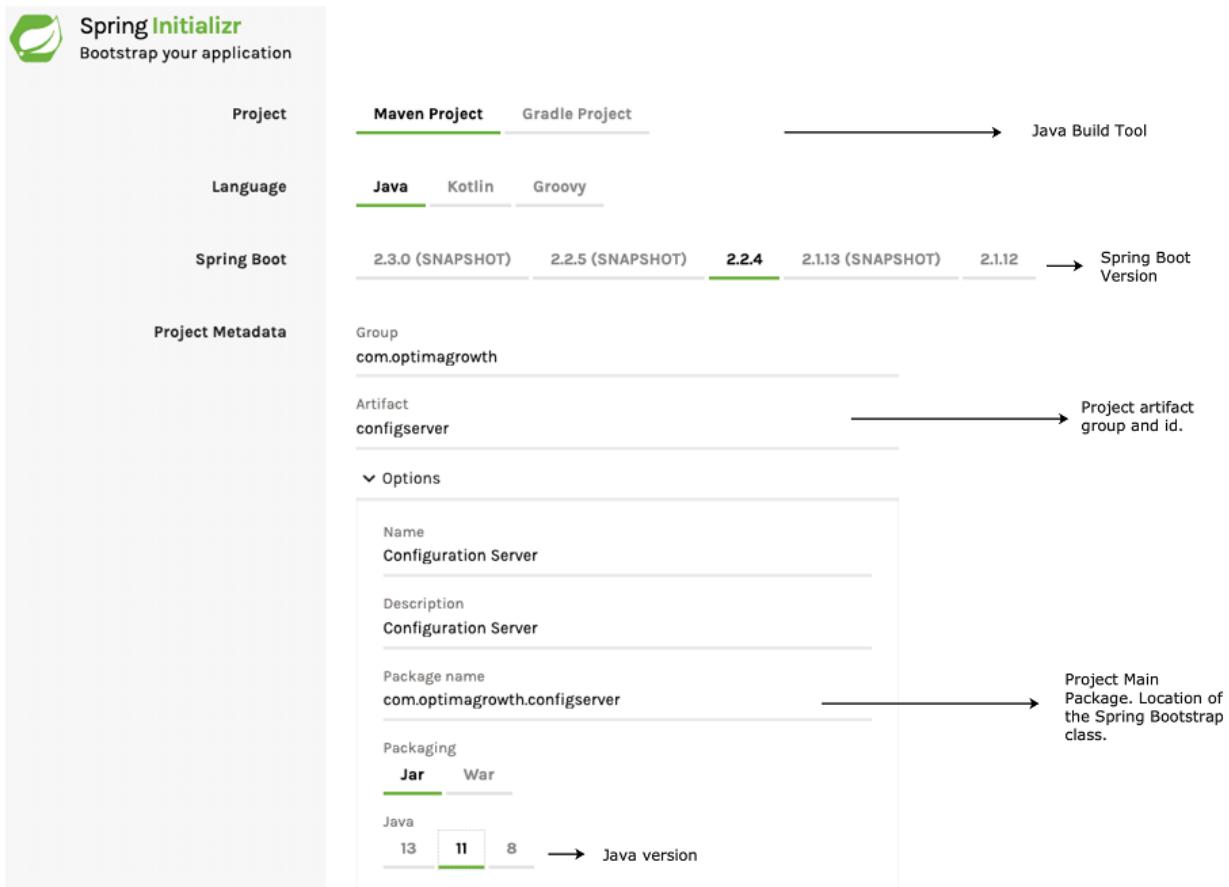


Figure 5.3 Spring Initializr configuration for the Spring Configuration server.

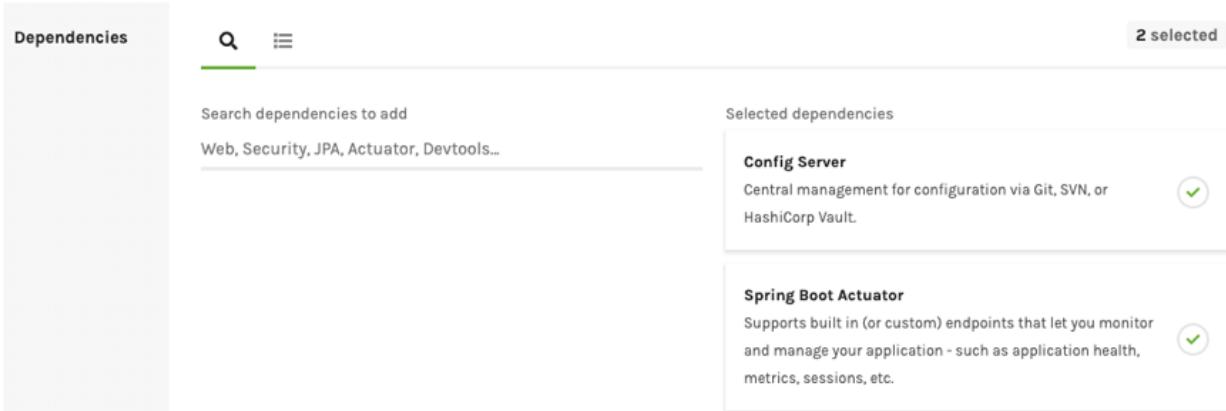


Figure 5.4 Spring Initializr dependencies Config Server and Spring Boot Actuator dependencies for the Spring Configuration server.

Once you've created and imported the project as a Maven project into your preferred Integrated Development Environment (IDE), you should have a pom.xml file in the root of the config server project that looks like the following code listing 5.1

Listing 5.1 Maven pom file for the Spring Configuration Server

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.4.RELEASE</version> #A
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.optimagrowth</groupId>
  <artifactId>configserver</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Configuration Server</name>
  <description>Configuration Server</description>
  <properties>
    <java.version>11</java.version>
    <spring-cloud.version>Hoxton.SR1</spring-cloud.version> #B
  </properties>
  <dependencies> #C
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <exclusions>
      <exclusion>
        <groupId>org.junit.vintage</groupId>
```

```

<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<dependencyManagement> #D
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>

```

#A The Spring boot version

#B The Spring Cloud version that is going to be used

#C The Spring Cloud projects and other dependencies needed to run the Spring configuration server.

#D Spring Cloud BOM definition

NOTE All the pom.xml files should contain the docker dependencies and configuration, but in order to save space I won't be adding those lines to the code listings. If you want to take a look at the docker configuration for the configuration server please visit the chapter 5 folder in the GitHub repository (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter5>)

We won't go through the entire file in detail but note a few key areas to begin with our Spring Configuration server. In the maven file in code listing 5.1, we can see four important parts. The first one is the Spring boot version. The next important part of the pom.xml file is the Spring Cloud version that we are going to use. In this example, we are going to use version Hoxton.SR1 of Spring Cloud.

The third point highlighted in the code listing the specific dependencies that we are going to use in the service and the last point is the Spring Cloud Configuration parent BOM (Bill of Materials) that we are going to use. This parent BOM contains all the third-party libraries and dependencies that are used in the cloud project and the version numbers of the individual projects that make up that version. In this example we are going to use the version defined previously in the properties section of the file. By using the BOM definition, we can guarantee that we are going to use compatible versions of the subprojects in Spring Cloud and it also means that we don't have to declare version numbers for our sub-dependencies.

Come on, ride the train, the release train

Spring Cloud uses a non-traditional mechanism for labeling Maven projects. Spring Cloud is a collection of independent subprojects. The Spring Cloud team does their releases through what they call the "release train." All the subprojects that make up Spring Cloud are packaged under one Maven bill of materials (BOM) and released as a whole. The Spring Cloud team has been using the name of London subway stops as the name of their releases, with each incrementing major release giving a London subway stop that has the next highest letter. There have been several releases such as: Angel, Brixton, Camden, Dalston, Edgware, Finchley, Greenwich and Hoxton. Hoxton is by far the newest release, but still has multiple release candidate branches for the subprojects within it.

One thing to note is that Spring Boot is released independently of the Spring Cloud release train. Therefore, different versions of Spring Boot are incompatible with different releases of Spring Cloud. You can see the version dependences between Spring Boot and Spring Cloud, along with the different subproject versions contained within the release train, by referring to the Spring Cloud website (<http://projects.spring.io/spring-cloud/>).

The next step to create our Spring Configuration config server is the set-up of one more file to define the core configuration of the server so it can be up and running. This file can be either an application.properties, application.yml, bootstrap.properties or bootstrap.yml. The bootstrap file is a specific file of Spring cloud and it is loaded before the application.properties or application.yml and is used for specifying the spring application name, the spring cloud configuration git location, encryption/decryption information and more. Specifically, the bootstrap file is loaded by a parent Spring ApplicationContext and that parent is loaded before the one that uses the application properties or yml files. As for the file extensions .yml and .properties are just different data formats and you can choose the one you prefer. From now on this book you'll see that I will use the bootstrap.yml to define the configuration of the config server and the microservices.

Now, in order to continue let's create our bootstrap.yml file in the /src/main/resources folder. This file will tell the Spring cloud configuration service what port to listen, the application name, the application profiles and the location where we are going to store the configuration data. Your bootstrap file should like the following code listing 5.2

Listing 5.2 bootstrap.yml file

```
spring:  
application:  
name: config-server #A  
server:  
port: 8071 #B
```

```
#A The Spring Cloud configuration server application name. In this case we are  
going to name it config-server.  
#B The server port.
```

There are two important parts to highlight from the previous code listing 5.2. The first one is the Application name. It is really important to name all of the services that we are going to create in our architecture for our service discovery topic that I will describe in the next chapter, chapter 6. The second point is the port where the Spring configuration server is going to be listening in order to provide the requested configuration data.

5.2.1 Setting up the Spring Cloud Config Bootstrap class

The next step to create our Spring Cloud Config service is to set up the Spring Cloud Config Bootstrap class. Every Spring Cloud service covered in this book always needs a bootstrap class that will be used to launch the service as I previously explained in the previous chapter where we created the licensing service. Remember this class contains several important parts, a Java main() method that acts as the entry point for the service to start in, and a set of Spring annotations that will tell the starting service what kind of Spring behaviors is going to launch for the service.

The following code listing 5.3 shows the bootstrap class for our Spring Cloud configuration server.

Listing 5.3 Setting up the bootstrap class

```
package com.optimagrowth.configserver;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
@SpringBootApplication #A
@EnableConfigServer #B
public class ConfigurationServerApplication {
    public static void main(String[] args) { #C
        SpringApplication.run(ConfigurationServerApplication.class, args);
    }
}
```

#A Our Spring Cloud Config service is a Spring Boot Application, so we must mark it with the @SpringBootApplication annotation.
#B The @EnableConfigServer annotation enables the service as a Spring Cloud Config service.
#C The main method that launches the service and starts the Spring container.

The next step will be defining the search location of our configuration data. Let's start with the simplest example. The filesystem.

5.2.2 Using Spring Cloud configuration server with the filesystem

The Spring Cloud configuration server uses an entry in the bootstrap.yml file to point to the repository that will hold the application configuration data. Setting up a filesystem-based repository is the easiest way to accomplish this.

To do this, let's update our bootstrap file. The following code listing 5.4 shows the contents your file should have in order to set up the filesystem repository.

Listing 5.4 bootstrap.yml with a filesystem-based repository

```
spring:
application:
name: config-server
profiles:
active: native #A
cloud:
config:
server:
#Local configuration: This locations can either of classpath or locations in the
filesystem.
native:
#Reads from a specific filesystem folder
search-locations: file:///{{FILE_PATH}} #B
server:
port: 8071
```

#A The Spring profile used that will be associated with the backend repository (filesystem).

#B The search location where the configuration files are stored.

Because we are going to use the filesystem for storing application configuration information, we must tell the Spring Cloud configuration server to run with the native profile. Remember, a Spring profile is a core feature that the Spring framework offers that allows us to map our beans to different environments such as dev, test, staging, production, native, and more.

NOTE Remember, native is just a profile created for the Spring Cloud configuration server that indicates that the configuration files are going to be retrieved or read from the classpath or filesystem.

When we are using the filesystem-based repository, we use the native profile because it is a profile in the Configuration server that doesn't use any GIT or Vault configurations. Instead, it loads the configuration data from a local classpath or a filesystem.

The last part of the bootstrap.yml shown in code listing 5.4 provides Spring Cloud configuration with the directory where

the application data resides.

```
server:  
native:  
search-locations: file:///Users/illary.huaylupo
```

The important parameter in the configuration entry is the search-locations attribute. This attribute provides a comma separated list of the directories for each application that's going to have properties managed by the configuration server. In the previous example we used the filesystem location, but we can also indicate a specific classpath to look for our configuration data. The following code will show you how.

```
server:  
native:  
search-locations: classpath:/config
```

NOTE The classpath code above will go look in the src/main/resources/config folder.

Now that we have set up our Spring Configuration server let's create our licensing service properties files. In order to make this example simpler we are going to use the classpath search location and we are going to create the licensing properties files in a config folder such as the example shown earlier.

5.2.3 Setting up the configuration files for a service

For this example, I will be using the licensing service example that we began on the initial chapters of this book as an example of how to use Spring Cloud config.

NOTE In case you didn't follow the previous chapters code listings, you can download the code created in chapter 4 from the following link:
<https://github.com/ihuaylupo/manning-smia/tree/master/chapter4>.

To keep this example simple, we will set up application configuration data for three environments: a default environment for when we run the service locally, a dev environment, and a production environment.

In Spring Cloud configuration, everything works off a hierarchy. Your application configuration is represented by the name of the application and then a property file for each environment you want to have configuration information for. In each of these environments, we'll set up two configuration properties.

- An example property that will be used directly by our licensing service
- A Spring Actuator configuration we will use in the licensing service
- Database configuration for the licensing service

Figure 5.5 illustrates how we will set up and use the Spring Cloud configuration service. One important point to highlight is that as you build out your config service, it will be another microservice running in your environment. Once it's set up, the contents of the service can be accessed via a http-based REST endpoint.

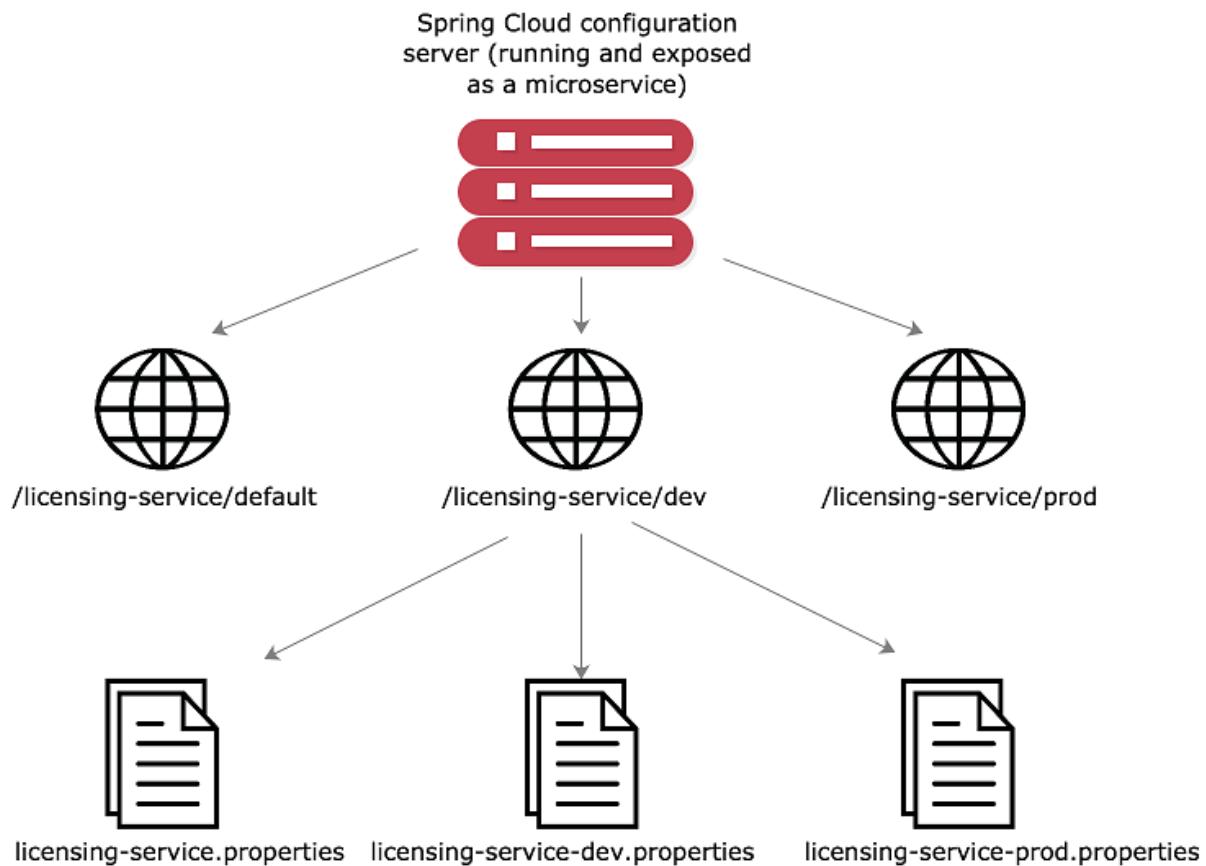


Figure 5.5 Spring Cloud configuration exposes environment-specific properties as HTTP-based endpoints.

The naming convention for the application configuration files are appname-env.properties or .yml. As you can see from the figure 5.5, the environment names translate directly into the URLs that will be accessed to browse configuration information. Later, when we start the licensing microservice example, the environment we want to run the service against is specified by the Spring Boot profile that you pass in on the command-line service startup. If a profile isn't passed in on the command line, Spring Boot will always

default to the configuration data contained in the application.properties file packaged with the application.

Here is an example of some of the application configuration data we will serve up for the licensing service. This is the data that will be contained within the configserver/src/main/resources/config/licensing-service.properties file that was referred to in figure 5.5. Here is a part of the contents of this file:

```
example.property= I AM THE DEFAULT
spring.jpa.hibernate.ddl-auto=none
spring.jpa.database=POSTGRES
spring.datasource.platform=postgres
spring.jpa.show-sql = true
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
spring.database.driverClassName= org.postgresql.Driver
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1
management.endpoints.web.exposure.include=*
management.endpoints.enabled-by-default=true
```

Think before you implement

I advise against using a filesystem-based solution for medium-to-large cloud applications. Using the filesystem approach means that you need to implement a shared file mount point for all cloud configuration servers that want to access the application configuration data. Setting up shared filesystem servers in the cloud is doable, but it puts the onus of maintaining this environment on you.

I'm showing the filesystem approach as the easiest example to use when getting your feet wet with Spring Cloud configuration server. In a later section, I'll show how to configure Spring Cloud configuration server to use GitHub and Hashicorp Vault to store your application configuration.

To continue, let's create a licensing-service-dev.properties that contains only the development data. The dev properties file should contain the following parameters.

```
example.property= I AM DEV  
spring.datasource.url = jdbc:postgresql://localhost:5433/ostock_dev  
spring.datasource.username = postgres  
spring.datasource.password = postgres
```

Now that we have enough work done to start the configuration server. Let's go ahead and start the configuration server using the mvn spring-boot:run or using the docker-compose up command.

NOTE From this point on, you will find a README file in the folder repository for each chapter containing a section called How to Use. This section will describe how to run all the services together using the docker-compose command.

The server should now come up with the Spring Boot splash screen on the command line. If you point your browser over to <http://localhost:8071/licensing-service/default> you'll see a JSON payload being returned with all of the properties contained within the licensing-service.properties file. Figure 5.6 shows the results of calling this endpoint.

```

// 20200201214255
// http://localhost:8071/licensing-service/default

{
  "name": "licensing-service",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": null,
  "state": null,
  "propertySources": [
    {
      "name": "classpath:/config/licensing-service.properties",
      "source": {
        "example.property": "I AM THE DEFAULT",
        "spring.jpa.hibernate.ddl-auto": "none",
        "spring.jpa.database": "POSTGRESQL",
        "spring.datasource.platform": "postgres",
        "spring.jpa.show-sql": "true",
        "spring.jpa.hibernate.naming-strategy": "org.hibernate.cfg.ImprovedNamingStrategy",
        "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
        "spring.database.driverClassName": "org.postgresql.Driver",
        "spring.datasource.testWhileIdle": "true",
        "spring.datasource.validationQuery": "SELECT 1",
        "management.endpoints.web.exposure.include": "*",
        "management.endpoints.enabled-by-default": "true"
      }
    }
  ]
}

```

The source file containing the properties in the config repository

Figure 5.6 Spring Cloud configuration exposes environment-specific properties as HTTP-based endpoints.

If you want to see the configuration information for the dev-based licensing service environment, hit the GET <http://localhost:8071/licensing-service/dev> endpoint. Figure 5.7 shows the result of calling this endpoint.

NOTE The port is the one we set previously on the bootstrap.yml file.

```

// 20200201214259
// http://localhost:8071/licensing-service/dev

{
  "name": "licensing-service",
  "profiles": [
    "dev"
  ],
  "label": null,
  "version": null,
  "state": null,
  "propertySources": [
    {
      "name": "classpath:/config/licensing-service-dev.properties",
      "source": {
        "example.property": "I AM DEV",
        "spring.datasource.url": "jdbc:postgresql://localhost:5433/ostock_dev",
        "spring.datasource.username": "postgres",
        "spring.datasource.password": "postgres"
      }
    },
    {
      "name": "classpath:/config/licensing-service.properties",
      "source": {
        "example.property": "I AM THE DEFAULT",
        "spring.jpa.hibernate.ddl-auto": "none",
        "spring.jpa.database": "POSTGRESQL",
        "spring.datasource.platform": "postgres",
        "spring.jpa.show-sql": "true",
        "spring.jpa.hibernate.naming-strategy": "org.hibernate.cfg.ImprovedNamingStrategy",
        "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
        "spring.database.driverClassName": "org.postgresql.Driver",
        "spring.datasource.testWhileIdle": "true",
        "spring.datasource.validationQuery": "SELECT 1",
        "management.endpoints.web.exposure.include": "*",
        "management.endpoints.enabled-by-default": "true"
      }
    }
  ]
}

```

When you request an environment-specific profile, both the requested profile and the default profile are returned.

Figure 5.7 Retrieving configuration information for the licensing service using the dev profile

If we look closely, we will see that when we hit the dev endpoint, the Spring Cloud configuration server return both the default configuration properties and dev licensing service configuration. The reason why Spring Cloud configuration returns both sets of configuration information is that the Spring framework implements a hierarchical mechanism for resolving problems. When the Spring Framework does property resolution, it will always look for the property in the

default properties first and then override the default with an environment-specific value if one is present.

In concrete terms, if you define a property in the licensing-service.properties file and don't define it in any of the other environment configuration (for example, licensing-service-dev.properties), the Spring framework will use the default value.

NOTE This isn't the behavior you'll see by directly calling the Spring Cloud configuration REST endpoint. The REST endpoint will return all configuration values for both the default and environment specific value that was called.

Now that we finish configuration everything in our Spring Cloud configuration service let's move on by integrating a Spring cloud config with our licensing microservice.

5.3 Integrating Spring Cloud Config with a Spring Boot client

In the previous chapters, we built a simple skeleton of our licensing service that did nothing more than return a hardcoded Java object representing a single licensing record. In the next example, we will build out the licensing service with a PostgreSQL database that will hold the licensing data.

Why use PostgreSQL?

PostgreSQL is considered an enterprise system and one of the most interesting and advanced options for open-source relational database management systems. PostgreSQL has many advantages compared to other relational databases, but the main two are that it offers a single license that is entirely free and open for any to use. The second advantage is that in terms of its ability and functionality, it allows us to work with more significant amounts of data without increasing the complexity.

Here are some of the main features of PostgreSQL:

Postgres uses a multi-version concurrency control that adds an image of the database status to each transaction made, allowing us to produce consistent transactions with better performance advantages.

Postgres doesn't use reading locks when it executes a transaction.

Postgres has something called Hot-StandBy. The Hot-StandBy allows the client to search in the servers while the server is in recovery or standby mode; in other words, it will enable us to perform maintenance without completely locking down the database.

Some of the main characteristics of PostgreSQL are:

It is supported by languages such as C, C++, Java, PHP, Python, and more.

It is capable of serving many clients while delivering the same information from its tables, without blockages.

It supports working with views, so users can query the data differently from how it is stored.

It is an object-relational database, allowing us to work with their data as if they were objects and offer object-orientation mechanisms.

It allows you to store and query JSON as a datatype.

We are going to communicate with the database using Spring Data and map your data from the licensing table to a Plain Old Java Object (POJO) holding the data. The database connection and a simple property are going to be read out of Spring Cloud configuration server. Figure 5.8 shows what's going to happen between the licensing service and the Spring Cloud configuration service.

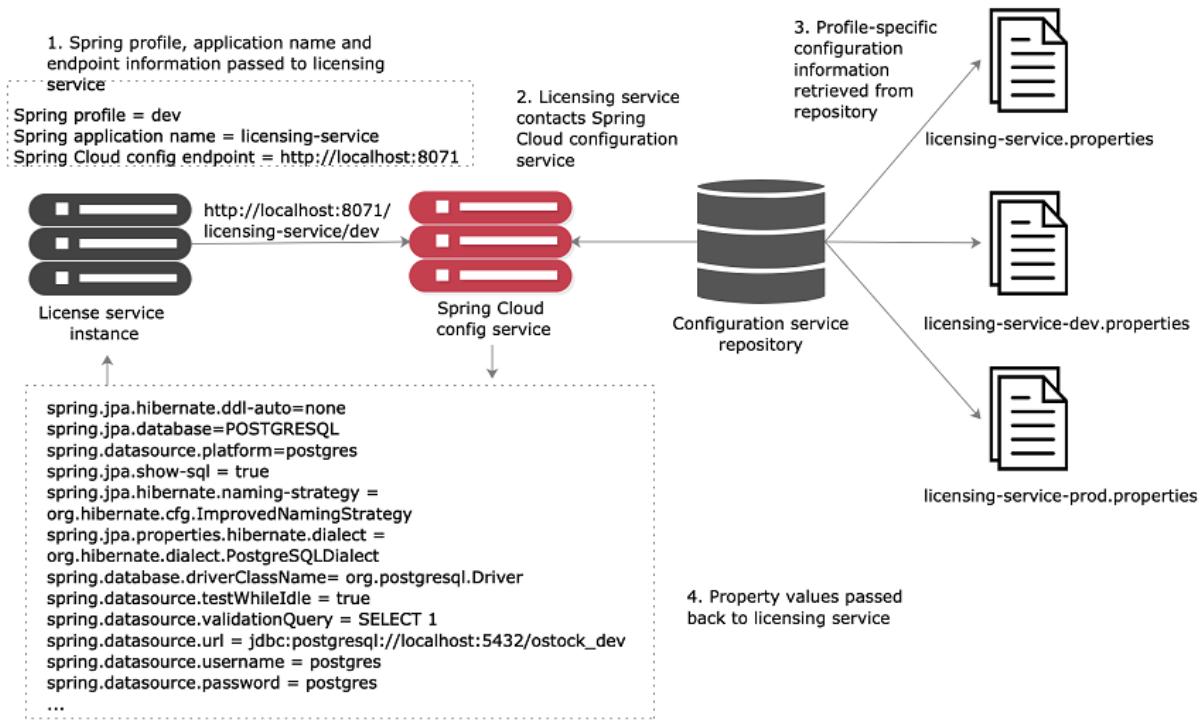


Figure 5.8 Retrieving configuration information using the dev profile

When the licensing service is first started, you'll pass it three pieces of information: The spring profile, the application name and the endpoint the licensing service should use to communicate with the Spring Cloud configuration service. The Spring profile value maps to the environment of the properties being retrieved for the Spring service. When the licensing service first boots up, it will contact the Spring Cloud Config service via an endpoint built from the Spring profile passed into it. The Spring Cloud Config service will then use the configured back end config repository (filesystem, Git, Vault) to retrieve the configuration information specific to the Spring profile value passed in on the URI. The appropriate property values are then passed back to the licensing service. The Spring Boot framework will

then inject these values into the appropriate parts of the application.

5.3.1 Setting up the licensing service Spring Cloud Config service dependencies

Let's change our focus from the configuration server to the licensing service. The first thing you need to do is add a couple of more entries to the Maven file in your licensing service. The entries that need to be added are shown in the following code listing 5.5.

Listing 5.5 Additional Maven dependencies needed by the licensing service

```
//Rest of pom.xml removed for conciseness
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId> #A
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId> #B
</dependency>
<dependency>
<groupId>org.postgresql</groupId> #C
<artifactId>postgresql</artifactId>
</dependency>
```

#A Tells Spring Boot that you should pull down the dependencies need for the Spring Cloud Config

#B Tells Spring Boot you're going to use Java Persistence API (JPA) in your service
#C Tells Spring Boot to pull down the Postgres drivers

The first dependency, the spring-cloud-starter-config, contains all the classes needed to interact with the Spring

Cloud configuration server. The second and third dependencies, spring-boot-starter-data-jpa and postgresql import the Spring Data Java Persistence API (JPA) and the Postgres JDBC drivers.

5.3.2 Configuring the licensing service to use Spring Cloud Config

After the Maven dependencies have been defined, we need to tell the licensing service where to contact the Spring Cloud configuration server. In a Spring Boot service that uses Spring Cloud Config, configuration information can be set in one of these files: bootstrap.yml, bootstrap.properties , application.yml or application.properties.

As I mentioned previously the bootstrap.yml file reads the application properties before any other configuration information used. In general, the bootstrap.yml file contains the application name for the service, the application profile, and the URI to connect to a Spring Cloud Config server. Any other configuration information that you want to keep local to the service (and not stored in Spring Cloud Config) can be set locally in the services in the application.yml file. Usually, the information you store in the application.yml file is configuration data that you might want to have available to a service even if the Spring Cloud Config service is unavailable. Both the bootstrap.yml and application.yml files are stored in a projects src/main/resources directory.

To have the licensing service communicate with your Spring Cloud Config service. These parameters can be defined in the bootstrap.yml file, in the docker-compose.yml file of the

licensing service or we can define them via JVM arguments when we start the service.

Code listing 5.6 shows how the bootstrap.yml should look like in your application if you choose this option.

Listing 5.6 Configuring the licensing services bootstrap.yml

```
spring:  
application:  
name: licensing-service #A  
profiles:  
active: dev #B  
cloud:  
config:  
uri: http://localhost:8071 #C
```

#A Specify the name of the licensing service so that Spring Cloud Config client knows which service is being looked up.

#B Specify the default profile the service should run. Profile maps to environment.

#C Specify the location of the Spring Cloud Config server.

NOTE The Spring Boot applications support two mechanisms to define a property: YAML (YAML Ain't Markup Language) and a “.” separated property -name. We chose YAML as the means for -configuring our application. The hierarchical format of YAML property values map directly to the spring.application.name, spring.profiles.active, and spring.cloud.config.uri names.

The `spring.application.name` is the name of your application (for example, `licensing-service`) and *must* map directly to the name of the config directory within your Spring Cloud configuration server.

The second property, the `spring.profiles.active`, is used to tell Spring Boot what profile the application should run as. Remember, a *profile* is a mechanism to differentiate the

configuration data consumed by the Spring Boot application. For the licensing service's profile, you'll support the environment the service is going to map directly to in your cloud configuration environment. For instance, by passing in dev as our profile, the Spring Cloud config server will use the dev properties. If you don't set a profile, the licensing service will use the default profile.

The third and last property, the `spring.cloud.config.uri`, is the location where the licensing service should look for the Spring Cloud configuration server endpoint. In this example, the licensing service will look for the configuration server at `http://localhost:8071`. Later in the chapter you'll see how to override the different properties defined in the `bootstrap.yml` and `application.yml` files on application startup. This will allow you to tell the licensing microservice which environment it should be running in.

Now, if you bring up the Spring Cloud configuration service, with the corresponding Postgres database running on your local machine, you can launch the licensing service using its dev profile. This is done by changing to the `licensing-services` directory and issuing the following commands:

```
mvn spring-boot:run
```

NOTE You need to launch first the configuration server to retrieve the configuration data for the licensing service.

By running this command without any properties set, the licensing server will automatically attempt to connect to the Spring Cloud configuration server using the endpoint

(<http://localhost:8071>) and the active profile (dev) defined previously in the bootstrap.yml file of the licensing service.

If you want to override these default values and point to another environment, you can do this by compiling the licensing-service project down to a JAR and then run the JAR with a -D system property override. The following command line call demonstrates how to launch the licensing service passing all the commands via JVM arguments.

```
java -Dspring.cloud.config.uri=http://localhost:8071 \
-Dspring.profiles.active=dev \
-jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

With the previous command line, we are overriding the two parameters: spring.cloud.config.uri and spring.profiles.active.

NOTE If you try to run the licensing service downloaded from the GitHub repository (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter5>) from your desktop using the previous Java command, it will fail because of two main reasons. The first one is that you don't have a desktop Postgres server running, and the second one is because the source code in the GitHub repository uses encryption on the config server. We'll cover using encryption later in the chapter. The previous example demonstrates how to override Spring properties via the command line.

In the examples we are hard-coding the values to pass in to the -D parameter values. In the cloud, most of the application config data you need will be in your configuration server.

All the code examples for each chapter can be completely run from within Docker containers. With Docker, you simulate

different environments through environment-specific Docker-compose files that orchestrate the startup of all of your services. Environment-specific values needed by the containers are passed in as environment variables to the container. For example, to start your licensing service in a dev environment, the dev/docker-compose.yml file contains the following entry for the licensing-service:

Listing 5.7 Dev docker-compose.yml

```
licensingservice:  
  image: ostock/licensing-service:0.0.1-SNAPSHOT  
  ports:  
    - "8080:8080"  
  environment: #A  
    SPRING_PROFILES_ACTIVE: "dev" #B  
    SPRING_CLOUD_CONFIG_URI: http://configserver:8071 #C
```

#A Specifies the start of the environment variables for the licensing-service container
#B The SPRING_PROFILES_ACTIVE environment variable is passed to the Spring Boot service command-line and tells Spring Boot what profile should be run.
#C The endpoint of the config service

The environment entry in the file contains the values of two variables SPRING_PROFILES_ACTIVE, which is the Spring Boot profile the licensing service is going to run under. The SPRING_CLOUD_CONFIG_URI is passed to your licensing service and defines the Spring Cloud configuration server instance the service is going to read its configuration data from.

Once you have the docker-compose set up, you can know run the services just by executing the following command where the docker-compose file is located:

```
docker-compose up
```

Because you enhance all your services with introspection capabilities via Spring Boot Actuator, you can confirm the environment you are running against by hitting <http://localhost:8080/actuator/env>. The /env endpoint will provide a complete list of the configuration information about the service, including the properties and endpoints the service has booted with, as shown in figure 5.9.

```
{
  "activeProfiles": [
    "dev"
  ],
  "propertySources": [
    {
      "name": "server.ports",
      "properties": {
        "local.server.port": {
          "value": 8080
        }
      }
    },
    {
      "name": "bootstrapProperties-classpath:/config/licensing-service-dev.properties",
      "properties": {
        "spring.datasource.username": {
          "value": "postgres"
        },
        "spring.datasource.url": {
          "value": "jdbc:postgresql://localhost:5433/ostock_dev"
        },
        "example.property": {
          "value": "I AM DEV"
        },
        "spring.datasource.password": {
          "value": "*****"
        }
      }
    },
    {
      "name": "bootstrapProperties-classpath:/config/licensing-service.properties",
      "properties": {
        "management.endpoints.web.exposure.include": {
          "value": "*"
        },
        "spring.jpa.properties.hibernate.dialect": {
          "value": "org.hibernate.dialect.PostgreSQLDialect"
        },
      }
    }
  ]
}
```

Figure 5.9 The licensing configuration service can be checked by calling the /actuator/env endpoint. In this example, you can see how both the licensing-service.properties and the licensing-service-dev.properties are being displayed.

On exposing too much information

Every organization is going to have different rules about how to implement security around their services. Many organizations believe services shouldn't broadcast any information about themselves and won't allow things like a /env endpoint to be active on a service as they believe (rightfully so) that this will provide too much information for a potential hacker. Spring Boot provides a wealth of capabilities on how to configure what information is returned by the Spring Actuators endpoints that are outside the scope of this book. Craig Walls' excellent book, *Spring Boot in Action*, covers this subject in detail, and I highly recommend that you review your corporate security policies and Walls' book to provide the right level of detail you want exposed through Spring Actuator.

5.3.3 Wiring in a data source using Spring Cloud configuration server

At this point, you have the database configuration information being directly injected into your microservice. With the database configuration set, configuring your licensing microservice becomes an exercise in using standard Spring components to build and retrieve the data from the Postgres database. In order to continue, with the example we need to refactor the licensing into different classes with each class having separate responsibilities. These classes are shown in table 5.2.

Table 5.2 Licensing service classes and locations

Class Name	Location
LicensingService	LicensingService.java

License	com.optimagrowth.license.model
LicenseRepository	com.optimagrowth.license.repository
LicenseService	com.optimagrowth.license.service

The License class is the model class that will hold the data retrieved from your licensing database. The following code listing 5.8 shows the code for the License class.

Listing 5.8 The JPA model code for a single license record

```
package com.optimagrowth.license.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import org.springframework.hateoas.RepresentationModel;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
@Getter @Setter @ToString
@Entity #A
@Table(name="licenses") #B
public class License {
@Id #C
@Column(name = "license_id", nullable = false) #D
private String licenseId;
private String description;
@Column(name = "organization_id", nullable = false)
private String organizationId;
@Column(name = "product_name", nullable = false)
private String productName;
@Column(name = "license_type", nullable = false)
private String licenseType;
@Column(name="comment")
private String comment;
public License withComment(String comment){
this.setComment(comment);
return this;
}
}
```

#A @Entity tells Spring that this is a JPA class.

#B @Table maps to the database table.

#C @Id marks this field as a primary key.

#D @Column maps the field to a specific database table.

The class uses several Java Persistence API (JPA) annotations that help the Spring Data framework map the data from the licenses table in the Postgres database to the Java object. The `@Entity` annotation lets Spring know that this Java POJO is going to be mapping objects that will hold data. The `@Table` annotation tells Spring/JPA what database table should be mapped. The `@Id` annotation identifies the primary key for the database. Finally, each one of the columns from the database that is going to be mapped to individual properties is marked with a `@Column` attribute.

NOTE Remember, if your attribute has the same name as the database column, you don't need to add the `@Column`.

The Spring Data and JPA framework provides your basic CRUD methods for accessing a database. For example:

- **count()**. Returns the number of entities available.
- **delete(entity)**. Deletes a given entity.
- **deleteAll()**. Deletes all entities managed by the repository.
- **deleteAll(entities)**. Deletes the given entities.
- **deleteById(id)**. Deletes the entity with the given id.
- **existsById(id)**. Returns whether an entity with the given id exists.
- **findAll()**. Returns all instances of the type.
- **findAllById(ids)**. Returns all instances of the type T with the given IDs.
- **findById(ID id)**. Retrieves an entity by its id.
- **save(entity)**. Saves a given entity.
- **saveAll(entities)**. Saves all given entities.

If you want to build methods beyond that, you can use a Spring Data Repository interface and basic naming conventions to build those methods. Spring will at startup parse the name of the methods from the Repository interface, convert them over to a SQL statement based on the names, and then generate a dynamic proxy class under the covers to do the work. The repository for the licensing service is shown in the following code listing 5.9

Listing 5.9 LicenseRepository interface defines the query methods

```
package com.optimagrowth.license.repository;
import java.util.List;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.optimagrowth.license.model.License;
@Repository #A
public interface LicenseRepository extends CrudRepository<License, String> { #B
    public List<License> findByOrganizationId(String organizationId); #C
    public License findByOrganizationIdAndLicenseId(String organizationId,
    String licenseId);
}
```

#A Tells Spring Boot that this is a JPA repository class. Annotation is optional when we already extend from a CrudRepository.
#B Defines that you're extending the Spring CrudRepository
#C Individual query methods are parsed by Spring into a SELECT...FROM query.

The repository interface, LicenseRepository, is marked with the @Repository annotation which tells Spring that it should treat this interface as a repository and generate a dynamic proxy for it. The dynamic proxy in this case provides a set of fully-feature ready to use objects. Spring offers different types of repositories for data access. You've chosen to use the Spring CrudRepository base class to extend your LicenseRepository class. The CrudRepository base class

contains basic CRUD methods. In addition to the CRUD (Create, Read, Update and Delete) methods extended from CrudRepository, you've added two custom query methods for retrieving data from the licensing table in the LicenseRepository interface. The Spring Data framework will pull apart the name of the methods to build a query to access the underlying data.

NOTE The Spring Data framework provides an abstraction layer over various database platforms and isn't limited to relational databases. NoSQL databases such as MongoDB and Cassandra are also supported.

Unlike the previous incarnation of the licensing service in chapter 3, you've now separated the business and data access logic for the licensing service out of the LicenseController and into a standalone Service class called LicenseService. The license service is shown in the following code listing 5.10.

NOTE Between this licensingService class and the previous version seen in the previous chapters there are a lot of changes, because I added the database connection. So, feel free to download the file from the following link. (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter5/licensing-service/src/main/java/com/optimagrowth/license/service/LicenseService.java>)

Listing 5.10 LicenseService class used to execute database commands

```
@Service
public class LicenseService {
    @Autowired
    MessageSource messages;
    @Autowired
    private LicenseRepository licenseRepository;
    @Autowired
    ServiceConfig config;
    public License getLicense(String licenseId, String organizationId){
```

```

License license =
licenseRepository.findByOrganizationIdAndLicenseId(organizationId, licenseId);
if (null == license) {
throw new
IllegalArgumentException(String.format(messages.getMessage("license.search.error.m
essage", null, null),licenseId, organizationId));
}
return license.withComment(config.getProperty());
}
public License createLicense(License license){
license.setLicenseId(UUID.randomUUID().toString());
licenseRepository.save(license);
return license.withComment(config.getProperty());
}
public License updateLicense(License license){
licenseRepository.save(license);
return license.withComment(config.getProperty());
}
public String deleteLicense(String licenseId){
String responseMessage = null;
License license = new License();
license.setLicenseId(licenseId);
licenseRepository.delete(license);
responseMessage = String.format(messages.getMessage("license.delete.message",
null, null),licenseId);
return responseMessage;
}
}

```

The controller, service, and repository classes are wired together using the standard Spring @Autowired annotation.

5.3.4 Directly reading properties using the @ConfigurationProperties annotation

In the LicenseService class in the previous section, you might have noticed that you're setting the license.withComment() value in the getLicense() code with a value from the config.getProperty() class. The code being referred to is shown here:

```
return license.withComment(config.getProperty());
```

If you look at the `com.optimagrowth.license.config.ServiceConfig.java` class, you'll see that the class is annotated with the `@ConfigurationProperties(prefix= "example")` annotation. The following code listing 5.11 shows the `@ConfigurationProperties` annotation being used.

Listing 5.11 ServiceConfig used to centralize application properties

```
package com.optimagrowth.license.config;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@ConfigurationProperties(prefix = "example")
public class ServiceConfig{
    private String property;
    public String getProperty(){
        return property;
    }
}
```

While Spring Data “auto-magically” injects the configuration data for the database into a database connection object, all other custom properties can be injected using the `@ConfigurationProperties` annotation. With the previous example, the `@ConfigurationProperties(prefix= "example")` annotation pulls all the `example.properties` from the Spring Cloud configuration server and injects it into the `property` attribute on the `ServiceConfig` class.

NOTE While it's possible to directly inject configuration values into properties in individual classes, I've found it useful to centralize all of the configuration information into a single configuration class and then inject the configuration class into where it's needed.

5.3.5 Refreshing your properties using Spring Cloud configuration server

One of the first questions that comes up from development teams when they want to use the Spring Cloud configuration server is how can they dynamically refresh their applications when a property changes. The Spring Cloud configuration server will always serve the latest version of a property. Changes made to a property via its underlying repository will be up to date.

However, Spring Boot applications will only read their properties at startup time, so property changes made in the Spring Cloud configuration server won't be automatically picked up by the Spring Boot application. Spring Boot Actuator does offer a `@RefreshScope` annotation that will allow a development team to access a `/refresh` endpoint that will force the Spring Boot application to reread its application configuration. The following code listing 5.12 shows the `@RefreshScope` annotation in action.

Listing 5.12 The `@RefreshScope` annotation

```
package com.optimagrowth.license;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
@SpringBootApplication
@RefreshScope
public class LicenseServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(LicenseServiceApplication.class, args);
    }
}
```

Note a couple of things about the `@RefreshScope` annotation. The annotation will only reload the custom Spring properties you have in your application configuration. Items such as your database configuration that are used by Spring Data won't be reloaded by the `@RefreshScope` annotation.

On refreshing microservices

When using Spring Cloud configuration service with microservices, one thing you need to consider before you dynamically change properties is that you might have multiple instances of the same service running, and you'll need to refresh all of those services with their new application configurations. There are several ways you can approach this problem:

Spring Cloud configuration service does offer a “push”-based mechanism called Spring Cloud Bus that will allow the Spring Cloud configuration server to publish to all the clients using the service that a change has occurred. Spring Cloud Bus requires an extra piece of middleware running (RabbitMQ). This is an extremely useful means of detecting changes, but not all Spring Cloud configuration backends support the “push” mechanism (that is, the Consul server).

In the next chapter you'll use Spring Service Discovery and Eureka to register all instances of a service. One technique I've used to handle application configuration refresh events is to refresh the application properties in Spring Cloud configuration and then write a simple script to query the service discovery engine to find all instances of a service and call the `/refresh` endpoint directly.

Finally, you can restart all the servers or containers to pick up the new property. This is a trivial exercise, especially if you're running your services in a container service such as Docker. Restarting Docker containers literally takes seconds and will force a reread of the application configuration.

Remember, cloud-based servers are ephemeral. Don't be afraid to start new instances of a service with their new configuration, direct traffic to the new services, and then tear down the old ones.

5.3.6 Using Spring Cloud configuration server with Git

As mentioned earlier, using a filesystem as the backend repository for Spring Cloud configuration server can be impractical for a cloud-based application because the development team has to set up and manage a shared filesystem that's mounted on all instances of the Cloud configuration server.

Spring Cloud configuration server integrates with different backend repositories that can be used to host application configuration properties. One I've used successfully is to use Spring Cloud configuration server with a Git source control repository.

By using Git, you can get all the benefits of putting your configuration management properties under source control and provide an easy mechanism to integrate the deployment of your property configuration files in your build and deployment pipeline.

To use Git, we need to add the following configuration to the Spring Cloud configuration service `bootstrap.yml` file with the following listing's configuration. Code listing 5.13 shows how your `bootstrap` file should look like.

Listing 5.13 Adding GIT to the Spring Cloud `bootstrap.yml`

```
spring:  
application:  
name: config-server  
profiles:
```

```
active:
- native, git #A
cloud:
config:
server:
native:
search-locations: classpath:/config
git: #B
uri: https://github.com/ihuaylupo/config.git #C
searchPaths: licensingservice #D
server:
port: 8071
```

#A Maps all the profiles this is a comma-separated list

#B Tells Spring Cloud Config to use Git as a backend repository

#C Tells Spring Cloud Config the URL to the Git server and Git repo

#D Tells Spring Cloud Config what the path in Git is to look for config files

The four key pieces of configuration in the previous example are the `spring.profiles.active`, `spring.cloud.config.server.git`, `spring.cloud.config.server.git.uri`, and the `spring.cloud.config.server.git.searchPaths` properties. The `spring.profiles.active` sets all the active profiles the Spring Configuration service will have. This comma-separated list uses the same precedence rules of a Spring Boot application, active profiles have precedence over the default profiles, and the last profile is the one that wins. The `spring.cloud.config.server.git` property tells the Spring Cloud configuration server to use a non-filesystem-based backend repository. In the previous example you're going to connect to the cloud-based Git repository, GitHub.

NOTE In case you have a GitHub authentication you need to set the username and password, personal token or SSH configuration in the git configuration on the `bootstrap.yml` of the configuration server.

The `spring.cloud.config.server.git.uri` properties provide the URL of the repository you're connecting to. Finally, the `spring.cloud.config.server.git.searchPaths` property tells the

Spring Cloud Config server the relative paths on the Git repository that should be searched when the Cloud configuration server comes up. Like the filesystem version of the configuration, the value in the `spring.cloud.config.server.git.searchPaths` attribute will be a comma-separated list for each service hosted by the configuration service.

NOTE The default implementation of Environment Repository in the Spring Cloud config is the Git backend.

5.3.7 Integrating Vault with Spring Cloud Config service

As mentioned earlier, also there is another backend repository that we are going to use and it is the Hashicorp Vault. Vault is a tool that allows us to securely access secrets. We can define secrets as any piece of information we want to have a restricted control access such as password, certificates, API keys and more.

To configure vault in our Spring Configuration service we must add the `vault` profile. This profile enables the integration with Vault and allows us to securely store the application properties of our microservices. To achieve this integration, I will be using docker to create a vault container.

The following command will create a development vault container for us.

```
docker run -d -p 8200:8200 --name vault -e 'VAULT_DEV_ROOT_TOKEN_ID=myroot' -e 'VAULT_DEV_LISTEN_ADDRESS=0.0.0.0:8200' vault
```

The docker run command contains the following parameters:

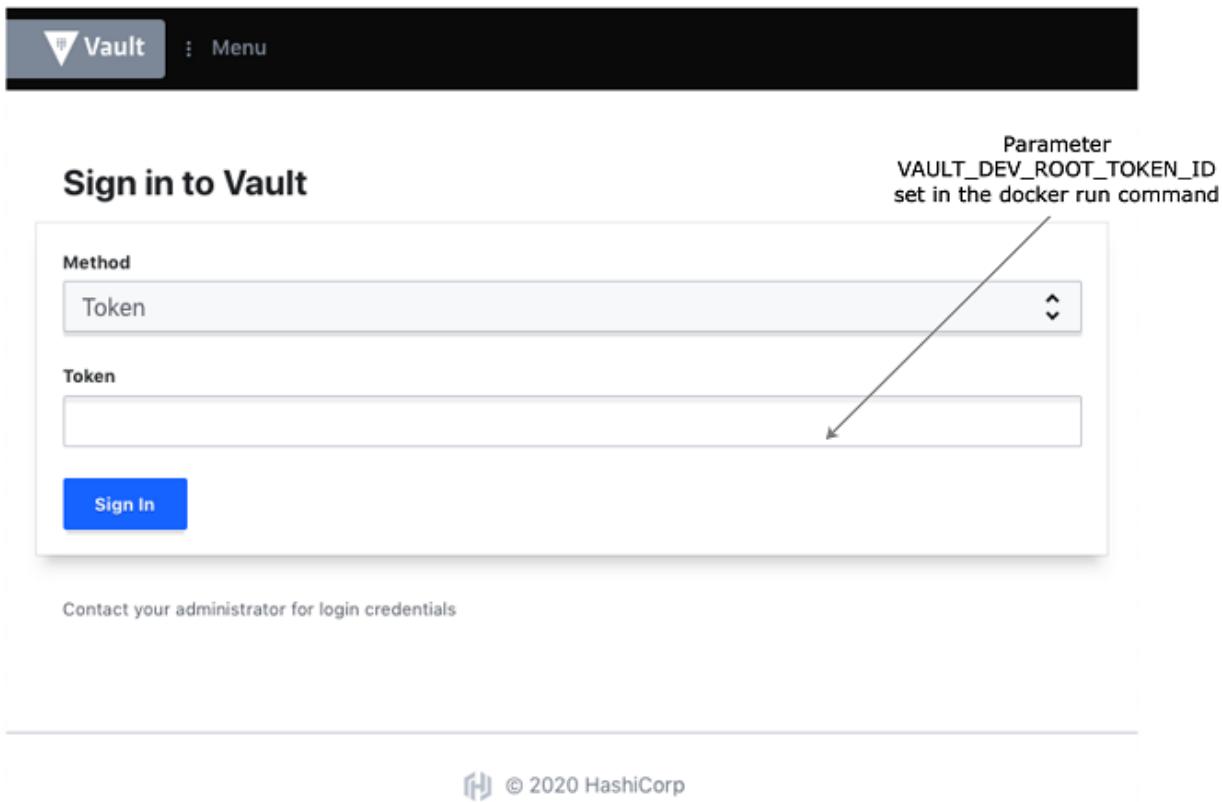
- **VAULT_DEV_ROOT_TOKEN_ID**: This parameter sets the ID of the generated root token. The root token is the initial access token to start configuring Vault. This sets the ID of the initial generated root token to the given value
- **VAULT_DEV_LISTEN_ADDRESS**: This parameter sets the IP address and port of the development server listener the default value is 0.0.0.0:8200.

NOTE In this example I will run Vault locally, if you need additional info on how to run Vault in server mode I highly recommend you to visit the official Vault dock image info (https://hub.docker.com/_/vault)

Once the latest Vault image is pulled into your Docker, we can start creating our secrets. To make this example more straightforward I will be using the Vault UI, but if you prefer to move on with the CLI commands go for it.

5.3.8 Vault UI

Vault also offers us a unified interface that facilitates the process of creating secrets, to access this UI we must enter the following URL `http://0.0.0.0:8200/ui/vault/auth`. This URL was defined by the `VAULT_DEV_LISTEN_ADDRESS` parameter set in the docker run command. Figure 5.10 shows the login page for the vault UI.



**Figure 5.10 Login page in the Vault UI. URL:
<http://0.0.0.0:8200/ui/vault/auth>**

The next step is to create a secret. To create the secret once, you've logged in you should click the secret/ engine in the Vault UI dashboard. For this example, I will be creating a secret called secret/licensingservice with a property called license.vault.property that will have Welcome to vault as value. Remember, this piece of information will have a restricted access control and will be crypted. To achieve this first we need to create a new secret engine and then add the specific secret to that engine. Figure 5.11 shows how to create a new secret engine in Vault UI.

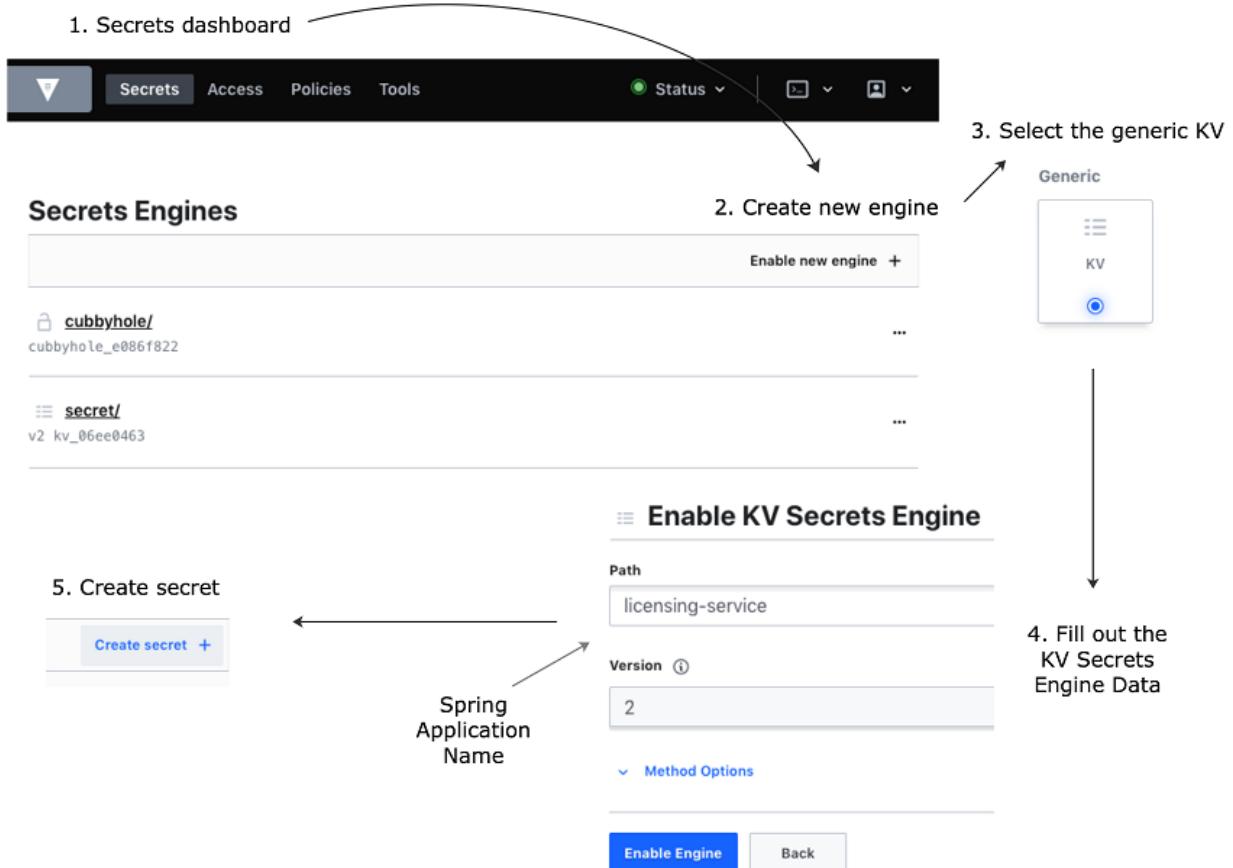


Figure 5.11 Create a new secret engine process in Vault UI

Now, that we have the new secret engine let's create our secret. Figure 5.12 will show you how.

⟨ licensing-service

Spring Profile

Create secret

JSON

Path for this secret

default

Secret metadata

Maximum Number of Versions

10

Require Check and Set ⓘ

Version data

license.vault.property	Welcome to Vault!	⊕	Add
------------------------	-------------------	---	-----

Save **Cancel**

Figure 5.12 Create a new secret in Vault UI

Now that we have configured the Vault and a secret, let's configure our Spring Configuration server to talk with Vault.

To do that let's add the Vault profile to our bootstrap.yml file in the config server. Code listing 5.14 shows how your bootstrap file should look like.

Listing 5.14 Adding Vault to the Spring Cloud bootstrap.yml

```
spring:  
application:  
name: config-server  
profiles:  
active:
```

```
- vault
  cloud:
    config:
      server:
        vault: #A
        port: 8200 #B
        host: 127.0.0.1 #C
        kvVersion: 2 #D
      server:
        port: 8071
```

#A Tells Spring Cloud Config to use Vault as a backend repository

#B Tells Spring Cloud Config the Vault port

#C Tells Spring Cloud Config the Vault host

#D Sets the KV Secrets Engine version.

An important point here is the KV Secrets Engine version, the default `spring.cloud.config.server.kv-version` is 1. It is recommended to use a version 2 when we use Vault 0.10.0 or later.

Now that have everything set, let's test our Spring Cloud config server via an HTTP request. Here you can use a CURL command or some REST client such as POSTMAN.

```
$ curl -X "GET" "http://localhost:8071/licensing-service/default" -H "X-Config-Token: myroot"
```

If everything was configured successfully the command should return a response like the following:

```
{
  "name": "licensing-service",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": null,
  "state": null,
  "propertySources": [
    {
```

```
"name": "vault:licensing-service",
"source": {
  "license.vault.property": "Welcome to vault"
}
}
]
}
```

5.4 Protecting sensitive configuration information

By default, Spring Cloud configuration server stores all properties in plain text within the application's configuration files. This includes sensitive information such as database credentials.

It's an extremely poor practice to keep sensitive credentials stored as plain text in your source code repository. Unfortunately, it happens far more often than you think. Spring Cloud Config does give you the ability to encrypt your sensitive properties easily. Spring Cloud Config supports using both symmetric (shared secret) and asymmetric encryption (public/private key). The asymmetric encryption is more secure than symmetric encryption because it uses modern and more complex algorithms. But sometimes it is more convenient to use the symmetric key since we only have to define a single property value in the bootstrap.yml file in the Spring Config Server.

5.4.1 Setting up a symmetric encryption key

The symmetric encryption key is nothing more than a shared secret that's used by the encrypter to encrypt a value and the decrypter to decrypt a value. With the Spring Cloud configuration server, the symmetric encryption key is a string of characters you select that can be either set in the bootstrap.yml file of your Spring Cloud configuration server or passed to the service via an operating system environment variable called ENCRYPT_KEY here you can select the option that best suits your situation.

NOTE Your symmetric key should be 12 or more characters long and ideally be a random set of characters.

Now said that let's start by doing an example of how to configure the symmetric key on the bootstrap file the Spring Cloud configuration server. The following code listing 5.15 will show you how.

Listing 5.15 Setting the symmetric key in the bootstrap.yml file

```
cloud:  
config:  
server:  
native:  
search-locations: classpath:/config  
git:  
uri: https://github.com/ihuaylupo/config.git  
searchPaths: licensingservice  
server:  
port: 8071  
encrypt:  
key: secretkey #A
```

#A Tells the Spring Cloud config server to use that value as the symmetric key.

For the purposes of this book I will always set the ENCRYPT_KEY environment variable to be

```
export ENCRYPT_KEY=IMSYMMETRIC
```

But feel free to use the bootstrap.yml file property in case you need to make local testings without using docker.

Managing encryption keys

For the purposes of this book, I did two things that I wouldn't normally recommend in a production deployment:

I set the encryption key to be a phrase. I wanted to keep the key simple so that I could remember it and it would fit nicely in reading the text. In a real-world deployment, I'd use a separate encryption key for each environment I was deploying to and I'd use random characters as my key.

I've hardcoded the ENCRYPT_KEY environment variable directly in the Docker files used within the book. I did this so that you as the reader could download the files and start them up without having to remember to set an environment variable. In a real runtime environment, I would reference the ENCRYPT_KEY as an operating system environment variable inside my Dockerfile. Be aware of this and don't hardcode your encryption key inside your Dockerfiles. Remember, your Dockerfiles are supposed to be kept under source control.

5.4.2 Encrypting and decrypting a property

We are now ready to begin encrypting properties for use in Spring Cloud Config. We will encrypt the licensing services Postgres database password you've been using to access Ostock data. This property, called `spring.datasource.password`, is currently set as plain text to be the value `postgres`.

When you fire up your Spring Cloud Config instance, Spring Cloud Config detects that the ENCRYPT_KEY environment variable or the bootstrap file property is set and automatically adds two new endpoints (/encrypt and /decrypt) to the Spring Cloud Config service. We will use the /encrypt endpoint to encrypt the postgres value.

Figure 5.13 shows how to encrypt the postgres value using the /encrypt endpoint and POSTMAN. Please note that whenever you call the /encrypt or /decrypt endpoints, you need to make sure you do a POST to these endpoints.

The screenshot shows the POSTMAN interface for interacting with a Spring Cloud Config service. At the top, a 'New encrypt endpoint' is indicated above the URL input field. The URL is set to `http://localhost:8071/encrypt`. Below the URL, the 'Body' tab is selected, showing a raw JSON payload with the value `1 postgres`. A label 'The value we want to encrypt' points to this payload. The 'Headers' tab shows 10 headers. At the bottom, the response status is 200 OK, time is 12ms, and size is 228 B. The response body contains the encrypted value `559ac661a1c93d52b9e093d3833a238a142de7772961d94751883b17c41746a6`. A label 'The encrypted result' points to this value.

Figure 5.13 Encrypt the spring data source password using the /encrypt endpoint.

If we wanted to decrypt the value, we need to use the /decrypt endpoint passing in the encrypted string in the call.

We can now add the encrypted property to your GitHub or filesystem-based configuration file for the licensing service using the following syntax:

```
spring.datasource.password =  
{cipher}559ac661a1c93d52b9e093d3833a238a142de7772961d94751883b17c41746a6
```

The following code listing 5.16 will show you how.

Listing 5.16 Add the encrypted value to the license service properties file

```
spring.datasource.url = jdbc:postgresql://localhost:5432/ostock_dev  
spring.datasource.username = postgres  
spring.datasource.password = {cipher}  
559ac661a1c93d52b9e093d3833a238a142de7772961d94751883b17c41746a6
```

Spring Cloud configuration server requires all encrypted properties to be prepended with a value of {cipher}. The {cipher} value tells Spring Cloud configuration server it's dealing with an encrypted value.

5.5 Closing thoughts

Application configuration management might seem like a mundane topic, but it's of critical importance in a cloud-

based environment. As we'll discuss in more detail in later chapters, it's critical that your applications and the servers they run on be immutable and that the entire server being promoted is never manually configured between environments. This flies in the face of traditional deployment models where you deploy an application artifact (for example, a JAR or WAR file) along with its property files to a "fixed" environment.

With a cloud-based model, the application configuration data should be segregated completely from the application, with the appropriate configuration data needs injected at runtime so that the same server/application artifact are consistently promoted through all environments.

5.6 Summary

- Spring Cloud configuration server allows you to set up application properties with environment specific values.
- Spring uses Spring profiles to launch a service to determine what environment properties are to be retrieved from the Spring Cloud Config service.
- Spring Cloud configuration service can use a file-based, Git-based or a Vault-based application configuration repository to store application properties.
- Spring Cloud configuration service allows you to encrypt sensitive property files using symmetric and asymmetric encryption.

6 On Service Discovery

This chapter covers

- Explaining why service discovery is important to any cloud-based application development
- Understanding the pros and cons of service discovery vs. the more traditional load-balancer approach
- Setting up a Spring Netflix Eureka Server
- Registering a Spring Boot-based microservice with Eureka
- Using Spring Cloud Load Balancer library to use client-side load balancing

In any distributed architecture, we need to find the hostname or IP address of where a machine is located. This concept has been around since the beginning of distributed computing and is known formally as service discovery. Service discovery can be something as simple as maintaining a property file with the addresses of all the remote services used by an application, or something as formalized as a Universal Description, Discovery, and Integration (UDDI) repository.

Service discovery is critical to microservice, cloud-based applications for two key reasons. First, it offers the application team the ability to quickly horizontally scale the number of service instances running in an environment. The service consumers are abstracted away from the physical location of the service via service discovery. Because the

service consumers don't know the physical location of the actual service instances, new service instances can be added or removed from the pool of available services.

This ability to quickly scale services without disrupting the service consumers is a compelling concept. It can move a development team used to building monolithic, single-tenant (for example, one customer) applications away from thinking about scaling only in terms of adding bigger, better hardware (vertical scaling) to the more robust approach to scaling by adding more servers with more services (horizontal scaling).

A monolithic approach usually drives development teams down the path of over-buying their capacity needs. Capacity increases come in clumps and spikes and are rarely a smooth, steady path. For example, consider the increment of requests made to e-commerce sites before some holidays. Microservices allow us to scale new service instances. Service discovery helps abstract that these deployments are occurring away from the service consumer.

The second benefit of service discovery is that it helps increase application resiliency. When a microservice instance becomes unhealthy or unavailable, most service discovery engines will remove that instance from its internal list of available services. The damage caused by a down service will be minimized because the service discovery engine will route services around the unavailable service.

All of this may sound somehow complicated, and you might be wondering why we can't use tried-and-true methods such as DNS (Domain Name Service) or a load balancer to help facilitate service discovery. Let's walk through why that won't

work with a microservices-based application, particularly one that's running in the cloud. Next, we'll learn how to implement Eureka Service Discovery in our architecture.

6.1 Where's my service?

Whenever you have an application calling resources spread across multiple servers, it needs to locate the physical location of those resources. In the non-cloud world, this service location resolution was often solved through a combination of DNS and a network load balancer. The following figure 6.1 illustrates this model.

An application needs to invoke a service located in another part of the organization. It attempts to invoke the service by using a generic DNS name along with a path that uniquely represents the service that the application was trying to invoke. The DNS name would resolve to a commercial load balancer, such as the popular F5 load balancer (<http://f5.com>) or an open-source load balancer such as HAProxy ([http:// haproxy.org](http://haproxy.org)).

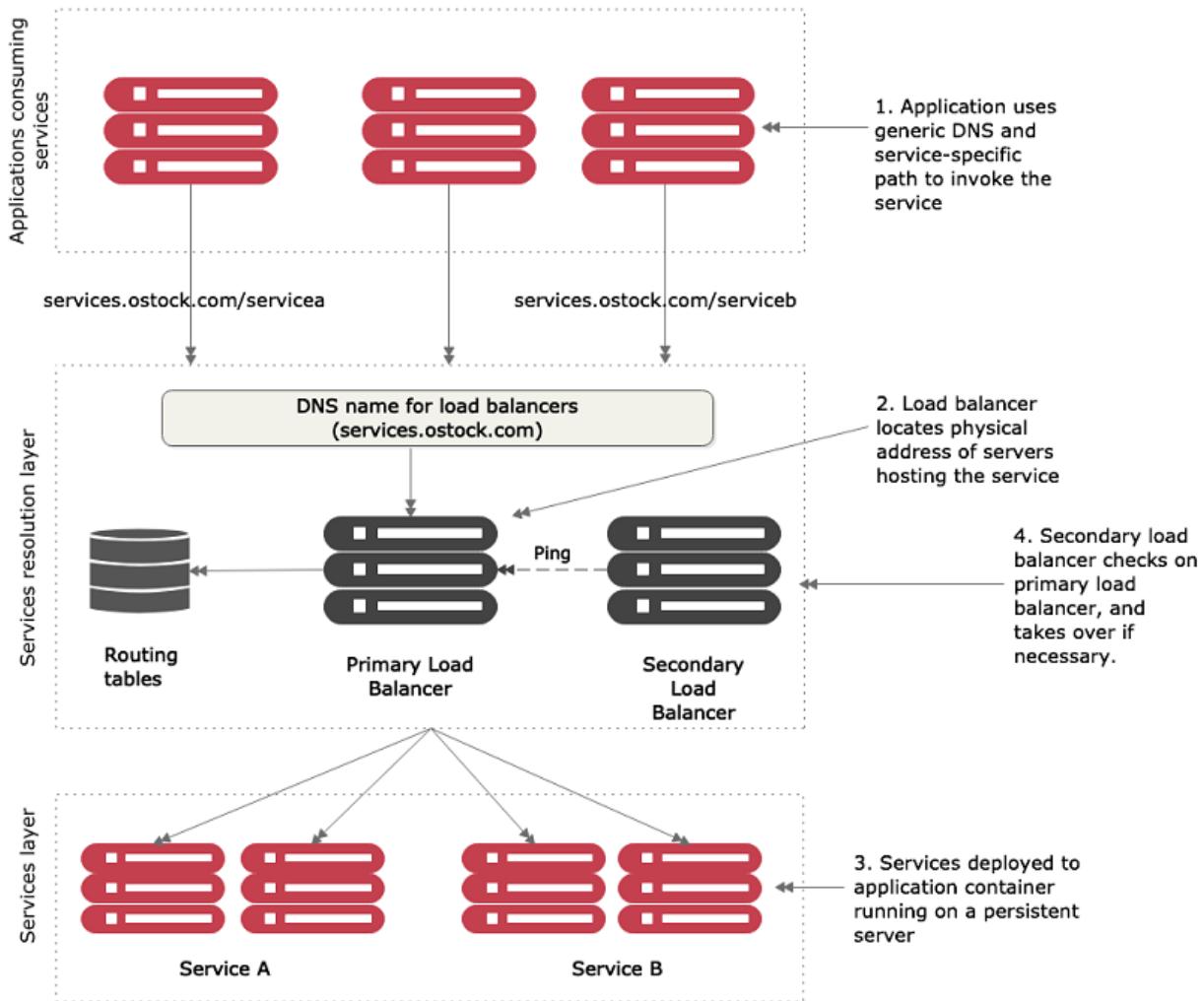


Figure 6.1 A traditional service location resolution model using DNS and a load balancer

The load balancer, upon receiving the request from the service consumer, locates the physical address entry in a routing table based on the path the user was trying to access. This routing table entry contains a list of one or more servers hosting the service. The load balancer then picks one of the servers in the list and forwards the request onto that server.

With this legacy model, each instance of a service used to be deployed in one or more application servers. The number of these application servers was often static (for example, the number of application servers hosting a service didn't go up and down) and persistent (for example, if a server running an application server crashed, it would be restored to the same state it was at the time of the crash, and would have the same IP and configuration that it had previously.)

To achieve a form of high availability, a secondary load balancer is sitting idle and pinging the primary load balancer to see if it's alive. If it isn't alive, the secondary load balancer becomes active, taking over the IP address of the primary load balancer and beginning serving requests.

While this type of model works well with applications running inside of the four walls of a corporate data center and with a relatively small number of services running on a group of static servers, it doesn't work well for cloud-based microservice applications. Reasons for this include:

- **Single point of failure.** While the load balancer can be made highly available, it's a single point of failure for your entire infrastructure. If the load balancer goes down, every application relying on it goes down too. While you can make a load balancer highly available, load balancers tend to be centralized chokepoints within your application infrastructure.
- **Limited horizontal scalability.** By centralizing your services into a single cluster of load balancers, you have limited ability to scale your load-balancing infrastructure across multiple servers horizontally. Many commercial load balancers are constrained by two things: their redundancy model and licensing costs. Most commercial load balancers use a hot-swap model for redundancy, so

you only have a single server to handle the load, while the secondary load balancer is there only for fail-over in the case of an outage of the primary load balancer. You are, in essence, constrained by your hardware. Second, commercial load balancers also have restrictive licensing models geared toward a fixed capacity rather than a more variable model.

- **Statically managed.** Most traditional load balancers aren't designed for fast registration and de-registration of services. They use a centralized database to store the routes for rules, and the only way to add new routes is often through the vendor's proprietary API (Application Programming Interface).
- **Complex.** Because a load balancer acts as a proxy to the services, service consumer requests have to have their requests mapped to the physical services. This translation layer often added a layer of complexity to your service infrastructure because the mapping rules for the service have to be defined and deployed by hand. In a traditional load balancer scenario, this registration of new service instances was done by hand and not at startup time of a new service instance.

These four reasons aren't a general indictment of load balancers. They work well in a corporate environment where the size and scale of most applications can be handled through a centralized network infrastructure. In addition, load balancers still have a role to play in terms of centralizing SSL termination and managing service port security. A load balancer can lock down inbound (ingress) and outbound (egress) port access to all the servers sitting behind it. This concept of least network access is often a critical component when trying to meet industry-standard certification requirements such as PCI (Payment Card Industry) compliance.

However, in the cloud where you have to deal with massive amounts of transactions and redundancy, a centralized piece of network infrastructure doesn't ultimately work as well because it doesn't scale effectively and isn't cost-efficient. Let's now look at how you can implement a robust-service discovery mechanism for cloud-based applications.

6.2 On service discovery in the cloud

The solution for a cloud-based microservice environment is to use a service discovery mechanism that's

- **Highly available.** Service discovery needs to be able to support a “hot” clustering environment where service lookups can be shared across multiple nodes in a service discovery cluster. If a node becomes unavailable, other nodes in the cluster should be able to take over. Remember, a cluster can be defined as a group of multiple server instances. All instances of this environment have an identical configuration and work together to provide high availability, reliability, and scalability. A cluster combined with a load balancer can offer failover to prevent service interruptions and session replication to store session data.
- **Peer-to-peer.** Each node in the service discovery cluster shares the state of a service instance.
- **Load balanced.** Service discovery needs to dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it. In many ways, service discovery replaces the more static, manually managed load balancers used in many early web application implementations.

- **Resilient.** The service discovery's client should “cache” service information locally. Local caching allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the information maintained in its local cache.
- **Fault-tolerant.** Service discovery needs to detect when a service instance isn't healthy and remove the instance from the list of available services that can take client requests. It should detect these faults with services and take action without human intervention.

In the following sections I'm going to

- Walk you through the conceptual architecture of how a cloud-based service discovery agent works.
- Show you how client-side caching and load-balancing allows service to continue to function even when the service discovery agent is unavailable.
- Show you how to implement service discovery using Spring Cloud and Netflix's Eureka service discovery agent.

6.2.1 The architecture of service discovery

To begin our discussion around service discovery architecture, we need to understand four concepts. These general concepts are shared across all service discovery implementations:

- **Service registration.** How does a service register with the service discovery agent?

- **Client lookup of service address.** How a service client looks up service information?
- **Information sharing.** How is service information shared across nodes?
- **Health monitoring.** How do services communicate their health back to the service discovery agent?

The principal objective of service discovery is to have an architecture where our own services are the one that indicate where they are physically located instead of having to manually configure where they are. The following figure 6.2 will show you how service instances are added/removed, and how they update the service discovery agent and become available to process user requests.

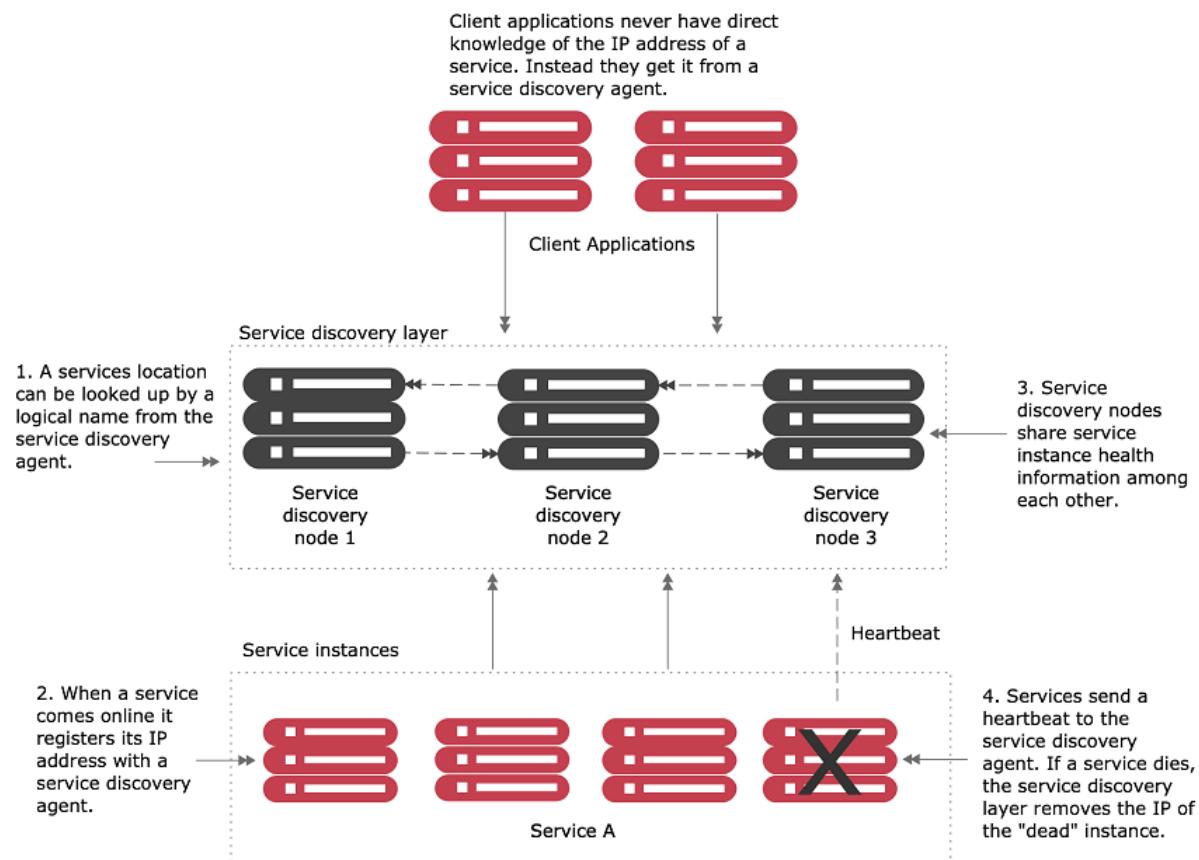


Figure 6.2 As service instances are added/removed, they will update the service discovery nodes and become available to process user requests.

Figure 6.2 shows the flow of the previous four bullets (Service registration, service discovery lookup of a service address, information sharing, and health monitoring) and what typically occurs in a service discovery pattern implementation.

In figure 6.2, one or more service discovery nodes have been started. These service discovery instances are usually don't have a load balancer that sits in front of them.

As service instances start up, they'll register their physical location, path, and port that they can be accessed by one or more service discovery instances. While each instance of a service will have a unique IP address and port, each service instance that comes up will register under the same service ID. A service ID is nothing more than a key that uniquely identifies a group of the same service instances.

A service will usually only register with one service discovery service instance. Most service discovery implementations use a peer-to-peer model of data propagation where the data around each service instance is communicated to all the other nodes in the cluster.

Depending on the service discovery implementation, the propagation mechanism might use a hard-coded list of services to propagate to or use a multi-casting protocol like the gossip or infection-style protocol to allow other nodes to "discover" changes in the cluster.

NOTE If you are interested in knowing more about the gossip or the infection-style protocols, I highly recommend you review the following documentation: Consul Gossip protocol (<https://www.consul.io/docs/internals/gossip.html>) or Brian Storti Swim: The scalable membership protocol (<https://www.brianstorti.com/swim/>).

Finally, each service instance will push to or have pulled from its status by the service discovery service. Any services failing to return a good health check will be removed from the pool of available service instances.

Once a service has registered with a service discovery service, it's ready to be used by an application or service that needs to use its capabilities. Different models exist for a client to "discover" a service. A client can rely solely on the service discovery engine to resolve service locations each time a service is called. With this approach, the service discovery engine will be invoked every time a call to a registered microservice instance is made. Unfortunately, this approach is brittle because the service client is completely dependent on the service discovery engine to be running to find and invoke a service.

A more robust approach is to use what's called client-side load balancing. The client-side load balancing uses an algorithm like zone specific or round-robin, to invoke the instances of the calling services. Remember, when we say a round-robin algorithm load balancing, we refer to a way of distributing client requests across several servers. It consists of forwarding a client request to each of the servers in turn. An advantage of using the client-side load balancer with Eureka is that when a service instance goes down, it is removed from the registry—once that is done, the client-side load balancer updates itself without any manual

interventions by establishing a constant communication with the registry service. Figure 6.3 illustrates this approach.

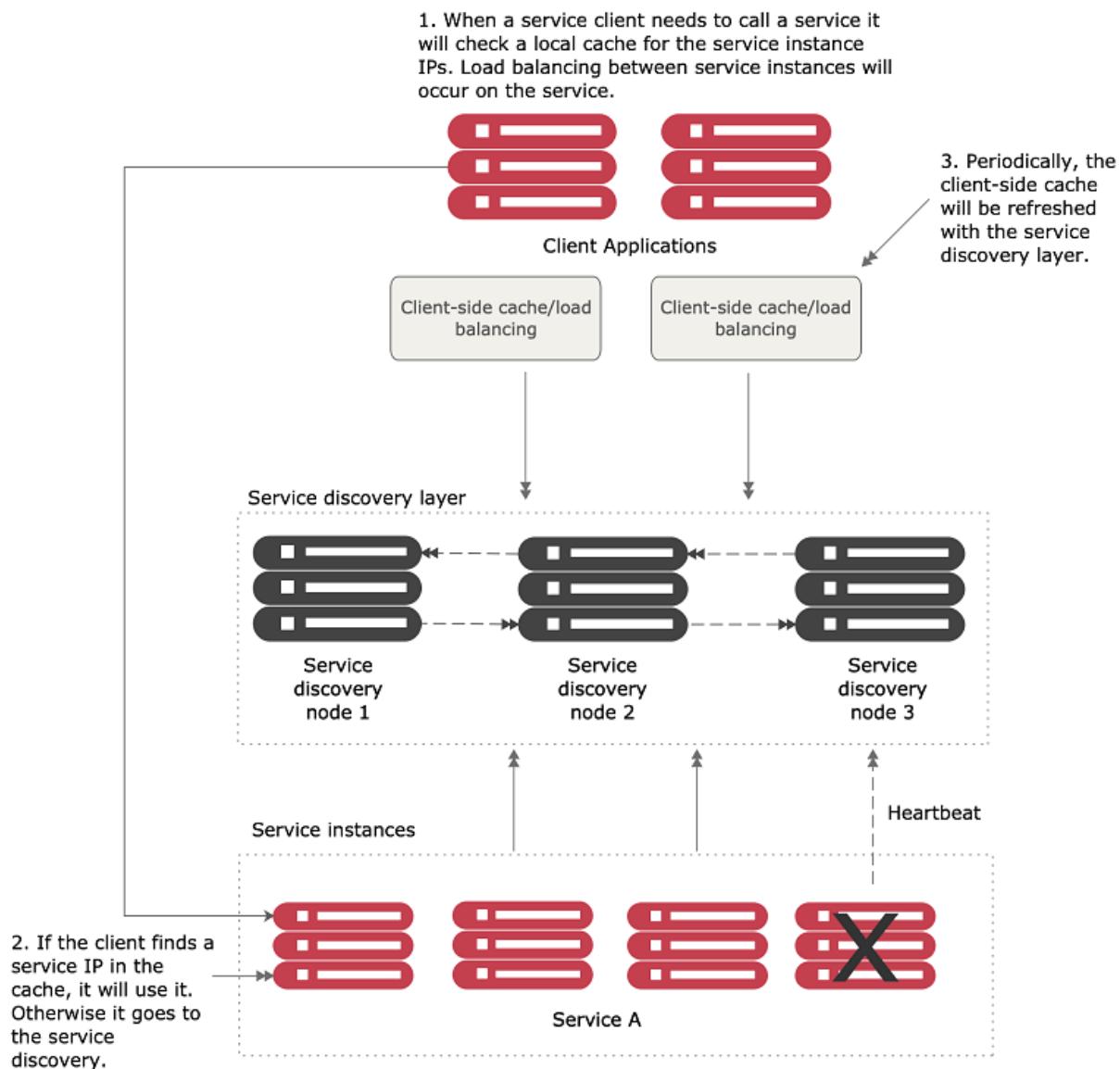


Figure 6.3 Client-side load balancing caches the location of the services so that the service client doesn't have to contact service discovery on every call.

In this model, when a consuming client needs to invoke a service

1. It will contact the discovery service for all the instances a service consumer (client) is asking for and then cache data locally on the service consumer's machine.
2. Each time a client wants to call the service, the service consumer will look up the location information for the service from the cache. Usually, client-side caching will use a simple load balancing algorithm like the "round-robin" load balancing algorithm to ensure that service calls are spread across multiple service instances.
3. The client will then periodically contact the discovery service and refresh its cache of service instances. The client cache is eventually consistent, but there's always a risk that between when the client contacts the service discovery instance for a refresh and calls are made, calls might be directed to a service instance that isn't healthy.

If, during the course of calling a service, the service call fails, the local service discovery cache is invalidated, and the service discovery client will attempt to refresh its entries from the service discovery agent.

Let's now take the generic service discovery pattern and apply it to your Ostock problem domain.

6.2.2 Service discovery in action using Spring and Netflix Eureka

Now we are going to implement service discovery by setting up a service discovery agent and then registering two services with the agent. With this implementation, we'll use the information retrieved by the service discovery to call a service from another service. Spring Cloud offers multiple methods for looking up information from a service discovery agent. We'll also walk through the strengths and weaknesses of each approach.

Once again, the Spring Cloud project makes this type of setup trivial to undertake. We'll use Spring Cloud and Netflix's Eureka service discovery engine to implement your service discovery pattern. For the client-side load balancing, we'll use Spring Cloud Load Balancer library.

It is essential to highlight that in this chapter, we are not going to use Ribbon. Ribbon was the de facto client-side load balancer for REST-based communications between applications using Spring Cloud. Netflix Ribbon client-side load balancing was a stable solution but now has entered a maintenance mode, so, unfortunately, it will not be developed anymore.

In this chapter, I will explain how to use the Spring Cloud Load Balancer, which is a replacement for Ribbon. Currently, the Spring Cloud Load Balancer is still under active development to expect some new functionalities soon.

In the previous two chapters, we kept our licensing service simple and included the organization name for the licenses with the license data. In this chapter, we'll break the organization information into its own service.

Figure 6.4 shows the implementation of the client-side caching with Eureka for our Ostock microservices.

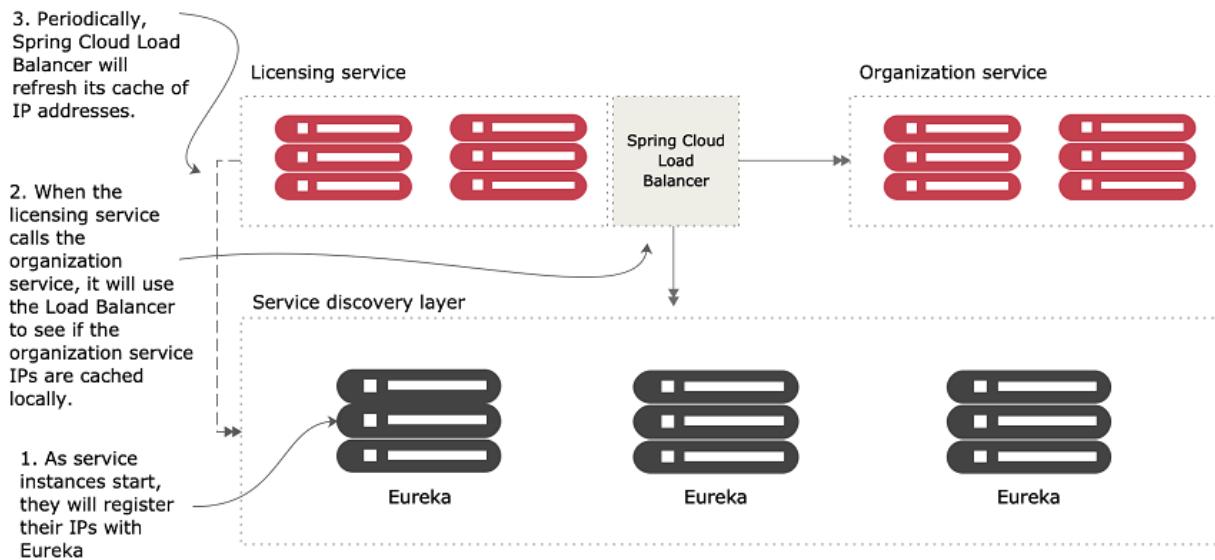


Figure 6.4 By implementing client-side caching and Eureka with the licensing and organization services, you can lessen the load on the Eureka servers and improve client stability if Eureka becomes unavailable.

When the licensing service is invoked, it will call the organization service to retrieve the organization information associated with the designated organization ID. The actual resolution of the organization service's location will be held in a service discovery registry. For this example, we'll register two instances of the organization service with a service discovery registry and then use client-side load balancing to look up and cache the registry in each service instance. Figure 6.4 shows this arrangement:

1. As the services are bootstrapping, the licensing and organization services will also register themselves with the Eureka Service. This registration process will tell Eureka the physical location and port number of each service instance along with a service ID for the service being started.
2. When the licensing service calls to the organization service, it will use the Spring Cloud Load Balancer library to provide client-side load balancing. The load balancer will contact the Eureka service to retrieve service location information and then cache it locally.
3. Periodically, the Spring Cloud Load Balancer library will ping the Eureka service and refresh its local cache of service locations.

Any new organization services instance will now be visible to the licensing service locally, while any non-healthy instances will be removed from the local cache.

Next, we'll implement this design by setting up our Spring Cloud Eureka service.

6.3 Building our Spring Eureka Service

In this section, we'll set up our Eureka service using Spring Boot. Like the Spring Cloud configuration service, setting up a Spring Cloud Eureka Service starts with building a new Spring Boot project and applying annotations and configurations. Let's begin by creating that new project with

the Spring Initializr (<https://start.spring.io/>), to achieve this let's follow the next steps:

1. Select maven as the project type.
2. Select Java as the language
3. Select the 2.2.x latest or more stable spring version.
4. Write com.optimagrowth as group and eurekaserver as artifact.
5. Expand the options list and write Eureka Server as name, Eureka server as description and com.optimagrowth.eureka as package name.
6. Select the JAR packaging.
7. Select Java 11 as the java version.
8. Add the Eureka Server, Config Client and Spring Boot Actuator dependencies as shown in figure 6.5

Selected dependencies

The screenshot shows the 'Selected dependencies' section of the Spring Initializr interface. It lists three dependencies, each with a checked checkbox icon to its right:

- Spring Boot Actuator**: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- Eureka Server**: spring-cloud-netflix Eureka Server.
- Config Client**: Client that connects to a Spring Cloud Config Server to fetch the application's configuration.

Figure 6.5 Eureka server dependencies in Spring Initializr

The following code listing 6.1 shows how the Eureka server pom.xml file should look like.

Listing 6.1 Maven pom file for the Eureka Server

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.5.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.optimagrowth</groupId>
  <artifactId>eurekaserver</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Eureka Server</name>
  <description>Eureka Server</description>
  <properties>
    <java.version>11</java.version>
    <spring-cloud.version>Hoxton.SR1</spring-cloud.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId> #A
      <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId> #B
    </dependency>
    <exclusions> #C
      <exclusion>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-ribbon</artifactId>
      </exclusion>
      <exclusion>
        <groupId>com.netflix.ribbon</groupId>
        <artifactId>ribbon-eureka</artifactId>
      </exclusion>
    </exclusions>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-loadbalancer</artifactId> #D
    </dependency>
  </dependencies>

```

```

</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
Rest of pom.xml removed for conciseness
...
</project>

```

#A Tells your maven build to include the client that connects to a Spring Cloud Configuration server to retrieve the application's configuration.

#B Tells your maven build to include the Eureka libraries

#C Exclude the Netflix Ribbon libraries.

#D Tells your maven build to include the Spring Cloud Load Balancer libraries.

The next step is to set up the src/main/resources/bootstrap.yml file with the configuration needed to retrieve the configuration from the Spring Config Server that we previously created on chapter 5 and with the configuration to disable Ribbon as our default client-side load balancer. Code listing 6.2 will show your bootstrap.yml file should look like.

Listing 6.2 Setting up the Eureka bootstrap.yml file

```

spring:
application:
name: eureka-server #A
cloud:
config:
uri: http://localhost:8071 #B
loadbalancer:
ribbon:
enabled: false #C

```

```
#A Specify the name of the Eureka service so that Spring Cloud Config client  
knows which service is being looked up.  
#B Specify the location of the Spring Cloud Config server.  
#C Because Ribbon is still used as the default client-side load balancer we need  
to disable it using the loadbalancer.ribbon.enabled configuration.
```

Once we added the Spring Configuration Server information in the bootstrap file on the Eureka Server and we disabled Ribbon as our load balancer, we can continue with the next step that is adding the configuration needed to set up the Eureka service running in standalone mode (for example, no other nodes in the cluster) in the Spring Configuration Server. In order to achieve this, we must create the eureka server configuration file in the repository we have set in the Spring Configuration Service. Remember, the repositories could be classpath, file system, GIT, or vault, and the configuration file should be named as the `spring.application.name` property previously defined in the Eureka bootstrap.yml file of the Eureka service. For purposes of this example, I will create the `eureka-server.yml` configuration file in the classpath and you can find it in the following path `/configserver/src/main/resources/config/eureka-server.yml`. The following code listing 6.3 will show you the content of this file.

NOTE In case you didn't follow the previous chapters code listings, you can download the code created in chapter 5 from the following link: <https://github.com/ihuaylupo/manning-smia/tree/master/chapter5>.

Listing 6.3 Setting up the Eureka configuration in the Spring Configuration Server

```
server:  
port: 8070 #A  
eureka:
```

```
instance:  
  hostname: localhost #B  
  client:  
    registerWithEureka: false #C  
    fetchRegistry: false #D  
  serviceUrl:  
    defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #E  
  server:  
    waitTimeInMsWhenSyncEmpty: 5 #F
```

#A Port Eureka Server is going to listen on.

#B Eureka instance hostname.

#C Don't register with Eureka service.

#D Don't cache registry information locally.

#E Provides the service URL.

#F Initial time to wait before server takes requests.

The key properties being set are the server.port attribute that sets the default port used and the eureka.instance.hostname that sets the eureka instance hostname for the Eureka service. The eureka.client.registerWithEureka attribute tells the service not to register with a Eureka service when the Spring Boot Eureka application starts because this is the Eureka service. The eureka.client.fetchRegistry attribute is set to false so that when the Eureka service starts, it doesn't try to cache its registry information locally. When running a Eureka client, you'll want to change this value for the Spring Boot services that are going to register with Eureka. The eureka.client.serviceUrl.defaultZone provides the service URL for any client; it is a combination of the eureka.instance.hostname and the server.port.

You'll notice that the last attribute, eureka.server.waitTimeInMsWhenSyncEmpty, indicates the milliseconds to wait before starting. When you're testing your service locally, you should use this line because Eureka

won't immediately advertise any services that register with it. It will wait five minutes by default to give all of the services a chance to register with it before advertising them. Using this line for local testing will help speed up the amount of time it will take for the Eureka service to start and show services registered with it.

Individual services registering will take up to 30 seconds to show up in the Eureka service because Eureka requires three consecutive heartbeat pings from the service spaced 10 seconds apart before it will say the service is ready for use. Keep this in mind as you're deploying and testing your own services.

The last piece of setup work we are going to do in setting up your Eureka service is adding an annotation to the application bootstrap class we are using to start your Eureka service. For the Eureka service, the application bootstrap class can be found in the `src/main/java/com/optimagrowth/eureka/EurekaServerApplication.java` class. The following code listing 6.4 shows where to add the annotations.

Listing 6.4 Annotating the bootstrap class to enable the Eureka server

```
package com.optimagrowth.eureka;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
@SpringBootApplication
@EnableEurekaServer #A
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

```
#A Enable Eureka server in the Spring service
```

At this point, we are only going to use a new annotation to tell our service to be a Eureka service; that's `@EnableEurekaServer`. Now, we can start up the Eureka service by running the `mvn spring-boot:run` or run `docker-compose` to start the service. Once this command is executed, we should have a running Eureka service with no services registered in it.

NOTE Remember, first; we will need to run the Spring Configuration service because that's the one that contains the Eureka application configuration. If you don't run first your configuration service, you will get the following error: Connect Timeout Exception on Url - `http://localhost:8071`. Will be trying the next url if available.
`com.sun.jersey.api.client.ClientHandlerException:`
`java.net.ConnectException: Connection refused (Connection refused)`. To avoid the previous issue try running the services with docker compose. Remember, you can find the `docker-compose.yml` file updated in the chapter repository on GitHub.

Next, we'll build out the organization service, and we will register the licensing and the organization services with our Eureka service.

6.4 Registering services with Spring Eureka

At this point, we have a Spring-based Eureka server up and running. In this section, we'll configure the organization and licensing services to register themselves with your Eureka server. This work is done in preparation for having a service client look up a service from our Eureka registry. By the time we're done with this section, you should have a firm

understanding of how to register a Spring Boot microservice with Eureka.

Registering a Spring Boot-based microservice with Eureka is a straightforward exercise. For the purposes of this chapter, we're not going to walk through all of the Java code involved with writing the service (we purposely kept that amount of code small), but instead, focus on registering the service with the Eureka service registry you created in the previous section.

NOTE From now on, we are going to use a new service called organization service, that will contain CRUD endpoints. You can download the code of the licensing and organization service from the following link (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter6/Initial>). Also, at this point, you can use other microservices you might have if that's the scenario. Just pay attention to the service id names while you are registering them into the service discovery.

The first thing we need to do is add the Spring Eureka dependency to our organization and license service's pom.xml file. The following code listing 6.5 shows the dependency that we need to add.

Listing 6.5 Adding Spring Eureka dependency in the organization's service pom.xml

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId> #A
</dependency>
```

#A Includes the Eureka libraries so that the service can register with Eureka

The spring-cloud-starter-netflix-eureka-client artifact holds the jar files that Spring Cloud will use to interact with your

Eureka service.

After we've set up the pom.xml file, we need to make sure we have set the spring.application.name in the bootstrap file of the service we want to register. In this case, we need to add the spring.application.name to the bootstrap.yml file of the organization and licensing services, the following code listings 6.6 and 6.7 will indicate you how.

Listing 6.6 Adding the spring.application.name in the organization service's bootstrap.yml file

```
spring:  
application:  
name: organization-service #A  
profiles:  
active: dev  
cloud:  
config:  
uri: http://localhost:8071
```

#A Logical name of the service that will be registered with Eureka

Listing 6.7 Adding the spring.application.name in the licensing service's bootstrap.yml file

```
spring:  
application:  
name: licensing-service  
profiles:  
active: dev  
cloud:  
config:  
uri: http://localhost:8071
```

Every service registered with Eureka will have two components associated with it: the application ID and the

instance ID. The application ID is used to represent a group service instance. In a Spring-Boot-based microservice, the application ID will always be the value set by the `spring.application.name` property. For our organization service, our `spring.application.name` is creatively named `organization-service` and `licensing-service` for our licensing service. The instance ID will be a random auto-generated number meant to represent a single service instance.

Next, we need to tell Spring Boot to register the organization and licensing services with Eureka. This registration is done via additional configuration in the service's configuration files managed by the Spring Configuration Service. For purposes of this example, those files are located in the Spring Configuration server project in the path `src/main/resources/config/organization-service.properties` and `src/main/resources/config/licensing-service.properties`. The following code listing 6.8 shows how to register the service with Eureka.

NOTE Remember, the configuration file could be either a YAML or a properties file. This configuration file could be located in the classpath, filesystem, git repository, or vault. It will depend on the configuration you have on the Spring configuration server. For this example, I selected the classpath and properties file, but feel free to make the changes that best suit your needs.

Listing 6.8 Modifying your organization and licensing services's application.properties to talk to Eureka

```
eureka.instance.preferIpAddress = true #A  
eureka.client.registerWithEureka = true #B  
eureka.client.fetchRegistry = true #C  
eureka.client.serviceUrl.defaultZone = http://localhost:8070/eureka/ #D
```

#A Register the IP of the service rather than the server name.
#B Register the service with Eureka.
#C Pull down a local copy of the registry.
#D Location of the Eureka Service.

In case you have an application.yml file, your file should look like the following code:

```
eureka:  
  instance:  
    preferIpAddress: true  
    client:  
      registerWithEureka: true  
      fetchRegistry: true  
      serviceUrl:  
        defaultZone: http://localhost:8070/eureka/
```

This configuration provides how and where the service should register with the Eureka service. The eureka.instance.preferIpAddress property tells Eureka that you want to register the service's IP address to Eureka rather than its hostname.

Why prefer IP address?

By default, Eureka will try to register the services that contact it by hostname. This works well in a server-based environment where a service is assigned a DNS-backed hostname. However, in a container-based deployment (for example, Docker), containers will be started with randomly generated hostnames and no DNS entries for the containers.

If you don't set the eureka.instance.preferIpAddress to true, your client applications won't correctly resolve the location of the hostnames because there will be no DNS entry for that container. Setting the preferIpAddress attribute will inform the Eureka service that the client wants to be advertised by IP address.

Personally, I always set this attribute to true. Cloud-based microservices are supposed to be ephemeral and stateless. They can be started up and shut down at will. IP addresses are more appropriate for these types of services.

The `eureka.client.registerWithEureka` attribute is the trigger to tell the organization and the licensing services to register itself with Eureka. The `eureka.client.fetchRegistry` attribute is used to tell the Spring Eureka Client to fetch a local copy of the registry. Setting this attribute to true will cache the registry locally instead of calling the Eureka service with every lookup. Every 30 seconds, the client software will re-contact the Eureka service for any changes to the registry.

These two properties are set by default to true, but I've included it in the application configuration's file for illustrative purposes. The code will work without setting those properties to true.

The last attribute, the `eureka.serviceUrl.defaultZone` attribute holds a comma-separated list of Eureka services the client will use to resolve to service locations. For our purposes, you're only going to have one Eureka service.

NOTE All the key, value properties defined previously can also be declared in the bootstrap file of each service. But the idea is to delegate the configuration to the Spring Configuration service. That's why we are registering all this configuration in the service's configuration files in our spring configuration service repository. So far, the bootstrap file of your services should only contain the application name, a profile (if needed), and the spring cloud configuration URI.

Eureka high availability

Setting up multiple URL services isn't enough for high availability. The `eureka.serviceUrl.defaultZone` attribute only provides a list of Eureka services for the client to communicate with. You also need to set up the Eureka services to replicate the contents of their registry with each other.

A group of Eureka registries communicate with each other using a peer-to-peer communication model where each Eureka service has to be configured to know about the other nodes in the cluster. Setting up a Eureka cluster is outside of the scope of this book. If you're interested in setting up a Eureka cluster, please visit the Spring Cloud project's website for further information (<https://projects.spring.io/spring-cloud/spring-cloud.html#spring-cloud-eureka-server>).

At this point, we'll have two services registered with our Eureka service.

We can use Eureka's REST API or the Eureka dashboard to see the contents of the registry.

6.4.1 Eureka's REST API

To see all the instances of a service hit the following GET endpoint:

```
http://<eureka service>:8070/eureka/apps/<APPID>
```

For instance, to see the organization service in the registry you can call <http://localhost:8070/eureka/apps/organization-service>. The response of that call should look the following figure 6.6.

```

    ▼<application>
      <name>ORGANIZATION-SERVICE</name>
      ▼<instance>
        <instanceId>192.168.0.100:organization-service:8081</instanceId>
        <hostName>192.168.0.100</hostName>
        ▶<app>ORGANIZATION-SERVICE</app>
        ▶<ipAddr>192.168.0.100</ipAddr>
        ▶<status>UP</status>
        ▶<overriddenStatus>UNKNOWN</overriddenStatus>
        <port enabled="true">8081</port>
        <securePort enabled="false">443</securePort>
        <countryId>1</countryId>
        ▼<dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
          <name>MyOwn</name>
        </dataCenterInfo>
        ▼<leaseInfo>
          <renewalIntervalInSecs>30</renewalIntervalInSecs>
          <durationInSecs>90</durationInSecs>
          <registrationTimestamp>1583080994584</registrationTimestamp>
          <lastRenewalTimestamp>1583082854994</lastRenewalTimestamp>
          <evictionTimestamp>0</evictionTimestamp>
          <serviceUpTimestamp>1583080994585</serviceUpTimestamp>
        </leaseInfo>
        ▼<metadata>
          <management.port>8081</management.port>
          <jmx.port>62817</jmx.port>
        </metadata>
        <homePageUrl>http://192.168.0.100:8081</homePageUrl>
        <statusPageUrl>http://192.168.0.100:8081/actuator/info</statusPageUrl>
        <healthCheckUrl>http://192.168.0.100:8081/actuator/health</healthCheckUrl>
        <vipAddress>organization-service</vipAddress>
        <secureVipAddress>organization-service</secureVipAddress>
        <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
        <lastUpdatedTimestamp>1583080994585</lastUpdatedTimestamp>
        <lastDirtyTimestamp>1583080994547</lastDirtyTimestamp>
        <actionType>ADDED</actionType>
      </instance>
    </application>

```

Annotations pointing to specific XML elements:

- Lookup key for the service**: Points to the `<name>ORGANIZATION-SERVICE</name>` element.
- IP address of the organization service instance**: Points to the `<ipAddr>192.168.0.100</ipAddr>` element.
- The service is currently up and functioning.**: Points to the `<status>UP</status>` element.

Figure 6.6 Calling the Eureka REST API to see the organization. The response will show the IP address of the service instances registered in Eureka, along with the service status.

The default format returned by the Eureka service is XML. Eureka can also return the data in figure 4.5 as a JSON payload, but you have to set the Accept HTTP header to be application/json. An example of the JSON payload is shown in figure 6.7.

The Accept HTTP header set to application/json will return the service information in JSON.

▶ Eureka Organization REST API

GET	http://localhost:8070/eureka/apps/organization-service					
Params	Authorization	Headers (7)	Body	Pre-request Script	Tests	Settings
▼ Headers (1)						
	KEY	VALUE				
<input checked="" type="checkbox"/>	Accept	application/json				
	Key	Value				
▶ Temporary Headers (6) ⓘ						
Body	Cookies	Headers (5)	Test Results			
Pretty	Raw	Preview	Visualize BETA	JSON	CSV	
<pre> 1 { 2 "application": { 3 "name": "ORGANIZATION-SERVICE", 4 "instance": [5 { 6 "instanceId": "192.168.0.100:organization-service:8081", 7 "hostName": "192.168.0.100", 8 "app": "ORGANIZATION-SERVICE", 9 "ipAddr": "192.168.0.100", 10 "status": "UP", 11 "overriddenStatus": "UNKNOWN", 12 "port": { 13 "\$": 8081, 14 "@enabled": "true" 15 }, 16 "securePort": { 17 "\$": 443, 18 "@enabled": "false" 19 } </pre>						

Figure 6.7 Calling the Eureka REST API with the results being JSON

6.4.2 Eureka dashboard

Once the Eureka service is up, we can point our browser to <http://localhost:8070> to view the Eureka dashboard. With the Eureka Dashboard, we can see the registration status of our

services. An example of the Eureka dashboard is shown in figure 6.8.

The screenshot shows the Spring Eureka dashboard with the following sections:

- System Status**: Displays environment (test), data center (default), current time (2020-03-01T13:28:08 -0600), uptime (00:14), lease expiration enabled (false), renew threshold (5), and renew (last min) (4). A callout bubble says "The service is currently up and running".
- DS Replicas**: Instances currently registered with Eureka:

Application	Ports	Availability Zones	Status
LICENSING-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.100/licensing-service:8080
ORGANIZATION-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.100/organization-service:8081
- General Info**: Metrics table:

Name	Value
total-available-memory	376mb
environment	test
num-of-cpus	12
current-memory-usage	179mb (48%)
server-uptime	00:14
registered-replicas	
unavailable-replicas	
available-replicas	
- Instance Info**: Instance details table:

Name	Value
ipAddr	192.168.0.100
status	UP

A red arrow points from the text "Lookup key for the service" to the "ipAddr" field in the Instance Info table.

Figure 6.8 Eureka dashboard with the organization and the licensing registered instances.

On Eureka and service startups: don't be impatient

When a service registers with Eureka, Eureka will wait for three successive health checks over the course of 30 seconds before the service becomes available via a Eureka. This warm-up period throws developers off because they think that Eureka hasn't registered their services if they try to call their service

immediately after the service has been launched. This is evident in our code examples running in the Docker environment because the Eureka service and the application services (licensing and organization services) all start up at the same time. Be aware that after starting the application, you may receive 404 errors about services not being found, even though the service itself has started. Wait 30 seconds before trying to call your services.

In a production environment, your Eureka services will already be running, and if you're deploying an existing service, the old services will still be in place to take requests.

Now, that we've registered the organization let's see how we can use the service discovery to look up a service.

6.5 Using service discovery to look up a service

In this section, I will explain how we can also have the licensing service call the organization service without having direct knowledge of the location of any of the organization services. The licensing service will look up the physical location of the organization by using Eureka.

For our purposes, we're going to look at three different Spring/Netflix client libraries in which a service consumer can interact with the Spring Cloud Load Balancer. These libraries will move from the lowest level of abstraction for interacting with the load balancer to the highest. The libraries we'll explore include

- Spring Discovery client
- Spring Discovery client enabled RestTemplate

- Netflix Feign client

Let's walk through each of these clients and see their use in the context of the licensing service. Before we start into the specifics of the client, I wrote a few convenience classes and methods in the code so you can play with the different client types using the same service endpoint.

First, I've modified the `src/main/java/com/optimagrowth/license/controller/LicenseController.java` to include a new route for the license services. This new route will allow you to specify the type of client you want to invoke the service with. This is a helper route so that as we explore each of the different methods for invoking the organization's service via the load balancer, you can try each mechanism through a single route. The following code listing 6.9 shows the code for the new route in the `LicenseController` class.

Listing 6.9 Calling the licensing service with different REST clients

```
@RequestMapping(value="/{licenseId}/{clientType}",method = RequestMethod.GET) #A
public License getLicensesWithClient(
@PathVariable("organizationId") String organizationId,
@PathVariable("licenseId") String licenseId,
@PathVariable("clientType") String clientType) {
    return licenseService.getLicense(organizationId,licenseId, clientType);
}
```

#A The `clientType` determines the type of Spring REST client to use.

In this code, the `clientType` parameter passed on the route will drive the type of client we're going to use in the code

examples. The specific types you can pass in on this route include

- **Discovery.** Uses the discovery client and a standard Spring RestTemplate class to invoke the organization service
- **Rest.** Uses an enhanced Spring RestTemplate to invoke the LoadBalance-based service
- **Feign.** Uses Netflix's Feign client library to invoke a service via the load balancer.

NOTE Because I'm using the same code for all three types of clients, you might see situations where you'll see annotations for specific clients even when they don't seem to be needed. For example, you'll see both the @EnableDiscoveryClient and @EnableFeignClients annotations in the code, even when the text is only explaining one of the client types. This is so I can use one code base for my examples. I'll call out these redundancies and code whenever they are encountered. The idea is that you choose the one that best suits your needs.

In the src/main/java/com/optimagrowth/license/service/LicenseService.java class, I've added a simple method called retrieveOrganizationInfo() that will resolve based on the clientType passed into the route the type of client that will be used to lookup an organization service instance. The getLicense() method on the LicenseService class will use retrieveOrganizationInfo() to retrieve the organization data from the Postgres database. The following code listing 6.10 shows the code for the getLicense() service in the LicenseService class.

Listing 6.10 getLicense() function will use multiple methods to perform a REST Call

```
public License getLicense(String licenseId, String organizationId, String clientType){  
    License license = licenseRepository.findByOrganizationIdAndLicenseId(organizationId, licenseId);  
    if (null == license) {  
        throw new IllegalArgumentException(String.format(  
            messages.getMessage("license.search.error.message", null, null),  
            licenseId, organizationId));  
    }  
    Organization organization = retrieveOrganizationInfo(organizationId, clientType);  
    if (null != organization) {  
        license.setOrganizationName(organization.getName());  
        license.setContactName(organization.getContactName());  
        license.setContactEmail(organization.getContactEmail());  
        license.setContactPhone(organization.getContactPhone());  
    }  
    return license.withComment(config.getExampleProperty());  
}
```

You can find each of the clients we built using the Spring DiscoveryClient, the Spring RestTemplate, or the Feign libraries in the src/main/java/com/optimagrowth/license/service/client package of the licensing-service source code.

To call the getLicense() services with the different clients, you must hit the following GET endpoint:

```
http://<licensing service Hostname/IP>:<licensing service Port>/v1/  
organization/<organizationID>/license/<licenseID>/<client type( feign, discovery,  
rest)>
```

6.5.1 Looking up service instances with Spring Discovery Client

The Spring DiscoveryClient offers the lowest level of access to the load balancer and the services registered within it. Using the DiscoveryClient, you can query for all the services

registered with the Spring Cloud load balancer client and their corresponding URLs.

Next, we'll build a simple example of using the DiscoveryClient to retrieve one of the organization service URLs from the load balancer and then call the service using a standard RestTemplate class. To begin using the DiscoveryClient, we first need to annotate the src/main/java/com/optimagrowth/license/LicenseServiceApplication.java class with the @EnableDiscoveryClient annotation, as shown in the next code listing 6.11.

Listing 6.11 Setting up the bootstrap class to use the Eureka Discovery Client

```
package com.optimagrowth.license;
@SpringBootApplication
@RefreshScope
@EnableDiscoveryClient #A
public class LicenseServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(LicenseServiceApplication.class, args);
    }
}
```

#A Activates the Spring DiscoveryClient for use

The @EnableDiscoveryClient annotation is the trigger for Spring Cloud to enable the application to use the DiscoveryClient and the Spring Cloud Load Balancer libraries. Now, let's look at our implementation of the code that calls the organization service via the Spring DiscoveryClient, as shown in the following code listing 6.12. You can find this in src/main/java/com/optimagrowth/license/service/client/OrganizationDiscoveryClient.java

Listing 6.12 Using the DiscoveryClient to look up information

```
@Component
public class OrganizationDiscoveryClient {
    @Autowired
    private DiscoveryClient discoveryClient; #A
    public Organization getOrganization(String organizationId) {
        RestTemplate restTemplate = new RestTemplate();
        List<ServiceInstance> instances =
            discoveryClient.getInstances("organization-service"); #B
        if (instances.size()==0) return null;
        String serviceUri = String.format
            ("%s/v1/organization/%s", instances.get(0).getUri().toString(),
            organizationId); #C
        ResponseEntity< Organization > restExchange = #D
        restTemplate.exchange(
            serviceUri,
            HttpMethod.GET,
            null, Organization.class, organizationId);
        return restExchange.getBody();
    }
}
```

#A DiscoveryClient is injected into the class.

#B Gets a list of all the instances of organization services.

#C Retrieves the service endpoint we are going to call

#D Uses a standard Spring REST Template class to call the service

The first item of interest in the code is the DiscoveryClient. This is the class you'll use to interact with the Spring Cloud Load Balancer. To retrieve all instances of the organization services registered with Eureka, you can use the getInstances() method, passing in the key of service you're looking for, to retrieve a list of ServiceInstance objects.

The ServiceInstance class is used to hold information about a specific instance of a service, including its hostname, port, and URI.

In the code listing 6.12, you take the first ServiceInstance class in your list to build a target URL that can then be used

to call your service. Once you have a target URL, you can use a standard Spring RestTemplate to call your organization service and retrieve data.

The DiscoveryClient and real life

The reality is that you should only use the DiscoveryClient directly when your service needs to query the load balancer to understand what services and service instances are registered with it. There are several problems with this code, including the following:

You aren't taking advantage of The Spring Cloud Load Balancer client-side load-balancing—By calling the DiscoveryClient directly, you get back a list of services, but it becomes your responsibility to choose which service instances returned you're going to invoke.

You're doing too much work—Right now, you have to build the URL that's going to be used to call your service. It's a small thing, but every piece of code that you can avoid writing is one less piece of code that you have to debug.

Observant Spring developers might have noticed that you're directly instantiating the RestTemplate class in the code. This is antithetical to usual Spring REST invocations, as usually, you'd have the Spring Framework inject the RestTemplate the class using it via the @Autowired annotation.

You instantiated the RestTemplate class in listing 6.11 because once you've enabled the Spring DiscoveryClient in the application class via the @EnableDiscoveryClient annotation, all RestTemplates managed by the Spring framework will have a LoadBalancer-enabled interceptor injected into them that will change how URLs are created with the RestTemplate class. Directly instantiating the RestTemplate class allows you to avoid this behavior.

6.5.2 Invoking services with a LoadBalancer-aware Spring RestTemplate

Next, we're going to see an example of how to use a RestTemplate that's a LoadBalancer-aware. This is one of the

more common mechanisms for interacting with the Load Balancer via Spring. To use a LoadBalancer-aware RestTemplate class, we need to define a RestTemplate bean construction method with a Spring Cloud annotation called @LoadBalanced. For the licensing service, the method that will be used to create the RestTemplate bean can be found in src/main/java/com/optimagrowth/license/LicenseServiceApplication.java.

The following code listing 6.13 shows the getRestTemplate() method that will create the LoadBalancer-backed Spring RestTemplate bean.

Listing 6.13 Annotating and defining a RestTemplate construction method

```
//...Most of import statements have been removed for consiceness
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
@RefreshScope
public class LicenseServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(LicenseServiceApplication.class, args);
    }
    @LoadBalanced #A
    @Bean
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}
```

#A Gets a list of all the instances of organization services.

Now that the bean definition for the backed RestTemplate class is defined, any time you want to use the RestTemplate bean to call a service, you only need to auto-wire it into the class using it.

Using the backed RestTemplate class pretty much behaves like a standard Spring RestTemplate class, except for one small difference in how the URL for target service is defined. Rather than using the physical location of the service in the RestTemplate call, you're going to build the target URL using the Eureka service ID of the service you want to call.

Let's see this difference by looking at the following listing. The code for this code listing 6.14 can be found in the src/main/java/com/optimagrowth/license/service/client/OrganizationRestTemplateClient.java class.

Listing 6.14 Using a LoadBalancer-backed RestTemplate to call a service

```
//Package and import definitions left off for conciseness
@Component
public class OrganizationRestTemplateClient {
@.Autowired
RestTemplate restTemplate;
public Organization getOrganization(String organizationId){
ResponseEntity<Organization> restExchange =
restTemplate.exchange(
"http://organization-service/v1/organization/{organizationId}", #A
HttpMethod.GET,
null, Organization.class, organizationId);
return restExchange.getBody();
}
}
```

#A When using a LoadBalancer-back RestTemplate, you build the target URL with the Eureka service ID.

This code should look somewhat similar to the previous example, except for two key differences. First, the Spring Cloud DiscoveryClient is nowhere in sight. Second, the URL being used in the restTemplate.exchange() call should look odd to you:

```
restTemplate.exchange(  
    "http://organization-service/v1/organization/{organizationId}",  
    HttpMethod.GET, null, Organization.class, organizationId);
```

The server name in the URL matches the application ID of the organization service key that you registered the organization service within Eureka:

```
http://{applicationid}/v1/organization/{organizationId}
```

The LoadBalancer-enabled RestTemplate will parse the URL passed into it and use whatever is passed in as the server name as the key to query the load balancer for an instance of a service. The actual service location and port are entirely abstracted from the developer.

Also, by using the RestTemplate class, the Spring Cloud Load Balancer will round-robin load balance all requests among all the service instances.

6.5.3 Invoking services with Netflix Feign client

An alternative to the Spring LoadBalancer-enabled RestTemplate class is Netflix's Feign client library. The Feign library takes a different approach to call a REST service by having the developer first define a Java interface and then annotating that interface with Spring Cloud annotations to map what Eureka-based service the Spring Cloud Load Balancer will invoke. The Spring Cloud framework will dynamically generate a proxy class that will be used to

invoke the targeted REST service. There's no code being written for calling the service other than an interface definition.

To enable the Feign client for use in your licensing service, you need to add a new annotation, @EnableFeignClients, to the licensing service's src/main/java/com/optimagrowth/license/LicenseServiceApplication.java class. The following code listing 6.15 shows this code.

Listing 6.15 Enabling the Spring Cloud/Netflix Feign client in the licensing service

```
@SpringBootApplication  
@EnableFeignClients #A  
public class LicenseServiceApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(LicenseServiceApplication.class, args);  
    }  
}
```

#A The @EnableFeignClients annotation is needed to use the FeignClient in your code.

Now that we've enabled the Feign client for use in our licensing service let's look at a Feign client interface definition that can be used to call an endpoint on the organization service. The following code listing 6.16 shows an example. The code in this listing can be found in the src/main/java/com/optimagrowth/license/service/client/OrganizationFeignClient.java class.

Listing 6.16 Defining a Feign interface for calling the organization service

```
//Package and import left off for conciseness
```

```
@FeignClient("organization-service") #A
public interface OrganizationFeignClient {
    @RequestMapping(
        method= RequestMethod.GET,
        value="/v1/organization/{organizationId}",
        consumes="application/json") #B
    Organization getOrganization
        (@PathVariable("organizationId") String organizationId); #C
}
```

#A Identify your service to Feign using the FeignClientAnnotation.

#B The path and action to your endpoint is defined using the @RequestMapping annotation.

#C The parameters passed into the endpoint are defined using the @PathVariable endpoint.

We start the Feign example by using the @FeignClient annotation and passing it the name of the application id of the service we want the interface to represent. Next, we'll define a method, getOrganization(), in our interface that can be called by the client to invoke the organization service.

How we define the getOrganization() method looks exactly like how we would expose an endpoint in a Spring Controller class. First, we're going to define a @RequestMapping annotation for the getOrganization() method that will map the HTTP verb and endpoint that will be exposed to the organization service invocation. Second, we'll map the organization ID passed in on the URL to an organizationId parameter on the method call, using the @PathVariable annotation. The return value from the call to the organization service will be automatically mapped to the Organization class that's defined as the return value for the getOrganization() method.

To use the OrganizationFeignClient class, all we need to do is autowire and use it. The Feign Client code will take care of all

the coding work for us.

On error handling

When you use the standard Spring RestTemplate class, all service calls' HTTP status codes will be returned via the ResponseEntity class's getStatusCode() method. With the Feign Client, any HTTP 4xx – 5xx status codes returned by the service being called will be mapped to a FeignException. The FeignException will contain a JSON body that can be parsed for the specific error message.

Feign does provide you the ability to write an error decoder class that will map the error back to a custom Exception class. Writing this decoder is outside the scope of this book, but you can find examples of this in the Feign GitHub repository at (<https://github.com/Netflix/feign/wiki/Custom-error-handling>).

6.6 Summary

- The service discovery pattern is used to abstract away the physical location of services.
- A service discovery engine such as Eureka can seamlessly add and remove service instances from an environment without the service clients being impacted.
- Client-side load balancing can provide an extra level of performance and resiliency by caching the physical location of a service on the client making the service call.
- Eureka is a Netflix project that, when used with Spring Cloud, is easy to set up and configure.
- You used three different mechanisms in Spring Cloud and Netflix Eureka to invoke a service. These mechanisms included
 - o Using a Spring Cloud service DiscoveryClient
 - o Using Spring Cloud and LoadBalancer-backed RestTemplate
 - o Using Spring Cloud and Netflix's Feign client

7 When bad things happen: Resiliency patterns with Spring Cloud and Resilience4j

This chapter covers

- Implementing circuit breakers, fallbacks, and bulkheads
- Using the circuit breaker pattern to conserve microservice client resources
- Using Resilience4j when a remote service is failing
- Implementing Resilience4j's bulkhead pattern to segregate remote resource calls
- Tuning Resilience4j circuit breaker and bulkhead implementations
- Customizing Resilience4j's concurrency strategy

All systems, especially distributed systems, will experience failure. How we build our applications to respond to that failure is a critical part of every software developer's job. However, when it comes to building resilient systems, most software engineers only take into account the complete failure of a piece of infrastructure or critical service. They focus on building redundancy into each layer of their application using techniques such as clustering key servers,

load balancing between services, and segregation of infrastructure into multiple locations.

While these approaches take into account the complete (and often spectacular) loss of a system component, they address only one small part of building resilient systems. When a service crashes, it's easy to detect that it's no longer there, and the application can route around it. However, when a service is running slow, detecting that poor performance and routing around it is extremely difficult because

- 1. Degradation of a service can start out as intermittent and build momentum.** The degradation might occur only in small bursts. The first signs of failure might be a small group of users complaining about a problem until suddenly the application container exhausts its thread pool and collapses completely.
- 2. Calls to remote services are usually synchronous and don't cut short a long-running call.** The caller of a service has no concept of a timeout to keep the service call from hanging out forever. The application developer calls the service to perform an action and waits for the service to return.
- 3. Applications are often designed to deal with complete failures of remote resources, not partial degradations.** Often, as long as the service has not entirely failed, an application will continue to call the service and won't fail fast. The application will continue to call the poorly behaving service. The calling application or service may degrade gracefully or, more likely, crash because of resource exhaustion. Resource

exhaustion is when a limited resource such as a thread pool or database connection maxes out, and the calling client must wait for that resource to become available.

What's insidious about problems caused by poorly performing remote services is that they're not only difficult to detect but can trigger a cascading effect that can ripple throughout an entire application ecosystem. Without safeguards in place, a single poorly performing service can quickly take down multiple applications. Cloud-based, microservice-based applications are particularly vulnerable to these types of outages because these applications are composed of a large number of fine-grained, distributed services with different pieces of infrastructure involved in completing a user's transaction.

Resiliency patterns are one of the most critical aspects of the microservices architecture. This chapter will explain four resiliency patterns and how to use Spring Cloud and Resilience4j to implement them in our licensing service so it can fail fast.

7.1 What are client-side resiliency patterns?

Client resiliency software patterns are focused on protecting a remote resource's (another microservice call or database lookup) client from crashing when the remote resource is failing because of errors or poor performance. The goal of these patterns is to allow the client to "fail fast," not consume valuable resources such as database connections

and thread pools and prevent the problem of the remote service from spreading “upstream” to consumers of the client.

There are four client resiliency patterns:

1. Client-side load balancing
2. Circuit breakers
3. Fallbacks
4. Bulkheads

Figure 7.1 demonstrates how these patterns sit between the microservice service consumer and the microservice.

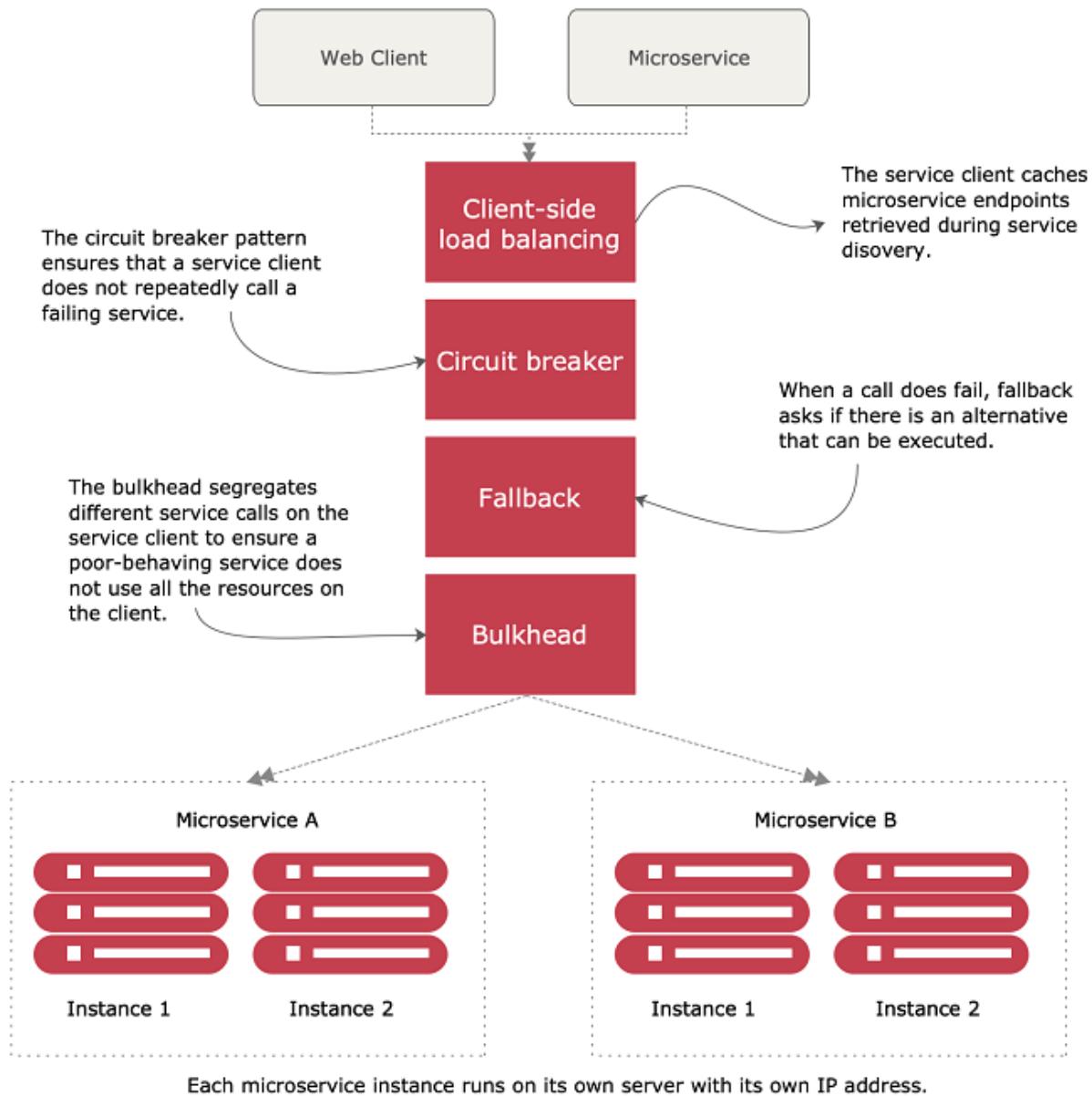


Figure 7.1 The four client resiliency patterns act as a protective buffer between a service consumer and the service.

These patterns are implemented in the client (microservice) calling the remote resource. The implementation of these

patterns logically sits between the client consuming the remote resources and the resource itself.

7.1.1 Client-side load balancing

We introduced the client-side load balancing pattern in the last chapter (chapter 6) when talking about service discovery. Client-side load balancing involves having the client look up all of a service's individual instances from a service discovery agent (like Netflix Eureka) and then caching the physical location of said service instances.

Whenever a service consumer needs to call that service instance, the client-side load balancer will return a location from the pool of service locations it's maintaining.

Because the client-side load balancer sits between the service client and the service consumer, the load balancer can detect if a service instance is throwing errors or behaving poorly. If the client-side load balancer detects a problem, it can remove that service instance from the pool of available service locations and prevent any future service calls from hitting that service instance.

This is precisely the behavior that load balancer libraries provide out of the box with no extra configuration. Because we already covered client-side load balancing with Spring Cloud Load Balancer in chapter 6, we won't go into any more detail on that in this chapter.

7.1.2 Circuit Breaker

The circuit breaker pattern is a client resiliency pattern that is modeled after an electrical circuit breaker. In an electrical system, a circuit breaker will detect if too much current is flowing through the wire. If the circuit breaker detects a problem, it will break the connection with the rest of the electrical system and keep the downstream components from being fried.

With a software circuit breaker, when a remote service is called, the circuit breaker will monitor the call. If the calls take too long, the circuit breaker will intercede and kill the call. Also, the circuit breaker will monitor all calls to a remote resource, and if enough calls fail, the circuit breaker implementation will pop, failing fast and preventing future calls to the failing remote resource.

7.1.3 Fallback processing

With the fallback pattern, when a remote service call fails, rather than generating an exception, the service consumer will execute an alternative code path and try to carry out an action through another means. This usually involves looking for data from another data source or queueing the user's request for future processing. The user's call will not be shown an exception indicating a problem, but they may be notified that their request will have to be fulfilled at a later date.

For instance, let's suppose you have an e-commerce site that monitors your user's behavior and tries to give them recommendations for other items they could buy. Typically, you might call a microservice to run an analysis of the user's

past behavior and return a list of recommendations tailored to that specific user. However, if the preference service fails, your fallback might be to retrieve a more general list of preferences that are based on all user purchases and is much more generalized. This data might come from a completely different service and data source.

7.1.4 Bulkheads

The bulkhead pattern is based on a concept from building ships. With a bulkhead design, a ship is divided into entirely segregated and watertight compartments called bulkheads. Even if the ship's hull is punctured, because the ship is divided into bulkheads, the bulkhead will keep the water confined to the area of the ship where the puncture occurred and prevent the entire ship from filling with water and sinking.

The same concept can be applied to a service that must interact with multiple remote resources. By using the bulkhead pattern, you can break the calls to remote resources into their own thread pools and reduce the risk that a problem with one slow remote resource call will take down the entire application. The thread pools act as the bulkheads for your service. Each remote resource is segregated and assigned to the thread pool. If one service is responding slowly, the thread pool for that one type of service call will become saturated and stop processing requests. Service calls to other services won't become saturated because they're assigned to other thread pools.

7.2 Why client resiliency matters?

We've talked about these different patterns in the abstract; however, let's drill down to a more specific example of where these patterns can be applied. Let's walk through a typical scenario and see why client resiliency patterns, such as the circuit breaker pattern, are critical for implementing a microservice-based architecture running in the cloud.

Figure 7.2 shows a typical scenario involving the use of remote resources like a database and remote service. This scenario doesn't contain any of the resiliency patterns that we have previously mentioned, so it illustrates how an entire architecture, an ecosystem can fail because of a single failing service. Let's take a look.

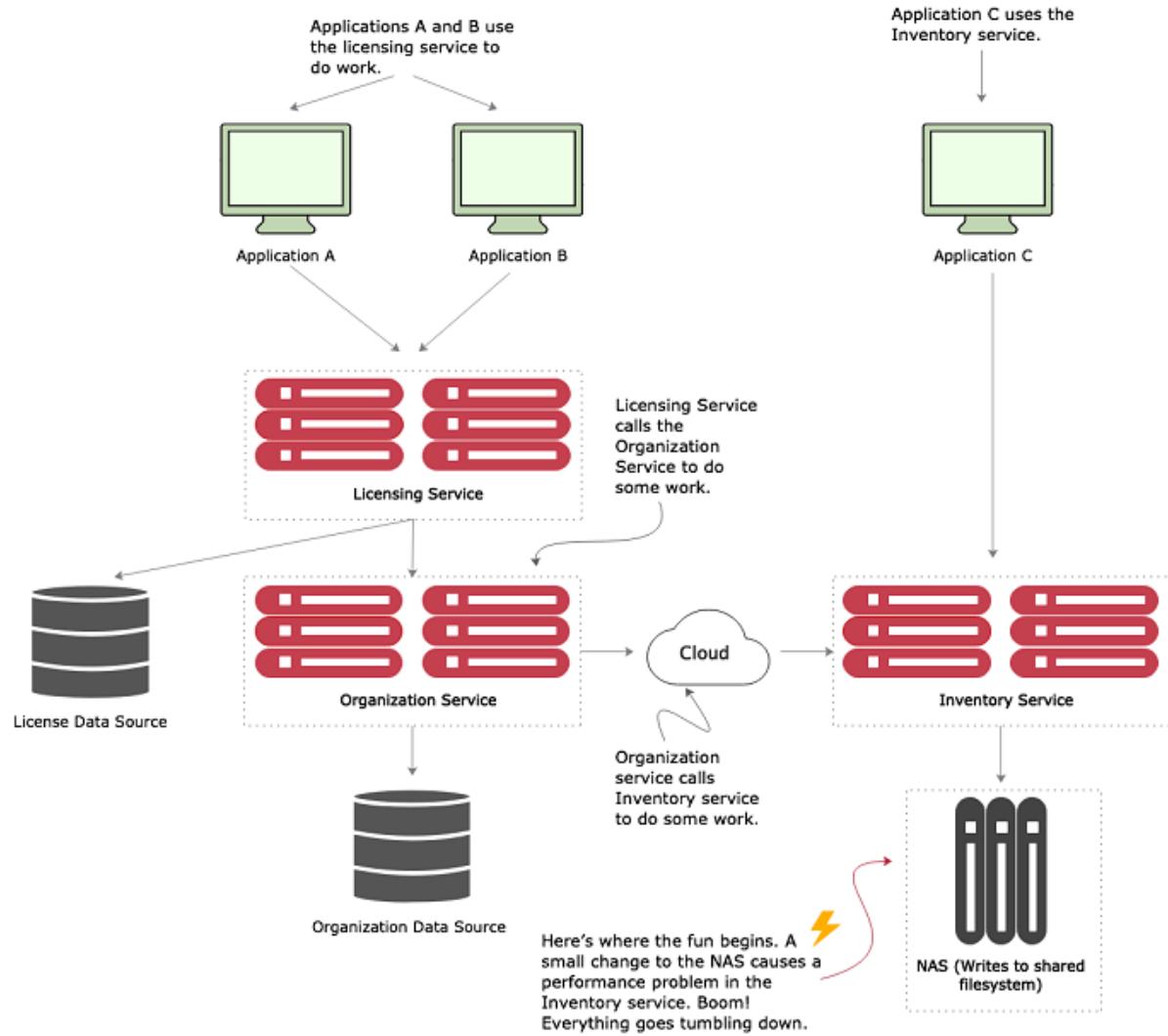


Figure 7.2 An application is a graph of interconnected dependencies. If you don't manage the remote calls between these, one poorly behaving remote resource can bring down all the services in the graph.

In the scenario in figure 7.2, three applications are communicating in one form or another with three different services. Applications A and B communicate directly with the Licensing service. The Licensing service retrieves data from a database and calls the Organization service to do work for it.

The Organization service retrieves data from a completely different database platform and calls out to another service, the Inventory service, from a third-party cloud provider whose service relies heavily on an internal Network Attached Storage (NAS) device to write data to a shared file system. Also, Application C directly calls the Inventory service.

Over the weekend, a network administrator made what they thought was a small tweak to the configuration on the NAS. This change appears to work fine, but on Monday morning, any reads to a particular disk subsystem start performing exceptionally slowly.

The developer who wrote the Organization service never anticipated slowdowns occurring with calls to Inventory service. They wrote their code so that the writes to their database and the reads from the service occur within the same transaction. When the Inventory service starts running slowly, not only does the thread pool for requests to the Inventory service start backing up, the number of database connections in the service container's connection pools become exhausted because these connections are being held open because of the calls out to the Inventory service never complete.

Finally, the Licensing service starts running out of resources because it's calling the Organization service, which is running slow because of the Inventory service. Eventually, all three applications stop responding because they run out of resources while waiting for requests to complete.

This whole scenario could be avoided if a circuit-breaker pattern had been implemented at each point where a

distributed resource had been called (either a call to the database or a call to the service). In figure 7.2, if the call to the Inventory service had been implemented with a circuit breaker, then when the Inventory service started performing poorly, the circuit breaker for that specific call to the Inventory service would have been tripped and failed fast without eating up a thread. If the Organization service had multiple endpoints, only the endpoints that interacted with that specific call to the Inventory service would be impacted. The rest of the Organization Service's functionality would still be intact and could fulfill user requests.

Remember, a circuit breaker acts as a middle man between the application and the remote service. In the previous scenario shown in Figure 7.2, a circuit breaker implementation could have protected Applications A, B, and C from completely crashing.

In figure 7.3, the licensing service is never going to directly invoke the organization service. Instead, when the call is made, licensing service is going to delegate the actual invocation of the service to the circuit breaker, which will take the call and wrap it in a thread (usually managed by a thread pool) that's independent of the originating caller. By wrapping the call in a thread, the client is no longer directly waiting for the call to complete. Instead, the circuit breaker is monitoring the thread and can kill the call if the thread runs too long.

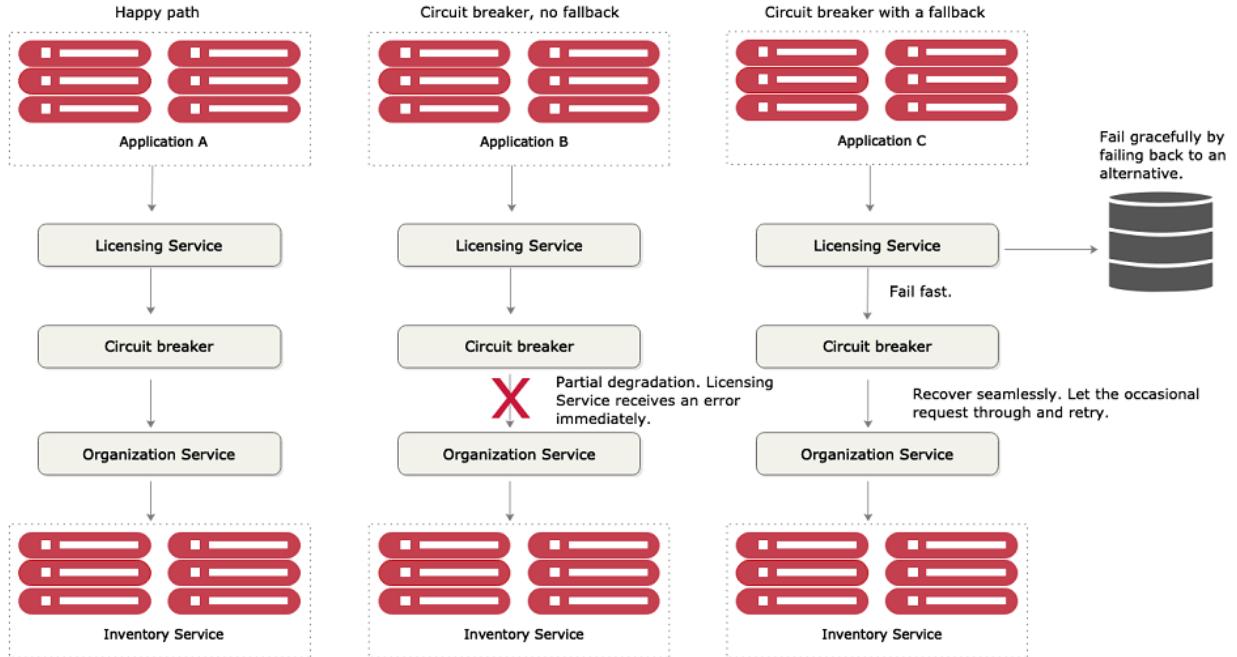


Figure 7.3 The circuit breaker trips and allows a misbehaving service call to fail quickly and gracefully.

Three scenarios are shown in figure 7.3. In the first scenario, the happy path, the circuit breaker will maintain a timer, and if the call to the remote service completes before the timer runs out, everything is good, and the licensing service can continue its work. In the partial degradation scenario two, the licensing service will call the organization service through the circuit breaker. This time, though, the organization service is running slow, and the circuit breaker will kill the connection out to the remote service if it doesn't complete before the timer on the thread maintained by the circuit breaker times out.

The licensing service will then get an error from making the call, but the licensing service won't have resources (that is, its own thread or connection pools) tied up waiting for the

organization service to complete. If the call to the organization service is timed-out by the circuit breaker, the circuit breaker will start tracking the number of failures that have occurred.

If enough errors on the service have occurred within a specific time period, the circuit breaker will now “trip” the circuit, and all calls to the organization service will fail without calling the organization service.

In the third scenario, the licensing service now immediately knows there’s a problem without having to wait for a timeout from the circuit breaker and it can choose to either completely fail or take action using an alternative set of code (a fallback). The organization service will be given an opportunity to recover because the licensing service isn’t calling it while the circuit breaker has been tripped. This allows the organization service to have a breathing room and helps prevent the cascading death that occurs when a service degradation occurs.

Finally, the circuit breaker will occasionally let calls through to a degraded service, and if those calls succeed enough times in a row, the circuit breaker will reset itself.

The key thing a circuit break patterns offers is the ability for remote calls to

- **Fail fast.** When a remote service is experiencing a degradation, the application will fail fast and prevent resource exhaustion issues that generally shut down the entire application. In most outage situations, it’s better to be partially down rather than entirely down.

- **Fail gracefully.** By timing out and failing fast, the circuit breaker pattern gives the application developer the ability to fail gracefully or seek alternative mechanisms to carry out the user's intent. For instance, if a user is trying to retrieve data from one data source, and that data source is experiencing a service degradation, then the application developer could try to retrieve that data from another location.
- **Recover seamlessly.** With the circuit-breaker pattern acting as an intermediary, the circuit breaker can periodically check to see if the resource being requested is back online and re-enable access to it without human intervention.

In a sizeable cloud-based application with hundreds of services, this graceful recovery is critical because it can significantly cut down on the amount of time needed to restore service and significantly lessen the risk of a tired operator or application engineer causing more significant problems by having them intervene directly (restarting a failed service) in the restoration of the service.

Before using Resilience4j, we used to work with Hystrix, one of the most common java libraries to implement the resiliency patterns in microservices. Now, Hystrix has been for some time into a maintenance mode, which means that new features are no longer included, and one of the most recommended libraries to use as a substitute is Resilience4j. That's the main reason why I chose it in this chapter. With Resilience4j, we will have similar and some additional benefits that we are going to see throughout this chapter.

7.3 Implementing Resilience4j

Resilience4j is a fault tolerance library inspired by Hystrix. That offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

- **Circuit breaker.** Use to stop making requests when a service invoked is failing.
- **Retry.** Use to make retries when a service has temporarily failed.
- **Bulkhead.** Limits the number of outgoing concurrent requests to a service to avoid overloading.
- **Rate limit.** Limits the number of calls that a service receives in a time.
- **Fallback.** Alternative paths to failing requests.

With Resilience4j, we can apply several patterns to the same method by defining the annotations for that method. For example, if we want to limit the number of outgoing calls with bulkhead and circuit breaker, we must define the @CircuitBreaker and the @Bulkhead annotation. It is important to highlight that the Resilience4j order is the following: Retry (CircuitBreaker (RateLimiter (TimeLimiter (Bulkhead (Function)))) so, Retry is applied at the end (if needed). This information is valuable when trying to combine patterns, but remember, we can also use the patterns as individual features.

Building implementations of the circuit breaker, retry, rate limit, fallback, and bulkhead patterns requires intimate knowledge of threads and thread management. To implement a high-quality set of implementations for these previously mentioned would require a tremendous amount of work. Fortunately, we can use Spring Boot and Resilience4j

library to provide us a battle-tested library that's used daily in several microservice architectures.

In the next several sections of this chapter, we're going to cover how to

- Configure the licensing service's maven build file (pom.xml) to include the Spring Boot/ Resilience4j wrappers.
- Use the Spring Boot/Resilience4j annotations to wrapper remote calls with a circuit breaker, retry, rate limit and bulkhead patterns.
- Customize the individual circuit breakers on a remote resource to use custom timeouts for each call made.
- Implement a fallback strategy in the event a circuit breaker has to interrupt a call, or the call fails.
- Use individual thread pools in your service to isolate service calls and build bulkheads between different remote resources being called.

7.4 Setting up the licensing server to use Spring Cloud and Resilience4j

To begin our exploration of Resilience4j, we need to set up our project pom.xml to import the dependencies. To achieve that, we will take the licensing service that we've been building and modify its pom.xml by adding the maven dependencies for Resilience4j. The following code listing 7.1 indicates how.

Listing 7.1 Adding Resilience4j dependency on pom.xml of the licensing service

```
<properties>
...
<resilience4j.version>1.5.0</resilience4j.version>
</properties>
<dependencies>
    Rest of pom.xml removed for conciseness
    ...
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-spring-boot2</artifactId>
        <version>${resilience4j.version}</version>
    </dependency>
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-circuitbreaker</artifactId>
        <version>${resilience4j.version}</version>
    </dependency>
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-timelimiter</artifactId>
        <version>${resilience4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
    Rest of pom.xml removed for conciseness
    ...
</dependencies>
```

The `<dependency>` tag (`resilience4j-spring-boot2`) tells Maven to pull down the Resilience4j Spring boot; this library will allow us to use the custom pattern annotations. The dependency `resilience4j-circuitbreaker` and `resilience4j-timelimiter` contains all the logic to implement the circuit breaker and time limiter, so let's add them. The final dependency is the `spring-boot-starter-aop`; we need this library in our project because the `spring-boot2` library requires spring AOP aspects to run. Remember, Aspect-Oriented Programming (AOP) is a programming paradigm that aims to increase modularity by allowing us to separate parts of the program that affect other parts of the system, in

other words, cross-cutting concerns. The AOP adds new behaviors to the existing code without modifying the code itself.

Now that we have the maven dependencies, we can go ahead and begin our Resilience4j implementation using the licensing and organization services we built in previous chapters.

NOTE In case you didn't follow the previous chapters code listings, you can download the code created in chapter 6 from the following link:
<https://github.com/ihuaylupo/manning-smia/tree/master/chapter6/Final>

7.5 Implementing a circuit breaker

To understand circuit breakers, let's make a pause and think of electrical systems. What happens when there is too much current passing through a wire in an electrical system? The circuit breaker detects a problem and breaks the connection with the rest of the system, avoiding further damage in other components. The same happens in our architecture; what we are trying to achieve with circuit breakers is to monitor remote calls and avoid long waits on services. In these scenarios, the circuit breaker will be in charge of killing those connections and monitoring if there are more failing calls to implement a fail fast and prevent future requests to a failing remote resource.

In Resilience4j the circuit breaker is implemented via a finite state machine with the following normal states:

- Closed
- Open
- Half-open

Figure 7.4 shows the interaction between the different states.

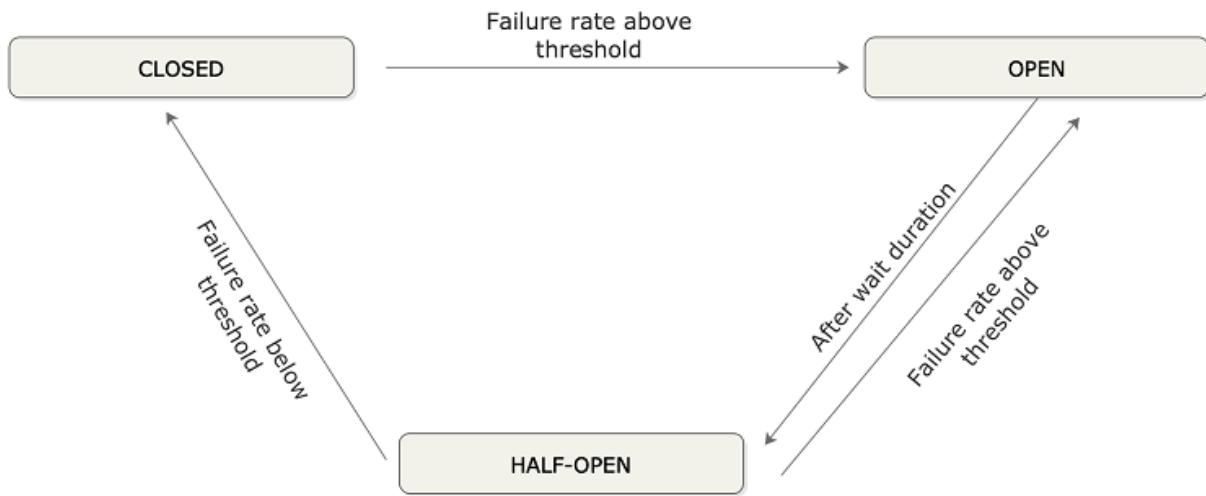


Figure 7.4 Resilience4j Circuit Breaker closed, open and half-open states.

Initially, the circuit breaker starts in a closed state and waits for client requests. The closed state uses a Ring bit buffer to store the success or failure status of the requests. When a successful request is made, it saves a 0 bit in the ring bit buffer. But if it fails to receive a response from the invoked service, it saves a 1 bit. Figure 7.5 shows a Ring buffer with twelve results:

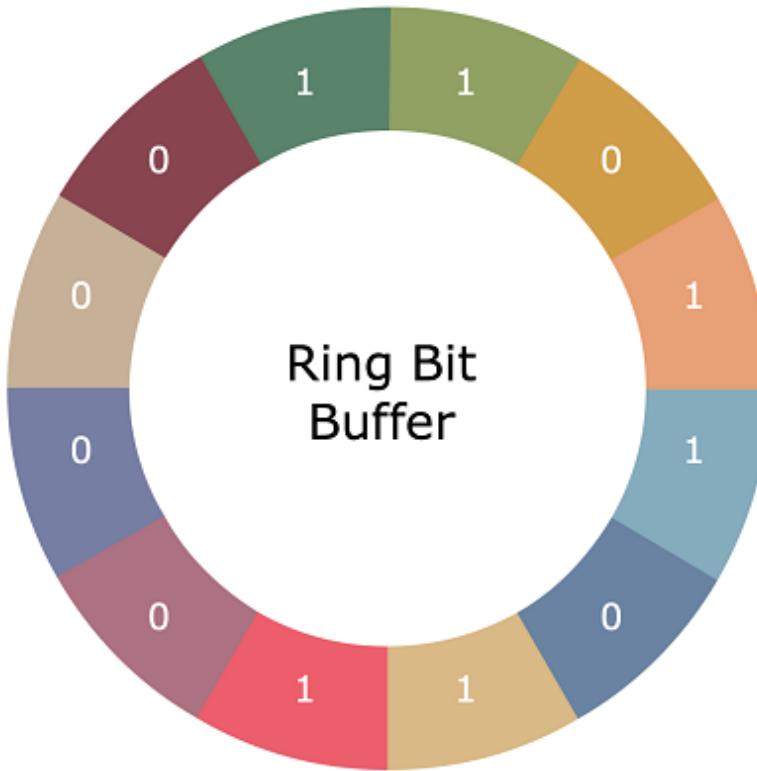


Figure 7.5 Resilience4j Circuit Breaker twelve Ring Bit Buffer. This ring contains 0 for all the successful requests and 1 when it fails to receive a response from the invoked service.

To calculate a failure rate, the ring must be full. For example, in the previous scenario, at least twelve calls must be evaluated before the failure rate can be calculated. If only eleven requests have been evaluated, the circuit breaker will not change to open even if all eleven calls have failed. It is essential to highlight that the circuit breaker will only turn to open when the failure rate is above the configurable threshold.

When the circuit breaker is in the open state, all calls are rejected during a configurable time, and the circuit breaker

throws a `CallNotPermittedException`. Once the configurable time expires, the circuit breaker changes to the Half-open state and allows a configurable number of requests to see if the service is still unavailable.

In the HALF-OPEN state, the circuit breaker uses another configurable ring bit buffer to evaluate the failure rate. If this new failure rate is above the configured threshold, the circuit breaker changes back to OPEN, and if it is below or equal to the threshold, it changes to CLOSED again. This might be somehow confusing, but just remember in the open state the circuit breaker rejects, and in the closed state, the circuit breaker accepts all the requests.

Also, in the Resilience4j circuit breaker pattern, you can define the following additional states.

- **Disabled.** Always allow access.
- **Forced_open.** Always deny access.

It is essential to highlight that the only way to exit from those states is to reset the circuit breaker or trigger a state transition. These two states are not on the scope of this book. If you want to know more about those states, I recommend you read the official Resilience4j documentation (<https://resilience4j.readme.io/v0.17.0/docs/circuitbreaker>).

In this section, we're going to look at implementing Resilience4j in two broad categories. In the first category, we're going to wrap all calls to our database in the licensing and organization service with a Resilience4j circuit breaker. We are then going to wrap the inter-service calls between the licensing service and the organization service using

Resilience4j. While these are two different categories calls, we'll see that the use of Resilience4j will be exactly the same. Figure 7.6 shows what remote resources we're going to wrap with a Resilience4j circuit breaker.

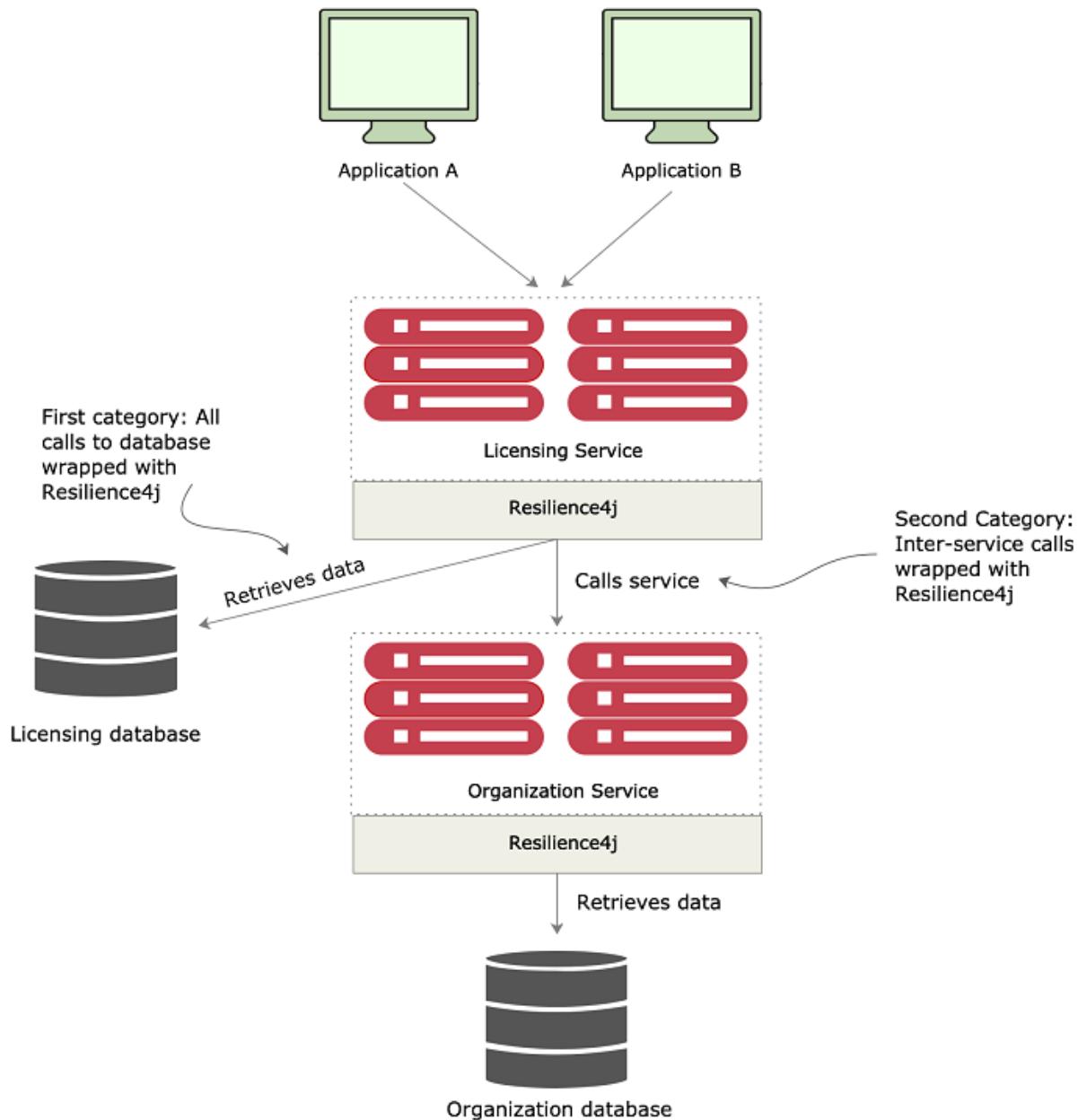


Figure 7.6 Resilience4j sits between each remote resource call and protects the client. It doesn't matter if the remote resource call is a database call or a REST-based service call.

Let's start our Resilience4j discussion by showing how to wrap the retrieval of licensing service data from the licensing database using a synchronous circuit breaker. With a synchronous call, the licensing service will retrieve its data but will wait for the SQL statement to complete or for a circuit-breaker time-out before continuing processing.

Resilience4j and Spring Cloud use the `@CircuitBreaker` annotation to mark Java class methods as being managed by a Resilience4j circuit breaker. When the Spring framework sees the `@CircuitBreaker`, it will dynamically generate a proxy that will wrapper the method and manage all calls to that method through a thread pool of threads specifically set aside to handle remote calls.

So, let's continue by adding the `@CircuitBreaker` annotation to the method `getLicensesByOrganization` of the `src/main/java/com/optimagrowth/license/service/LicenseService.java` class, as shown in the following code listing 7.2

Listing 7.2 Wrapping a remote resource call with a circuit breaker

```
Rest of LicenseService.java removed for conciseness
@CircuitBreaker(name = "licenseService") #A
public List<License> getLicensesByOrganization(String organizationId) {
    return licenseRepository.findByOrganizationId(organizationId);
}
```

#A @CircuitBreaker annotation is used to wrap the getLicenseByOrganization() method with a Resilience4j circuit breaker.

NOTE If you look at the code in listing 7.2 in the source code repository, you'll see several more parameters on the @CircuitBreaker annotation than what's shown in the previous listing. We'll get into those parameters later in the chapter. The code in listing 7.2 is using the @CircuitBreaker annotation with all its default values.

This doesn't look like a lot of code, and it's not, but there is a lot of functionality inside this one annotation. With the use of the @CircuitBreaker annotation, any time the getLicensesByOrganization() method is called, the call will be wrapped with a Resilience4j circuit breaker. The circuit breaker will interrupt any failure call to the getLicensesByOrganization() method.

This code example would be boring if the database is working correctly. So, let's simulate the getLicensesByOrganization() method running into a slow/timeout database query. The following code listing 7.3 demonstrates this.

Listing 7.3 Randomly timing out a call to the licensing service database

```
Rest of LicenseService.java removed for conciseness
private void randomlyRunLong(){ #A
Random rand = new Random();
int randomNum = rand.nextInt(3) + 1;
if (randomNum==3) sleep();
}
private void sleep(){
try {
Thread.sleep(5000); #B
throw new java.util.concurrent.TimeoutException();
} catch (InterruptedException e) {
logger.error(e.getMessage());
}
}
@CircuitBreaker(name = "licenseService")
public List<License> getLicensesByOrganization(String organizationId) {
```

```
randomlyRunLong();
return licenseRepository.findByOrganizationId(organizationId);
}
```

#A The randomlyRunLong() method gives you a one in three chance of a database call running long.

#B You sleep for 5.000 milliseconds (5 seconds) and then throw a Timeout Exception.

If you hit the `http://localhost:8080/v1/organization/e6a625cc-718b-48c2-ac76-1dfdff9a531e/license/` endpoint enough times, you should see the following error message returned from the licensing service.

```
{
  "timestamp": 1595178498383,
  "status": 500,
  "error": "Internal Server Error",
  "message": "No message available",
  "path": "/v1/organization/e6a625cc-718b-48c2-ac76-1dfdff9a531e/license/"}
```

If we keep executing the failing service, the ring buffer would eventually be filled, and we should receive the error shown in Figure 7.7.

The screenshot shows a REST API response in JSON format. The response includes tabs for 'Body', 'Cookies', 'Headers (5)', and 'Test Results'. The 'Body' tab is selected and displays the following JSON:

```
1  {
2    "metadata": {
3      "status": "NOT_ACCEPTABLE"
4    },
5    "errors": [
6      {
7        "message": "CircuitBreaker 'licenseService' is OPEN and does not permit further calls",
8        "code": null,
9        "detail": "CircuitBreaker 'licenseService' is OPEN and does not permit further calls"
10      }
11    ]
12 }
```

The JSON structure indicates a failure with a status of 'NOT_ACCEPTABLE'. The error message is related to a 'CircuitBreaker' named 'licenseService' being in an 'OPEN' state, which prevents further calls. The code and detail fields provide additional context about the error condition.

Figure 7.7 A Circuit Breaker error indicating the circuit breaker is now in the Open state.

Now that we have our circuit breaker working, let's continue by setting up the circuit breaker for the organization microservice.

7.5.1 Adding the circuit breaker to the organization microservice

The beauty of using method-level annotations for tagging calls with circuit-breaker behavior is that it's the same annotation whether you're accessing a database or calling a microservice.

For instance, in our licensing service, we need to look up the name of the organization associated with the license. If we want to wrap our call to the organization service with a circuit breaker, it's as simple as breaking the RestTemplate call into its method and annotating it with the @CircuitBreaker annotation:

```
@CircuitBreaker(name = "organizationService")
private Organization getOrganization(String organizationId) {
    return organizationRestClient.getOrganization(organizationId);
}
```

NOTE While using the @CircuitBreaker is easy to implement, we do need to be careful about using the default values of the @CircuitBreaker annotation. I highly recommend, to always analyze and set the configuration that best suits your needs. To see the default values of your circuit breaker, you can hit the following URL (http://localhost:<service_port>/actuator/health). By default, the circuit breaker exposes the configuration in the Spring Boot Actuator health service.

7.5.2 Customizing the circuit breaker

In this section, I will answer one of the most common questions developers encountered while using Resilience4j. This question is how to customize the Resilience4j circuit breaker. This is easily accomplished by adding some parameters to the application.yml or bootstrap.yml or the service configuration file located in the Spring Configuration server repository. The following code listing 7.4 demonstrates how to customize the circuit breaker pattern.

Listing 7.4 Customizing the circuit breaker in bootstrap.yml file of the licensing service

```
Rest of bootstrap.yml removed for conciseness
resilience4j.circuitbreaker:
instances:
licenseService: #A
registerHealthIndicator: true #B
ringBufferSizeInClosedState: 5 #C
ringBufferSizeInHalfOpenState: 3 #D
waitDurationInOpenState: 10s #E
failureRateThreshold: 50 #F
recordExceptions: #G
- org.springframework.web.client.HttpServerErrorException
- java.io.IOException
- java.util.concurrent.TimeoutException
- org.springframework.web.client.ResourceAccessException
organizationService: #H
registerHealthIndicator: true
ringBufferSizeInClosedState: 6
ringBufferSizeInHalfOpenState: 4
waitDurationInOpenState: 20s
failureRateThreshold: 60
```

#A License service instance configuration. (Name given to the circuit breaker in the annotation)

#B Parameter that indicates if we want to expose the configuration over the health endpoint.

#C Sets the ring buffer size at the closed state.

#D Sets the ring buffer size in the half-open state.

#E Sets the wait duration in the open state.

```
#F Sets the failure rate threshold percentage.  
#G Set the exceptions that should be recorded as failure  
#H Organization service instance configuration. (Name given to the circuit  
breaker in the annotation)
```

Resilience4j allows us to customize the behavior of the circuit breakers through the application properties. We can configure as many instances as we want, and each instance can have a different configuration. The previous code listing 7.4 contains the following configuration:

- **ringBufferSizeInClosedState.** Sets the size of the ring bit buffer when the circuit breaker is in the closed state. The default value is 100.
- **ringBufferSizeInHalfOpenState.** Sets the size of the ring bit buffer when the circuit breaker is in the half-open state. The default value is 10
- **waitDurationInOpenState.** Sets the time the circuit breaker should wait before changing the status from open to half-open. The default value is 60000ms.
- **failureRateThreshold.** This parameter configures the percentage of the failure rate threshold. Remember, when the failure rate is greater or equal than this threshold, the circuit breaker changes to the open state and starts short-circuiting calls. The default value is 50.
- **recordExceptions.** List of exceptions that will be considered as failures. By default, all exceptions are recorded as failures.

In this book, I will not cover all of the Resilience4j circuit breaker parameters, but if you want to know more about the possible configuration parameters, I recommend you visit the following link.

<https://resilience4j.readme.io/docs/circuitbreaker>

7.6 Fallback processing

Part of the beauty of the circuit breaker pattern is that because a “middle man” is between the consumer of a remote resource and the resource itself, we have an opportunity for the developer to intercept a service failure and choose an alternative course of action to take.

In Resilience4j, this is known as a fallback strategy and is easily implemented. Let’s see how to build a simple fallback strategy for our licensing service that returns a licensing object that says no licensing information is currently available. The following code listing 7.5 demonstrates this.

Listing 7.5 Implementing a fallback in Resilience4j

```
Rest of LicenseService.java removed for conciseness
@CircuitBreaker(name= "licenseService",fallbackMethod= "buildFallbackLicenseList")
#A
public List<License> getLicensesByOrganization(String organizationId) throws
TimeoutException {
logger.debug("getLicensesByOrganization Correlation id: {}",

UserContextHolder.getContext().getCorrelationId());
randomlyRunLong();
return licenseRepository.findByOrganizationId(organizationId);
}
private List<License> buildFallbackLicenseList(String organizationId, Throwable t)
{ #B
List<License> fallbackList = new ArrayList<>();
License license = new License();
license.setLicenseId("0000000-00-00000");
license.setOrganizationId(organizationId);
license.setProductName("Sorry no licensing information currently available");
fallbackList.add(license);
return fallbackList;
}
```

#A The fallbackMethod attribute defines a single function in your class that will be called if the call service fails.

#B In the fallback method you return a hard-coded value.

To implement a fallback strategy with Resilience4j, we have to do two things. First, we need to add an attribute called `fallbackMethod` to the `@CircuitBreaker` annotation or any other annotation, I will explain this later on. This attribute will contain the name of a method that will be called when Resilience4j has to interrupt a call because of a failure.

The second thing we need to do is define a fallback method to be executed. This fallback method must reside in the same class as the original method that was protected by the `@CircuitBreaker`. To create the fallback method in Resilience4j, we need to create a method that contains the same method signature as the originating function plus one extra parameter that is the target exception parameter. The idea with the same signature is to pass all the parameters from the original method to the fallback method.

In the example in code listing 7.5, the fallback method `buildFallbackLicenseList()` is simply constructing a single `License` object containing dummy information. We could have our fallback method read this data from an alternative data source, but for demonstration purposes, we're going to construct a list that would have been returned by our original function call.

On fallbacks

Here are a few things to keep in mind as you determine whether you want to implement a fallback strategy:

1. Fallbacks are a mechanism to provide a course of action when a resource has timed out or failed. If you find yourself using fallbacks to catch a timeout exception and then doing nothing more than logging the error, then you should probably use a standard `try..catch` block around your service invocation, catch the exception, and put the logging logic in the `try..catch` block.

2. Be aware of the actions you're taking with your fallback functions. If you call out to another distributed service in your fallback service, you may need to wrap the fallback with a `@CircuitBreaker` annotation. Remember, the same failure that you're experiencing with your primary course of action might also impact your secondary fallback option. Code defensively.

Now that we have our fallback method in place let's go ahead and call our endpoint again. This time when we hit it and encounter a timeout error (remember we have a one in 3 chance), we shouldn't get an exception back from the service call, but instead, have the dummy license values returned like figure 7.8

The screenshot shows a JSON response in a developer tools interface. The 'Body' tab is selected, displaying a JSON object with the following structure:

```
1 [  
2 {  
3   "licenseId": "0000000-00-00000",  
4   "organizationId": "e6a625cc-718b-48c2-ac76-1dfdff9a531e",  
5   "productName": "Sorry no licensing information currently available",  
6   "links": []  
7 }]  
8 ]
```

A curly arrow points from the text "Results of fallback code" to the "productName" field of the JSON object.

Figure 7.8 Your service invocation using a Resilience4j fallback.

7.7 Implementing the bulkhead pattern

In a microservice-based application, we'll often need to call multiple microservices to complete a particular task. Without using a bulkhead pattern, the default behavior for these calls is that the calls are executed using the same threads that are reserved for handling requests for the entire Java container. In high volumes, performance problems with one service out of many can result in all of the threads for the Java container being maxed out and waiting to process work, while new requests for work back up. The Java container will eventually crash. The bulkhead pattern segregates remote resource calls in their own thread pools so that a single misbehaving service can be contained and not crash the container.

Resilience4j provides two different implementations of the bulkhead pattern used to limit the number of concurrent executions.

- **Semaphore Bulkhead.** Uses a semaphore isolation approach and limits the number of concurrent requests to the service; once the limit is hit, it starts rejecting requests.
- **ThreadPool Bulkhead.** Uses a bounded queue and a fixed thread pool. This approach only rejects when the pool and the queue are full.

Resilience4j, by default, uses the semaphore bulkhead type. Figure 7.9 illustrates this type.

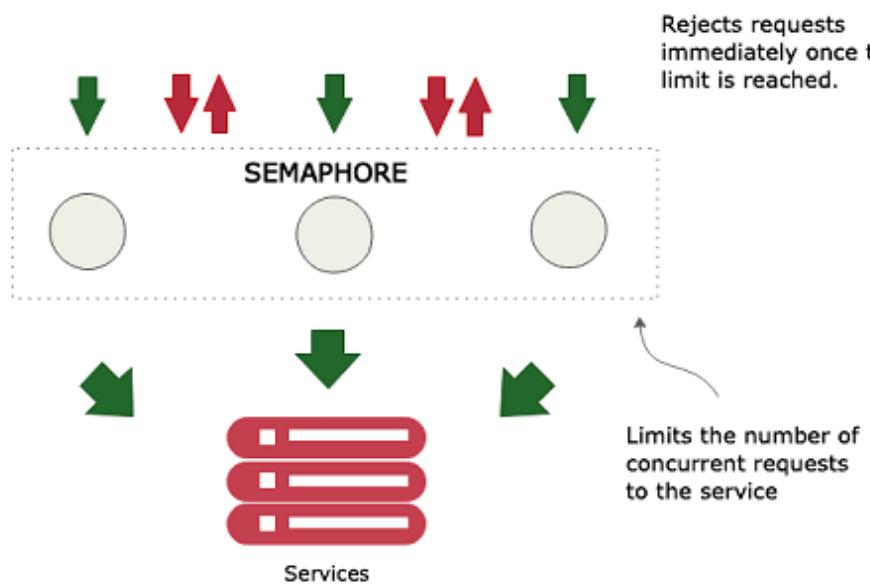


Figure 7.9 Default Resilience4j bulkhead type, the semaphore approach.

This model works fine when we have a small number of remote resources being accessed within an application, and the call volumes for the individual services are relatively evenly distributed. The problem is if we have services that have far higher volumes or longer completion times than other services, we can end up introducing thread exhaustion into our thread pools because one service ends up dominating all of the threads in the default thread pool.

Fortunately, Resilience4j provides an easy-to-use mechanism for creating bulkheads between different remote resource calls. Figure 7.10 shows managed resources look like when they're segregated into their own “bulkheads.”

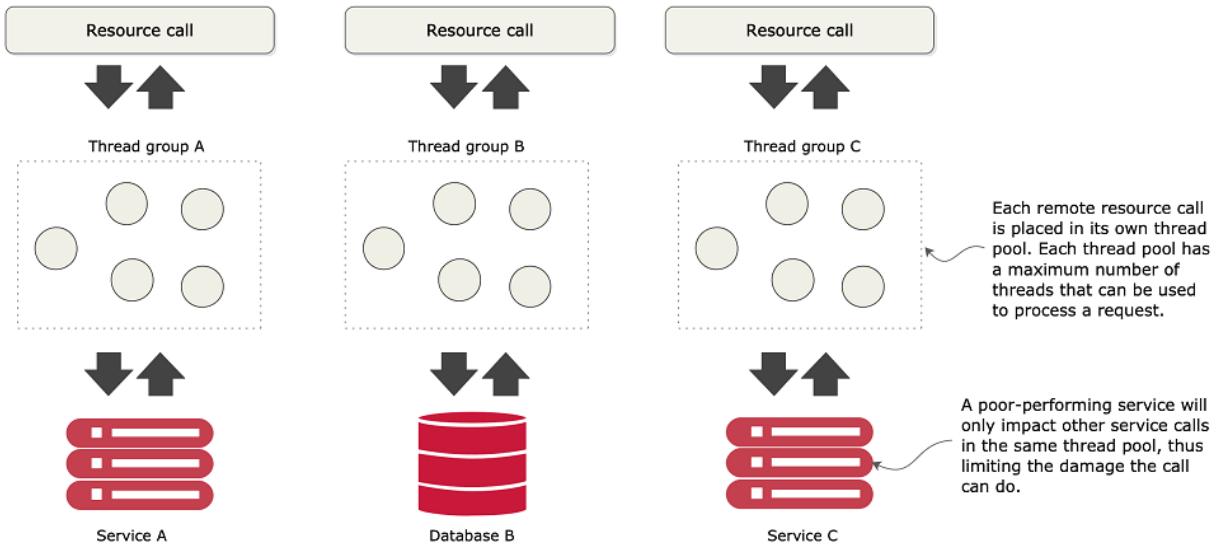


Figure 7.10 Resilience4j command tied to segregated thread pools

To implement the bulkheads patterns in Resilience4j, we need to use additional configuration to combine it with the `@CircuitBreaker` annotation. Let's look at some code that will

1. Set up a separate thread pool for the `getLicensesByOrganization()` call
2. Create the bulkhead configuration in the `bootstrap.yml` file.
3. In the semaphore approach set the `maxConcurrentCalls` and the `maxWaitDuration` time.
4. In the threadpool approach set also the `maxThreadPoolSize`, `coreThreadPoolSize`, `queueCapacity`, `keepAliveDuration`.

The following code listing 7.6 shows the `bootstrap.yml` for the licensing service where it contains the bulkhead

configuration parameters.

Listing 7.6 Configuring the bulkhead pattern in the bootstrap.yml

```
Rest of bootstrap.yml removed for conciseness
resilience4j.bulkhead:
instances:
bulkheadLicenseService:
maxWaitDuration: 10ms #A
maxConcurrentCalls: 20 #B
resilience4j.thread-pool-bulkhead:
instances:
bulkheadLicenseService:
maxThreadPoolSize: 1 #C
coreThreadPoolSize: 1 #D
queueCapacity: 1 #E
keepAliveDuration: 20ms #F
```

#A The maxWaitDuration attribute defines the maximum amount of time a thread should be blocked when trying to enter a full bulkhead.

#B The maxConcurrentCalls attribute lets you define the maximum amount of concurrent calls.

#C The maxThreadPoolSize attribute lets you define the maximum number of threads in the thread pool.

#D The coreThreadPoolSize lets you define the core thread pool size.

#E The queueCapacity lets you define the capacity of the queue.

#F The keepAliveDuration lets you define the maximum time that idle threads will wait for new tasks before terminating.

Resilience4j also allows us to customize the behavior of the bulkhead patterns through the application properties. Like the circuit breaker, we can create as many instances as we want, and each instance can have different configurations. The previous code listing 7.6 contains the following configuration:

- **maxWaitDuration.** Sets the maximum amount of time a thread should be blocked to enter a saturated bulkhead. The default value is 0.

- **maxConcurrentCalls.** Sets the maximum amount of concurrent calls allowed by the bulkhead. The default value is 25.
- **maxThreadPoolSize.** Sets the maximum thread pool size. The default value is the `Runtime.getRuntime().availableProcessors()`
- **coreThreadPoolSize.** Sets the core thread pool size. The default value is the `Runtime.getRuntime().availableProcessors() - 1`
- **queueCapacity.** Sets the capacity of the queue. The default value is 100.
- **KeepAliveDuration.** It sets the maximum time that idle threads will wait for new tasks before terminating. This happens when the number of threads is higher than the core thread. The default value is 20ms.

What's the proper sizing for a custom thread pool? To answer that question, you can use the next formula:

(requests per second at peak when the service is healthy * 99th percentile latency in seconds) + small amount of extra threads for overhead

We often don't know the performance characteristics of a service until it has been under load. A key indicator that the thread pool properties need to be adjusted is when a service call is timing out, even if the targeted remote resource is healthy.

The following code listing 7.7 demonstrates how to set up a bulkhead around all calls surrounding the look-up of licensing data from our licensing service.

Listing 7.7 Creating a bulkhead around the `getLicensesByOrganization()` method

```
Rest of LicenseService.java removed for conciseness
@CircuitBreaker(name= "licenseService", fallbackMethod=
"buildFallbackLicenseList")
@Bulkhead(name= "bulkheadLicenseService", fallbackMethod=
"buildFallbackLicenseList") #A
public List<License> getLicensesByOrganization(String organizationId) throws
TimeoutException {
logger.debug("getLicensesByOrganization Correlation id: {}", UserContextHolder.getContext().getCorrelationId());
randomlyRunLong();
return licenseRepository.findByOrganizationId(organizationId);
}
```

#A Sets the instance name and the fallback method for the Bulkhead pattern.

The first thing we should notice is that we've introduced a new annotation, the @Bulkhead annotation. This indicates that we are going to set up a bulkhead pattern. If we set no further values in the application properties, Resilience4j will use the default value I previously mentioned.

The second thing to highlight from the previous code is that we are not setting up the bulkhead type so in this case the bulkhead pattern is going to work with the semaphore approach. In order to change this and to use the threadPool approach we need to add the type to the @Bulkhead annotation.

```
@Bulkhead(name = "bulkheadLicenseService", type = Bulkhead.Type.THREADPOOL,
fallbackMethod = "buildFallbackLicenseList")
```

7.8 Implementing the retry pattern

As its name implies, the retry pattern is responsible for making attempts when a service has temporarily failed. The key concept behind this pattern is to provide a way to get the expected response by retrying to invoke the same service one or more times despite the failure, for example, a network disruption.

For this pattern, we must specify the number of retries for a given service instance, and the interval we want to pass between each retry. Like the circuit breaker, Resilience4j allows us to configure to which exceptions we want to retry and not to retry.

The following code listing 7.8 shows the bootstrap.yml for the licensing service, where it contains the retry configuration parameters.

Listing 7.8 Configuring the retry pattern in the bootstrap.yml

```
Rest of bootstrap.yml removed for conciseness
resilience4j.retry:
instances:
retryLicenseService:
maxRetryAttempts: 5 #A
waitDuration: 10000 #B
retry-exceptions: #C
- java.util.concurrent.TimeoutException
```

#A The maxRetryAttempts allows you to define the maximum number of retry attempts.

#B The waitDuration allows you to define the duration between the retry attempts.

#C The retry-exceptions allows you to define the list of exceptions you want to retry.

The first parameter, the `maxRetryAttempts` allows us to define the maximum number of retry attempts for our service. The default value for this parameter is 3. The second parameter, the `waitDuration` allows us to define the duration between the retry attempts. The default value for this parameter is 500ms. The third parameter, the `retryExceptions` sets a list of error classes that will be retried. The default value is empty.

For purposes of this book, I will only be using these three parameters, but you can also set the following patterns:

- **intervalFunction**. Sets a function to update the waiting interval after a failure.
- **retryOnResultPredicate**. Configures a predicate that evaluates if a result should be retried. It is important to highlight that this predicate should return true if we want to retry.
- **retryOnExceptionPredicate**. Configures a predicate that evaluates if an exception should be retried. Same as the previous predicate, we must return true if we want to retry.
- **ignoreExceptions**. Sets a list of error classes that are ignored and will not be retried. The default value is empty.

The following code listing 7.9 demonstrates how to set up the retry pattern around all calls surrounding the look-up of licensing data from our licensing service.

Listing 7.9 Creating a bulkhead around the `getLicensesByOrganization()` method

```
Rest of LicenseService.java removed for conciseness
@CircuitBreaker(name= "licenseService", fallbackMethod=
"buildFallbackLicenseList")
```

```
@Retry(name = "retryLicenseService", fallbackMethod = "buildFallbackLicenseList")
#A
@Bulkhead(name= "bulkheadLicenseService", fallbackMethod=
"buildFallbackLicenseList")
public List<License> getLicensesByOrganization(String organizationId) throws
TimeoutException {
logger.debug("getLicensesByOrganization Correlation id: {}", UserContextHolder.getContext().getCorrelationId());
randomlyRunLong();
return licenseRepository.findByOrganizationId(organizationId);
}
```

#A Sets the instance name and the fallback method for the retry pattern.

Now that we know how to implement the circuit breaker and the retry pattern let's continue with the rate limiter. Remember, Resilience4j allows us to combine different patterns in the same method calls.

7.9 Implementing the rate limiter pattern

The idea with this Retry pattern is to stop overloading the service with more calls more than it can consume in a given time. This is an imperative technique to prepare our API for high availability and reliability. In modern cloud architectures, is also a good option to have auto scaling, but I will not cover this topic on the book. Resilience4j provides two implementations for this pattern the AtomicRateLimiter and SemaphoreBasedRateLimiter. The default implementation for the RateLimiter is the AtomicRateLimiter.

The SemaphoreBasedRateLimiter is the simplest one. This implementation is based on the idea of having one java.util.concurrent.Semaphore to store the current

permissions. In this scenario, all the user threads will call the method `semaphore.tryAcquire`, this method will trigger a call to an additional internal thread by executing the `semaphore.release` when new `limitRefreshPeriod` starts.

Unlike the `SemaphoreBasedRate`, the `AtomicRateLimiter` does not need the management of threads because the user threads themselves to execute all the permissions logic. The `AtomicRateLimiter` splits all nanoseconds from the start into cycles, and each cycle duration is the refresh period in nanoseconds. Then at the beginning of each cycle should set the active permissions for a limit for period.

To better understand this approach, let's highlight the following concepts.

- **ActiveCycle.** The cycle number that was used by the last call.
- **ActivePermissions.** The count of available permissions after the last call.
- **NanosToWait.** The count of nanoseconds to wait for permission for the last call.

This implementation contains a tricky logic, to better understand it, we can consider the following Resilience4j statements:

- Cycles are equal time pieces.
- If the available permissions are not enough, we can perform a permission reservation, just by decreasing the current permissions and calculating the time we need to wait for it to appear.

For this pattern, we must specify the timeout duration, the limit refresh, and the limit for the period. The following code

listing 7.10 shows the bootstrap.yml for the licensing service, where it contains the retry configuration parameters.

Listing 7.10 Configuring the retry pattern in the bootstrap.yml

```
Rest of bootstrap.yml removed for conciseness
resilience4j.ratelimiter:
instances:
licenseService:
timeoutDuration: 1000ms #A
limitRefreshPeriod: 5000 #B
limitForPeriod: 5 #C
```

#A The timeoutDuration allows you to define the time a thread waits for permission.

#C The limitRefreshPeriod allows you to define the period of a limit refresh.

#B The limitForPeriod allows you to define the number of permissions available during a limit refresh period.

The first parameter, the timeoutDuration allows us to define the time a thread waits for permission; the default value for this parameter is 5s. The second parameter, the limitRefreshPeriod, will enable us to set the period of a limit refresh. Remember, after each period, the rate limiter sets the permissions count back to the limitForPeriod value. The default value for the limitRefreshPeriod is 500ns (nanoseconds). The final parameter is the limitForPeriod, and this parameter allows us to set the number of permissions available during one limit refresh period. The default value for the limitForPeriod is 50.

The following code listing 7.11 demonstrates how to set up the retry pattern around all calls surrounding the look-up of licensing data from our licensing service.

Listing 7.11 Creating a bulkhead around the getLicensesByOrganization() method

```
Rest of LicenseService.java removed for conciseness
@CircuitBreaker(name= "licenseService", fallbackMethod=
"buildFallbackLicenseList")
@RateLimiter(name = "licenseService", fallbackMethod = "buildFallbackLicenseList")
#A
@Retry(name = "retryLicenseService", fallbackMethod = "buildFallbackLicenseList")
@Bulkhead(name= "bulkheadLicenseService", fallbackMethod=
"buildFallbackLicenseList")
public List<License> getLicensesByOrganization(String organizationId) throws
TimeoutException {
logger.debug("getLicensesByOrganization Correlation id: {}",
UserContextHolder.getContext().getCorrelationId());
randomlyRunLong();
return licenseRepository.findByOrganizationId(organizationId);
}
```

#A Sets the instance name and the fallback method for the rate limiter pattern.

The main difference between the bulkhead and the Rate limiter pattern is that bulkhead is in charge of limiting the number of concurrent calls at a time. For example, it only allows X concurrent calls at a time. With Rate limiter, we can limit the number of total calls in a given time. For example, allow X number of calls every Y seconds.

In order to choose from one another, double-check what your needs are. If you want to block concurrent times, your best choice is a bulkhead, but if you want to limit the total number of calls in a specific time, your best option is RateLimiter. If you are looking for both scenarios, remember that you can also combine them.

7.10 ThreadLocal and Resilience4j

In this section, we will be defining some values in ThreadLocal just to see if there are propagated through the methods using the Resilience4J annotations. Remember, Java ThreadLocal allows us to create variables that can be read and write only by the same threads. For example, when we work with threads, all the threads of a specific object share its variables, making them not thread-safe. The most common way to make it thread-safe in Java is to use synchronization, but if we want to avoid synchronization, we can also use ThreadLocal variables.

Let's see a concrete example. Often in a REST-based environment, we are going to want to pass contextual information to a service call that will help us operationally manage the service. For example, we might pass a correlation ID or authentication token in the HTTP header of the REST call that can then be propagated to any downstream service calls. The correlation ID allows us to have a unique identifier that can be traced across multiple service calls in a single transaction.

To make this value available anywhere in our service call, we might use a Spring Filter class to intercept every call into your REST service and retrieve this information from the incoming HTTP request and store this contextual information in a custom UserContext object. Then, anytime our code needs to access this value in our REST service call, our code can retrieve the UserContext from the ThreadLocal storage variable and read the value.

The following code listing 7.12 shows an example of a Spring Filter that we can use in our licensing service.

NOTE You can find this code at /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContextFilter.java in the source code of chapter 7. The link of the repository is the following: <https://github.com/ihuaylupo/manning-smia/tree/master/chapter7>

Listing 7.12 The UserContextFilter parsing the HTTP header and retrieving data

```
package com.optimagrowth.license.utils;
...
Imports removed for conciseness
@Component
public class UserContextFilter implements Filter {
    private static final Logger logger =
        LoggerFactory.getLogger(UserContextFilter.class);
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
        servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
        UserContextHolder.getContext().setCorrelationId(
            httpServletRequest.getHeader(UserContext.CORRELATION_ID)); #A
        UserContextHolder.getContext().setUserId(
            httpServletRequest.getHeader(UserContext.USER_ID)); #A
        UserContextHolder.getContext().setAuthToken(
            httpServletRequest.getHeader(UserContext.AUTH_TOKEN)); #A
        UserContextHolder.getContext().setOrganizationId(
            httpServletRequest.getHeader(UserContext.ORGANIZATION_ID)); #A
        filterChain.doFilter(httpServletRequest, servletResponse);
    }
    ...
    Rest of UserContextFilter.java removed for conciseness
}
```

#A Retrieving values set in the HTTP header of the call into a UserContext, which is stored in UserContextHolder.

The UserContextHolder class is used to store the UserContext in a ThreadLocal class. Once it's stored in the ThreadLocal storage, any code that's executed for a request will use the UserContext object stored in the UserContextHolder. The UserContextHolder class is shown in the following code listing 7.13. This class is found at /licensing-

service/src/main/java/com/optimagrowth/license/utils/UserContextHolder.java

Listing 7.13 All UserContext data is managed by UserContextHolder

```
...
Imports removed for conciseness
public class UserContextHolder {
    private static final ThreadLocal<UserContext> userContext #A
    = new ThreadLocal<UserContext>();
    public static final UserContext getContext(){ #B
        UserContext context = userContext.get();
        if (context == null) {
            context = createEmptyContext();
            userContext.set(context);
        }
        return userContext.get();
    }
    public static final void setContext(UserContext context) {
        userContext.set(context);
    }
    public static final UserContext createEmptyContext(){
        return new UserContext();
    }
}
```

#A The UserContext is stored in a static ThreadLocal variable.

#B The getContext() method will retrieve the UserContext object for consumption.

NOTE We must be careful when we work directly with the ThreadLocal. An incorrect development inside the ThreadLocal can lead to memory leak issues in our application.

The UserContext is a POJO class that contains all the specific data we want to store in the UserContextHolder. The following code listing 7.14 shows the content of this class. This class is located at /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContext.java

Listing 7.14 User Context Code

```

...
Imports removed for conciseness
@Component
public class UserContext {
    public static final String CORRELATION_ID = "tmx-correlation-id";
    public static final String AUTH_TOKEN = "tmx-auth-token";
    public static final String USER_ID = "tmx-user-id";
    public static final String ORGANIZATION_ID = "tmx-organization-id";
    private String correlationId= new String();
    private String authToken= new String();
    private String userId = new String();
    private String organizationId = new String();
    public String getCorrelationId() { return correlationId;}
    public void setCorrelationId(String correlationId) {
        this.correlationId = correlationId;
    }
    public String getAuthToken() {
        return authToken;
    }
    public void setAuthToken(String authToken) {
        this.authToken = authToken;
    }
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getOrganizationId() {
        return organizationId;
    }
    public void setOrganizationId(String organizationId) {
        this.organizationId = organizationId;
    }
}

```

The last step to finish our example, is adding the logging instruction to the LicenseController.java class found at com/optimagrowth/license/controller/ LicenseController.java. The following code listing 7.15 will show you how.

Listing 7.15 Add logger to LicenseController getLicenses() method.

```

//Some code removed for conciseness
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
@RestController
@RequestMapping(value="v1/organization/{organizationId}/license")
public class LicenseController {

```

```

private static final Logger logger =
LoggerFactory.getLogger(LicenseController.class);
// Some code removed for conciseness
@RequestMapping(value = "/", method = RequestMethod.GET)
public List<License> getLicenses(@PathVariable("organizationId") String
organizationId) {
logger.debug("LicenseServiceController Correlation id: {}",
UserContextHolder.getContext().getCorrelationId());
return licenseService.getLicensesByOrganization(organizationId);
}
}

```

At this point, we have several log statements in our licensing service. We already add logging to the following licensing service classes and methods:

- com/optimagrowth/license/utils/UserContextFilter.java
doFilter() method
- com/optimagrowth/license/controller/LicenseController.java
va getLicenses() method
- com/optimagrowth/license/service/LicenseService.java
getLicensesByOrganization() method. This method is annotated with a @CircuitBreaker, @Retry, @Bulkhead and @RateLimiter.

To execute our example, we'll call our service passing in a correlation ID using an HTTP header called tmx-correlation-id and a value of TEST-CORRELATION-ID. Figure 7.11 shows a HTTP GET call to

<http://localhost:8080/v1/organization/958aa1bf-18dc-405c-b84a-b69f04d98d4f/license/> in Postman.

The screenshot shows the Postman interface with a request titled "Get licenses by Organization". The method is set to "GET" and the URL is "http://localhost:8080/v1/organization/958aa1bf-18dc-405c-b84a-b69f04d98d4f/license/". The "Headers" tab is selected, displaying one header: "tmx-correlation-id" with the value "TEST-CORRELATION-ID". Other tabs like "Params", "Authorization", "Body", "Pre-request Script", "Tests", and "Settings" are also visible.

Figure 7.11 Adding a correlation ID to the licensing service call's HTTP Header.

Once this call is submitted, we should see three log messages in the console writing out the passed in correlation ID as it flows through the UserContext, LicenseController and LicenseService classes.

```
UserContextFilter Correlation id: TEST-CORRELATION-ID  
LicenseServiceController Correlation id: TEST-CORRELATION-ID  
LicenseService:getLicensesByOrganization Correlation id:
```

If you don't see the log messages on your console, just add the following code lines shown in code listing 7.16 on the application.yml or application.properties file of the licensing service.

Listing 7.16 Logger configuration on the licensing service application.yml file

```
//Some code removed for conciseness  
logging:  
  level:
```

```
org.springframework.web: WARN  
com.optimagrowth: DEBUG
```

Then build again and execute your microservices. If you are using docker you can execute the following commands in the root directory where the parent pom.xml is located.

```
mvn clean package dockerfile:build  
docker-compose -f docker/docker-compose.yml up
```

You'll see that once the call hits the resiliency protected method, we still get the values written out for the correlation ID, meaning that the parent thread values are available on the methods using the Resilience4j annotations.

Resilience4j is an excellent choice to implement a resilience pattern in our application. With Hystrix going into a maintenance mode, Resilience4j has become the number one choice in the Java ecosystem.

Now that we have seen what can be achievable with Resilience4j, we can move on with our next chapter, the Spring Cloud Gateway.

7.11 Summary

- When designing highly distributed applications such as a microservice-based application, client resiliency must be taken into account.
- Outright failures of a service (for example, the server crashes) are easy to detect and deal with.
- A single poorly performing service can trigger a cascading effect of resource exhaustion as threads in the calling

client are blocked waiting for a service to complete.

- Three core client resiliency patterns are the circuit-breaker pattern, the fallback pattern, and the bulkhead pattern.
- The circuit breaker pattern seeks to kill slow-running and degraded system calls so that the calls fail fast and prevent resource exhaustion.
- The fallback pattern allows you as the developer to define alternative code paths in the event that a remote service call fails or the circuit breaker for the call fails.
- The bulkhead pattern segregates remote resource calls away from each other, isolating calls to a remote service into their own thread pool. If one set of service calls is failing, its failures shouldn't be allowed to eat up all the resources in the application container.
- The rate limiter pattern limits the number of total calls in a given time.
- Resilience4j allows us to stack and use several patterns at the same time.
- The retry pattern is responsible for making attempts when a service has temporarily failed.
- The main difference between the bulkhead and the Rate limiter pattern is that bulkhead is in charge of limiting the number of concurrent calls at a time and the rate limiter limits the number of total calls in a given time.
- Spring Cloud and the Resilience4j libraries provide implementations for the circuit breaker, fallback, retry, rate limiter and bulkhead patterns.
- The Resilience4j libraries are highly configurable and can be set at global, class, and thread pool levels.

8 Service routing with Spring Cloud Gateway

This chapter covers

- Using a service gateway with your microservices
- Implementing a service gateway using Spring Cloud Gateway
- Mapping microservice routes in the gateway
- Building filters to use correlation ID and tracking

In a distributed architecture like a microservices one, there will come the point where we'll need to ensure that critical behaviors such as security, logging, and tracking of users across multiple service calls occur. To implement this functionality, we'll want these attributes to be consistently enforced across all of our services without the need for each individual development team to build their own solutions. While it's possible to use a common library or framework to assist with building these capabilities directly in an individual service, doing so has three implications.

First, it's challenging to implement these capabilities in each service consistently. Developers are focused on delivering functionality, and in the whirlwind of day-to-day activity, they can easily forget to implement service logging or tracking unless they work in a regulated industry where it's required.

Second, properly implementing these capabilities is a challenge. Things like microservice security can be a pain to set up and configure with each service being implemented. Pushing the responsibilities to implement a cross-cutting concern like security down to the individual development teams greatly increases the odds that someone will not implement it properly or will forget to do it. Cross-cutting concerns refer to parts or features of the design of a program that is applicable throughout the application, and may affect other parts of the application—for example, security and logging.

Third, we've now created a hard dependency across all our services. The more capabilities we build into a common framework shared across all our services, the more difficult it is to change or add behavior in our common code without having to recompile and redeploy all our services. Suddenly an upgrade of core capabilities built into a shared library becomes a months-long migration process.

To solve this problem, we need to abstract these cross-cutting concerns into a service that can sit independently and act as a filter and router for all the microservice calls in our architecture. This cross-cutting concern is called a services gateway. Our service clients no longer directly call a microservice. Instead, all calls are routed through the service gateway, which acts as a single Policy Enforcement Point (PEP), and are then routed to a final destination.

In this chapter, we're going to see how to use Spring Cloud Gateway to implement a services gateway. Specifically, we're going to look at how to use Spring Cloud Gateway to

- Put all service calls behind a single URL and map those calls using service discovery to their actual service instances
- Inject correlation IDs into every service call flowing through the service gateway
- Inject the correlation ID back from the HTTP response sent back to the client

Let's dive into more detail on how a services gateway fits into the overall microservices being built in this book.

8.1 What is a services gateway?

Until now, with the microservices we've built in earlier chapters, we've either directly called the individual services through a web client or called them programmatically via a service discovery engine such as Eureka as shown in the following Figure 8.1.

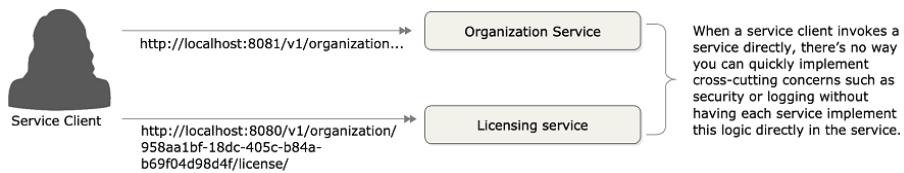


Figure 8.1 Without a services gateway, the service client will call distinct endpoints for each service.

A service gateway acts as an intermediary between the service client and a service being invoked. The service client talks only to a single URL managed by the service gateway. The service gateway pulls apart the path coming in from the service client call and determines what service the service client is trying to invoke. Figure 8.2 illustrates how like a "traffic cop" directing traffic, the service gateway directs the user to a target microservice and corresponding instance.

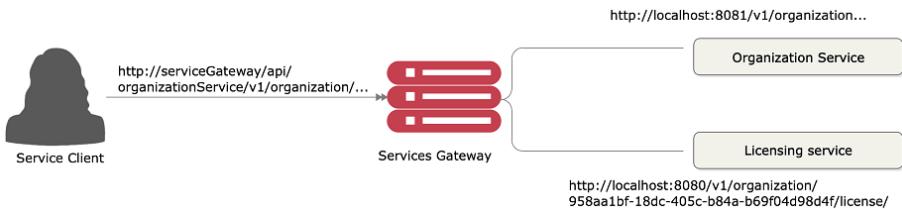


Figure 8.2 The service gateway sits between the service client and the corresponding service instances. All service calls (both internal-facing and external) should flow through the service gateway.

The service gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application. With a service gateway in place, our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway.

Because a service gateway sits between all calls from the client to the individual services, it also acts as a central Policy Enforcement Point (PEP) for service calls. The use of a centralized PEP means that cross-cutting service concerns can be implemented in a single place without the individual development teams having to implement these concerns. Examples of cross-cutting concerns that can be implemented in a service gateway include

- **Static routing.** A service gateway places all service calls behind a single URL and API route. This simplifies development as developers only have to know about one service endpoint for all of their services.
- **Dynamic routing.** A service gateway can inspect incoming service requests and, based on data from the incoming request, perform intelligent routing based on who the service caller is. For instance, customers participating in a beta program might have all calls to a service routed to a specific cluster of services that are running a different version of code from what everyone else is using.
- **Authentication and authorization.** Because all service calls route through a service gateway, the service gateway is a natural place to check whether the caller of a service has authenticated themselves.
- **Metric collection and logging.** A service gateway can be used to collect metrics and log information as a service call passes through the service gateway. You can also use the service gateway to ensure that critical pieces of information are in place on the user request to ensure logging is uniform. This doesn't mean that you still shouldn't collect metrics from within your individual services, but rather a services gateway allows you to centralize the collection of many of your basic metrics, like the number of times the service is invoked and service response time.

Wait—isn't a service gateway a single point of failure and potential bottleneck?

Earlier in chapter 6, when I introduced Eureka, I talked about how centralized load balancers can be a single point of failure and a bottleneck for your services. A service gateway, if not implemented correctly, can carry the same risk. Keep the following in mind as you build your service gateway implementation.

Load balancers are still useful when out in front of individual groups of services. In this case, a load balancer sitting in front of multiple service gateway instances is an appropriate design and ensures your service gateway implementation can scale. Having a load balancer sitting in front of all your service instances isn't a good idea because it becomes a bottleneck.

Keep any code you write for your service gateway stateless. Don't store any information in memory for the service gateway. If you aren't careful, you can limit the scalability of the gateway and have to ensure that the data gets replicated across all service gateway instances.

Keep the code you write for your service gateway light. The service gateway is the "chokepoint" for your service invocation. Complex code with multiple database calls can be the source of difficult-to-track-down performance problems in the service gateway.

Let's now look at how to implement a service gateway using Spring Cloud Gateway.

NOTE In this book, I use Spring Cloud Gateway, because it is the current preferred API gateway implementation from the Spring Cloud team. This implementation is built on Spring 5 and is a non-blocking gateway that integrates much easier with the other Spring cloud projects we are going to be using throughout the book.

8.2 Introducing Spring Cloud Gateway

Spring Cloud Gateway is the API Gateway implementation by Spring Cloud Built on Spring Framework 5, Project Reactor and Spring Boot 2.0. This gateway is a non-blocking gateway. What means non-blocking? Remember, non-blocking applications are written in a way main threads are never blocked; instead, they are always available to serve requests and to process them asynchronously in the background to return the response once processing is done.

The Spring Cloud Gateway offers several capabilities, including

- **Mapping the routes for all the services in your application to a single URL.** The Spring Cloud Gateway isn't limited to a single URL. Actually, with it, we can define multiple route entries, making the route mapping extremely fine-grained (each service endpoint gets its own route mapping). However, the first and most common use case is to build a single-entry point through which all service client calls will flow.
- **Building filters that can inspect and act on the requests and responses coming through the gateway.** These filters allow us to inject policy enforcement points in our code and perform a wide number of actions on all of our service calls in a consistent fashion. In other words, they can allow us to modify the incoming and outgoing HTTP requests and responses.
- **Building predicates.** The Spring Cloud Gateway includes a set of build-in Route Predicate factories. These predicates are objects that allow us to check if the requests fulfill a or a set of given conditions before executing or processing a request.

To get started with Spring Cloud Gateway, we're going to do two things:

1. Set up a Spring Cloud Gateway Spring Boot project and configure the appropriate Maven dependencies.
2. Configure the gateway to communicate with Eureka.

8.2.1 Setting up the Gateway Spring Boot project

In this section, we'll set up our Spring Cloud Gateway service using Spring Boot. Like the Spring Cloud configuration service and the Eureka Service that we already created in previous chapters, setting up a Spring Cloud Gateway Service starts with building a new Spring Boot project and applying annotations and configurations. Let's begin by creating that new project with the Spring Initializr (<https://start.spring.io/>), as shown on figure 8.3



Figure 8.3 Spring Initializr with our Spring Cloud Gateway information.

To achieve this let's follow the next steps

1. Select maven as the project type.
2. Select Java as the language
3. Select the latest or more stable 2.x.x spring version.
4. Write com.optimagrowth as group and gatewayserver as artifact.
5. Write API Gateway Server as name, API Gateway Server as description and com.optimagrowth.gateway as package name.
6. Select the JAR packaging.
7. Select Java 11 as the java version.
8. Add the Eureka Client, Config Client, Gateway, and Spring Boot Actuator dependencies as shown in figure 8.4

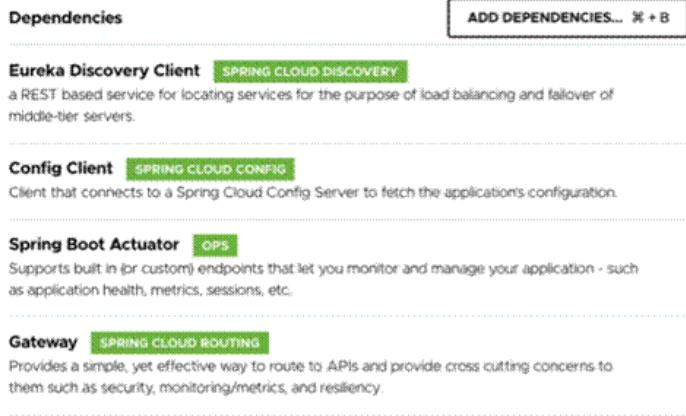


Figure 8.4 Gateway server dependencies in Spring Initializr

The following code listing 8.1 shows how the Gateway server pom.xml file should look like.

Listing 8.1 Maven pom file for the Gateway Server

```
Rest of pom.xml removed for conciseness
...
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId> #A
<artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
<exclusions>
<exclusion>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-ribbon</artifactId>
</exclusion>
<exclusion>
<groupId>com.netflix.ribbon</groupId>
<artifactId>ribbon-eureka</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
Rest of pom.xml removed for conciseness
...
</project>
```

#A Tells your maven build to include the Gateway libraries.

The next step is to set up the src/main/resources/bootstrap.yml file with the configuration needed to retrieve the configuration from the Spring Config Server that we previously created on chapter 5. Code listing 8.2 will show your bootstrap.yml file should look like.

Listing 8.2 Setting up the Gateway bootstrap.yml file

```
spring:  
application:  
name: gateway-server #A  
cloud:  
config:  
uri: http://localhost:8071 #B
```

#A Specify the name of the Gateway service so that Spring Cloud Config client knows which service is being looked up.

#B Specify the location of the Spring Cloud Config server.

NOTE In case you didn't follow the previous chapters code listings, you can download the code created in chapter 7 from the following link: <https://github.com/ihuaylupo/manning-smia/tree/master/chapter7>.

8.2.2 Configuring the Spring Cloud Gateway to communicate with Eureka

The Spring Cloud Gateway can integrate with the Netflix Eureka service discovery we created in chapter 6. To achieve this integration, we must add the Eureka configuration in the configuration server for the Gateway service we've just created. This may sound somehow complicated, but don't worry, it is something that we already achieved for the previous chapter. Let's follow the next three steps to add the new Gateway configuration.

The first step is to create a new configuration file for the Gateway service in the repository of the Spring Configuration Server. Remember, this could be Vault, Git, or Filesystem/classpath. For purposes of this example, I've created the gateway-server.yml file in the classpath of the project under the following path: /configserver/src/main/resources/config/gateway-server.yml

NOTE Remember, the name of the file goes by the hand of the spring.application.name property file you defined in the bootstrap.yml of the service. For example, for the gateway service, we defined the spring.application.name to be gateway-server, so the configuration file must be named gateway-server as well. As for the extension, you can choose between .properties or .yml.

The second step will be adding the Eureka configuration data into the just created configuration file. The following code listing 8.3 will show you how.

Listing 8.3 Setting up the Eureka configuration in the Spring Configuration Server

```
server:  
port: 8072  
eureka:  
instance:  
preferIpAddress: true  
client:  
registerWithEureka: true  
fetchRegistry: true  
serviceUrl:  
defaultZone: http://eurekaserver:8070/eureka/
```

The third and final step is adding the @EnableEurekaClient in the /gatewayserver/src/main/java/com/optimagrowth/gateway/ApiGatewayServerApplication.java class as show in code listing 8.4

Listing 8.4 Adding the @EnableEurekaClient in the ApiGatewayServerApplication class

```
package com.optimagrowth.gateway;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
@SpringBootApplication  
@EnableEurekaClient  
public class ApiGatewayServerApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(ApiGatewayServerApplication.class, args);  
    }  
}
```

Now, that we've created the basic configuration to our Spring Cloud Gateway let's start routing our services.

8.3 Configuring routes in Spring Cloud Gateway

The Spring Cloud Gateway at its heart is a reverse proxy. A reverse proxy is an intermediate server that sits between the client trying to reach a resource and the resource itself. The client has no idea it's even communicating to a server other than a proxy. The reverse proxy takes care of capturing the client's request and then calls the remote resource on the client's behalf.

In the case of a microservices architecture, the Spring Cloud Gateway (our reverse proxy) takes a microservice call from a client and forwards it onto the upstream service. The service client thinks it's only communicating with the gateway. But it is not as simple as that actually for the gateway to communicate with the upstream

services has to know how to map the incoming call to an upstream route. The Spring Cloud Gateway has several mechanisms to do this, including

- Automated mapping of routes via service discovery
- Manual mapping of routes using service discovery

8.3.1 Automated mapping of routes via service discovery

All route mappings for the gateway are done by defining the routes in the /configserver/src/main/resources/config/gateway-server.yml file. However, the Spring Cloud Gateway can automatically route requests based on their service IDs by adding the configuration lines shown in code listing 8.5 in the gateway-server configuration file.

Listing 8.5 Setting up the discovery locator in the gateway-server.yml file.

```
spring:  
cloud:  
gateway:  
discovery.locator: #A  
enabled: true  
lowerCaseServiceId: true
```

#A Enables the Gateway to create routes based on services registered on the service discovery.

By adding the previous lines, the Spring Cloud Gateway will automatically use the Eureka service ID of the service being called and map it to a downstream service instance. For instance, if we wanted to call our organization-service and used automated routing via the Spring Cloud Gateway, we would have our client call the gateway service instance, using the following URL as the endpoint:

```
http://localhost:8072/organization-service/v1/organization/958aa1bf-18dc-405c-b84a-b69f04d98d4f
```

The Gateway server is accessed via <http://localhost:8072>. The service we're trying (organization-service) to invoke is represented by the first part of the endpoint path in the service. Figure 8.5 illustrates this mapping in action.

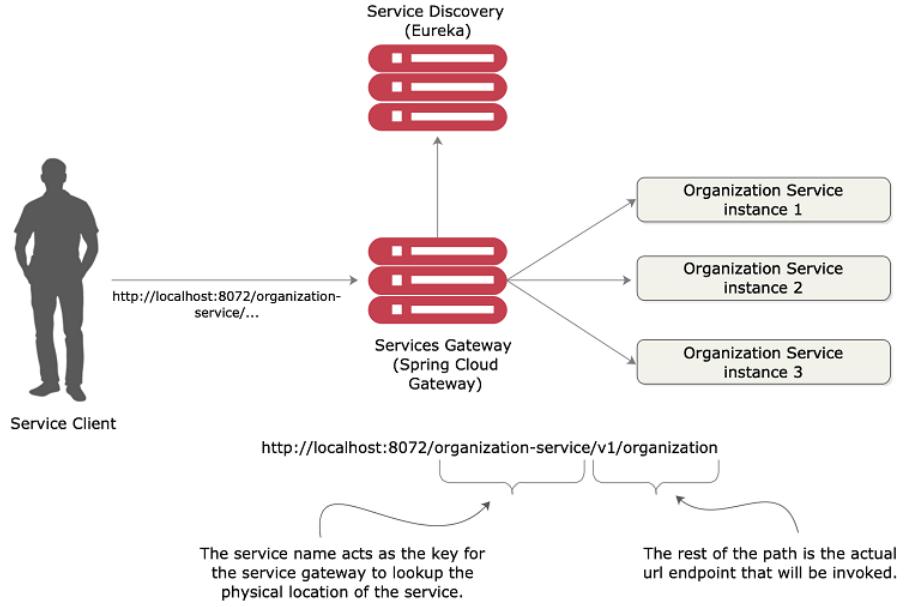


Figure 8.5 The Spring Cloud Gateway will use the organization-service application name to map requests to organization service instances.

The beauty of using Spring Cloud Gateway with Eureka is that not only do we now have a single endpoint that we can make calls through but with Eureka, we can also add and remove instances of service without ever having to modify the gateway. For instance, we can add a new service to Eureka, and the gateway will automatically route to it because it's communicating with Eureka about where the actual physical services endpoints are located.

If we want to see the routes being managed by the gateway server, we can list the routes via the `actuator/gateway/routes` endpoint on the gateway server. This will return a listing of all the mappings on our service. Figure 8.6 shows the output from hitting <http://localhost:8072/actuator/gateway/routes>.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
[{"id": "1", "predicates": [{"path": "/licensing-service/**"}], "filters": [{"filter": "[[RewritePath /licensing-service/(?<remaining>.*)) = /${remaining}'], order = 1]"}], "routeId": "ReactiveCompositeDiscoveryClient_LICENSING-SERVICE", "uri": "lb://LICENSING-SERVICE", "order": 0}, {"id": "2", "predicates": [{"path": "/gateway-server/**"}], "filters": [{"filter": "[[RewritePath /gateway-server/(?<remaining>.*)) = /${remaining}'], order = 1]"}], "routeId": "ReactiveCompositeDiscoveryClient_GATEWAY-SERVER", "uri": "lb://GATEWAY-SERVER", "order": 0}, {"id": "3", "predicates": [{"path": "/organization-service/**"}], "filters": [{"filter": "[[RewritePath /organization-service/(?<remaining>.*)) = /${remaining}'], order = 1]"}], "routeId": "ReactiveCompositeDiscoveryClient_ORGANIZATION-SERVICE", "uri": "lb://ORGANIZATION-SERVICE", "order": 0}]
  
```

Figure 8.6 Each service that is mapped in Eureka will now be mapped as a Spring Cloud Gateway route.

In figure 8.6 the mappings for the services registered with the Spring Cloud Gateway are shown with additional data such as predicate, management port, route id, filters, and more.

8.3.2 Mapping routes manually using service discovery

Spring Cloud Gateway allows us to be more fine-grained by allowing us to explicitly define route mappings rather than relying solely on the automated routes created with the service's Eureka service ID. Suppose we wanted to simplify the route by shortening the organization name rather than having your organization service accessed in the gateway via the default route of /organization-service/v1/organization/{organization- id}. You can do this by manually defining the route mapping in /configserver/src/main/resources/config/gateway-server.yml

located in the Spring Cloud Configuration server repository. The following code listing 8.6 shows you how.

Listing 8.6 Mapping routes manually in the gateway-server.yml file

```
spring:
cloud:
gateway:
discovery.locator:
enabled: true
lowerCaseServiceId: true
routes:
- id: organization-service #A
uri: lb://organization-service #B
predicates: #C
- Path=/organization/** #D
filters: #D
- RewritePath=/organization/(?<path>.*), /$\\{path} #E
```

#A The ID is an arbitrary ID given to the route. (Optional).

#B The URI is the destination URI of the route.

#C The predicates set a collection of predicates used to match various pieces of the incoming HTTP request.

#D Filters set a collection of Spring WebFilters to modify the request or response before or after sending the response.

#E The RewritePath rewrites the request path by taking as parameter path regexp and a replacement order. Rewrites from /organization/** to /**

By adding this configuration, we can now access the organization service by hitting the /organization/v1/organization/{organization-id} route. If we recheck the gateway server's endpoint, we should see the results shown in figure 8.7.

```

[
{
  "predicate": "Paths: [/licensing-service/**], match trailing slash: true",
  "metadata": {
    "management.port": "8080"
  },
  "route_id": "ReactiveCompositeDiscoveryClient_LICENSING-SERVICE",
  "filters": [
    "[[RewritePath /licensing-service/(?<remaining>.*)) = '/${remaining}'], order = 1]"
  ],
  "uri": "lb://LICENSING-SERVICE",
  "order": 0
},
{
  "predicate": "Paths: [/organization-service/**], match trailing slash: true",
  "metadata": {
    "management.port": "8081"
  },
  "route_id": "ReactiveCompositeDiscoveryClient_ORGANIZATION-SERVICE",
  "filters": [
    "[[RewritePath /organization-service/(?<remaining>.*)) = '/${remaining}'], order = 1]"
  ],
  "uri": "lb://ORGANIZATION-SERVICE",
  "order": 0
},
{
  "predicate": "Paths: [/gateway-server/**], match trailing slash: true",
  "metadata": {
    "management.port": "8072"
  },
  "route_id": "ReactiveCompositeDiscoveryClient_GATEWAY-SERVER",
  "filters": [
    "[[RewritePath /gateway-server/(?<remaining>.*)) = '/${remaining}'], order = 1]"
  ],
  "uri": "lb://GATEWAY-SERVER",
  "order": 0
},
{
  "predicate": "Paths: [/organization/**], match trailing slash: true",
  "route_id": "organization-service",
  "filters": [
    "[[RewritePath /organization/(?<path>.*)) = '/${path}'], order = 1]"
  ],
  "uri": "lb://organization-service",
  "order": 0
}
]

```

We still have the Eureka service ID-based route here.

Notice the custom route for the organization service.

Figure 8.7 The results of the gateway /actuator/gateway/routes call with a manual mapping of the organization service.

If you look carefully at figure 8.7, you'll notice that two entries are present for the organization service. One service entry is the mapping we defined in the gateway-server.yml file: “organization/**”: “organization-service.” The other service entry is the automatic mapping created by the gateway based on the organization service's Eureka ID: “/organization-service/**”: “organization-service.”

NOTE When we use automated route mapping where the gateway exposes the service based solely on the Eureka service ID, if no instances of the service are running, the gateway will not expose the route for the service. However, if we manually map a route to a service discovery ID and there are no instances registered with Eureka, the gateway will still show the route. If we try to call the route for the non-existent service, it will return a 500 error.

If we want to exclude the automated mapping of the Eureka service ID route and only have available the organization service route that we've defined, we can

remove the `spring.cloud.gateway.discovery.locator` entries we added in the `gateway-server.yml` file, as shown in the following code listing 8.7.

NOTE The decision whether to use automated routing or not should be taken carefully. In a stable environment, where not many new services are added, having to add the route manually is a straightforward task. However, in a large environment with many new services, this is a bit tedious.

Listing 8.7 Removing the discovery locator entries in the gateway-server.yml file

```
spring:  
cloud:  
gateway:  
routes:  
- id: organization-service  
uri: lb://organization-service  
predicates:  
- Path=/organization/**  
filters:  
- RewritePath=/organization/(?<path>.*), /${path}
```

Now, when we call the `actuator/gateway/routes` endpoint on the gateway server, we should only see the organization service mapping we've defined. Figure 8.8 shows the outcome of this mapping.

The screenshot shows a JSON response from a REST endpoint. The URL in the address bar is `http://localhost:8072/actuator/gateway/routes`. The response is a single-element array containing a route configuration. The JSON is formatted with line numbers:

```
1 [  
2   {  
3     "predicate": "Paths: [/organization/**], match trailing slash: true",  
4     "route_id": "organization-service",  
5     "filters": [  
6       "[[RewritePath /organization/(?<path>.*), /${path}]]",  
7     ],  
8     "uri": "lb://organization-service",  
9     "order": 0  
10   }  
11 ]
```

Figure 8.8 The results of the gateway /actuator/gateway/routes call with only a manual mapping of the organization service.

8.3.3 Dynamically reload route configuration

The next thing we're going to look at in terms of configuring routes in Spring Cloud Gateway is how to dynamically refresh routes. The ability to dynamically reload routes is useful because it allows us to change the mapping of routes without having to restart the gateway server(s). Existing routes can be modified quickly,

and new routes added within have to go through the act of recycling each gateway server in our environment.

If you hit the actuator/gateway/routes endpoint, we should see our organization service currently shown in the gateway. Now, if we wanted to add new route mappings on the fly, all we have to do is make the changes to the config file and then commit them back to the Git repository where Spring Cloud Config is pulling its configuration data from.

Then we can commit the changes to GitHub. Spring Actuator exposes a POST-based endpoint route actuator/gateway/refresh that will cause it to reload its route configuration. Once this actuator/gateway/refresh is hit, if you then hit the /routes endpoint, you'll see that the two new routes are exposed. The response of the actuator/gateway/refresh returns a 200 without response body.

8.4 The real power of the Spring Cloud Gateway: predicate and filters factories

While being able to proxy all requests through the gateway does allow us to simplify our service invocations, the real power of the gateway comes into play when we want to write custom logic that will be applied against all the service calls flowing through the gateway. Most often, this custom logic is used to enforce a consistent set of application policies like security, logging, and tracking against all the services.

These application policies are considered cross-cutting concerns because we want them to be applied to all the services in our application without having to modify each service to implement them. In this fashion, the Spring Cloud Gateway predicate and filter factories can be used in a similar way as Spring Aspect that can match or intercept a wide body of behaviors and decorate or change the behavior of the call without the original coder being aware of the change. While a servlet filter or Spring Aspect is localized to a specific service, using the gateway, its predicates and filters allows us to implement cross-cutting concerns across all the services being routed through the gateway. Remember, predicates allow us to check if the requests fulfill a set of conditions before processing the request.

The following figure 8.9 shows the architecture that the Spring Cloud Gateway uses to apply the predicates and filters when a request comes to the gateway.

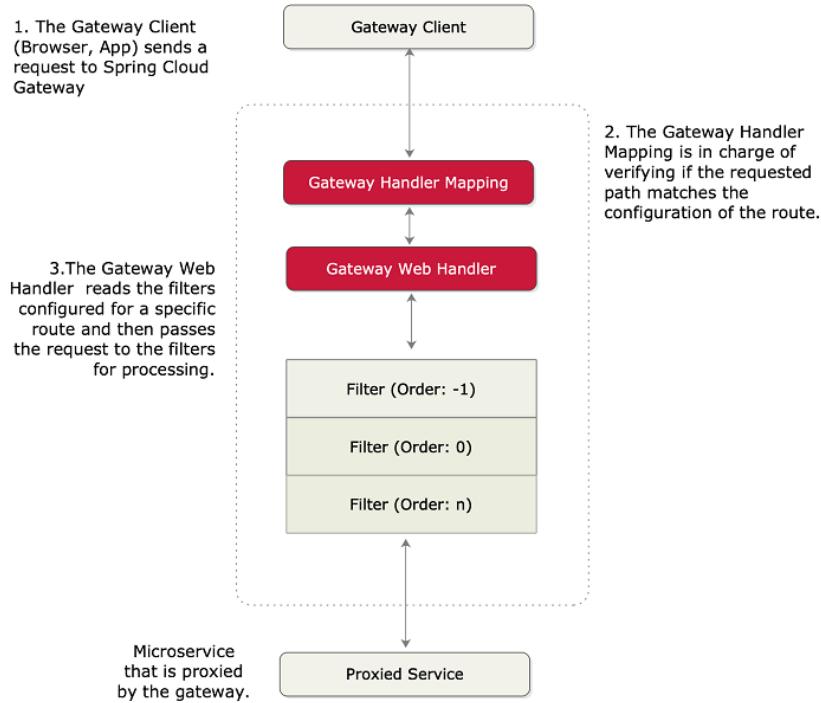


Figure 8.9 Spring Cloud Gateway architecture to apply the predicates and filters when a request is made.

The first step is when the gateway client (browsers, apps, ...) sends a request to our Spring Cloud Gateway. Once that request is received, it goes directly to the Gateway Handler that is in charge of verifying that the requested path matches the configuration of the specific route it is trying to access. If everything matches, it enters the Gateway Web Handler that is in charge of reading the filters and send the request to those filters for further processing. Once the request passes all the filters, it forwards it to the routing configuration (a microservice).

8.4.1 Built-in Predicates Factories

These predicates are objects that allow us to check if the requests fulfill a set of given conditions before executing or processing a request. We can set multiple route predicate factories, and they are used and combined via the logical "and".

These predicates can be applied in the code in a programmatically way or via the configuration like the ones we previously created. For purposes of this book, I will only use them via the configuration file under the predicates section.

```
predicates:
- Path=/organization/**
```

The following table 8.1 lists all the built-in predicate factories in the Spring Cloud Gateway.

Table 8.1 Built-in predicate factories in the Spring Cloud Gateway

Predicate	Description	Example
Before Route	This predicate takes a date-time parameter and matches all the requests that happen before it.	Before=2020-03-11T...
After Route	This predicate takes a date-time parameter and matches all the requests that happen after it.	After=2020-03-11T...
Between Route	This predicate takes two date-time parameters and matches all the requests between them. The first date-time is inclusive, and the second one is exclusive.	Between=2020-03-11T..., 2020-04-11T...
Header Route	This predicate receives two parameters the name of the header and a regular expression and then matches its value with the provided regular expression.	Header=X-Request-Id, \d+
Host Route	This predicate receives an Ant-style pattern separated with "." hostname pattern as a parameter. Then it matches the Host header with the given the pattern.	Host=**.example.com
Method Route	This predicate receives the HTTP method to match.	Method=GET
Path Route	This predicate receives a Spring PathMatcher.	Path=/organization/{id}
Query Route	This predicate receives two parameters: Required parameter and an optional regular expression to match them with the query parameters.	Query=id, 1
Cookie Route	This predicate takes two parameters a name for the cookie and a regular expression and finds the cookies in the HTTP request	Cookie=SessionID, abc.

	header and matches its value with the provided regular expression.	
RemoteAddrRoute	This predicate receives a list of IP addresses and matches them with the remote address of a request.	RemoteAddr=192.168.3.5/24

8.4.2 Built-in Filters Factories

These filters allow us to inject policy enforcement points in our code and perform a wide number of actions on all of our service calls in a consistent fashion. In other words, they can allow us to modify the incoming and outgoing HTTP requests and responses. The following table 8.2 contains a list of all the built-in filters in the Spring Cloud Gateway.

Table 8.1 Built-in filters factories in the Spring Cloud Gateway

Predicate	Description	Example
AddRequestHeader	Adds a HTTP Request header with the name and the value received as parameters.	AddRequestHeader=X-Organization-ID, F39s2
AddResponseHeader	Adds a HTTP Response header with the name and the value received as parameters.	AddResponseHeader=X- Organization-ID, F39s2
AddRequestParameter	Adds a HTTP Query Parameter with the name and the value received as parameters.	AddRequestParameter=Organizationid, F39s2
PrefixPath	Adds a prefix to the HTTP request path.	PrefixPath=/api
RequestRateLimiter	Receives three parameters. The	RequestRateLimiter=10, 20, #{@userKeyResolver}

	<p>first one is the <code>replenishRate</code> that represents the requests per seconds we want to allow the user to do, the second one is the <code>capacity</code> which defines how much bursting capacity would be allowed, and the third one is the <code>keyResolverName</code> <i>that defines the name of a bean that implements the Key Resolver interface.</i></p>	
<code>RedirectTo</code>	Takes two parameters a status and URL. The status should be a 300 redirect HTTP code.	<code>RedirectTo=302, http://localhost:8072</code>
<code>RemoveNonProxyHeaders</code>	It removes some headers such as <code>Keep-Alive</code> , <code>Proxy-Authenticate</code> , or <code>Proxy-Authorization</code>	
<code>RemoveRequestHeader</code>	It removes a header that matches the name received as parameter from the HTTP Request.	<code>RemoveRequestHeader=X-Request-Foo</code>

RemoveResponseHeader	It removes a header that matches the name received as parameter from the HTTP Response.	RemoveResponseHeader=X-Organization-ID
RewritePath	Takes a path regexp parameter and a replacement parameter.	RewritePath=/organization/(?<path>.*), /\${path}
SecureHeaders	It adds some secure headers to the response It receives a Path template parameter and changes the request path.	-
SetPath		SetPath=/organization
SetStatus	Receives a valid HTTP status and changes the status of a HTTP response	SetStatus=500
SetResponseHeader	It takes name and value parameters to set a header on the HTTP response	SetResponseHeader=X-Response-ID, 123

8.4.3 Custom Filters

While being able to proxy all requests through the gateway does allow us to simplify our service invocations. The real power of Spring Cloud Gateway comes into play when we want to write custom logic that will be applied against all the

service calls flowing through the gateway. Most often, this custom logic is used to enforce a consistent set of application policies like security, logging, and tracking against all the services.

The Spring Cloud Gateway allows us to build custom logic using a filter within the gateway. Remember, a filter allows us to implement a chain of business logic that each service request passes through as it's being implemented.

Spring Cloud Gateway supports two types of filters:

1. **Pre-filters.** A pre-filter is invoked before the actual request is sent to the target destination. A pre-filter usually carries out the task of making sure that the service has a consistent message format (key HTTP headers are in place, for example) or acts as a gatekeeper to ensure that the user calling the service is authenticated (they are whom they say they are).
2. **Post-filters.** A post-filter is invoked after the target service has been invoked, and a response is being sent back to the client. Usually, a post-filter will be implemented to log the response back from the target service, handle errors, or audit the response for sensitive information.

Figure 8.10 shows how the pre- and post-filter fit together in terms of processing a service client's request.

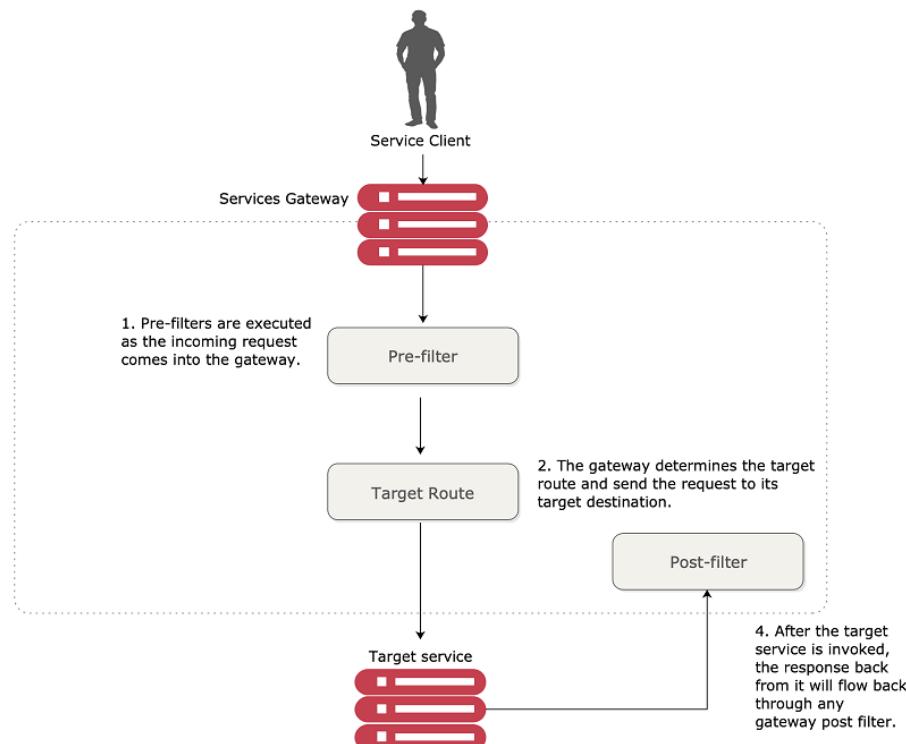


Figure 8.10 The pre-, route, and post-filters form a pipeline in which a client request flows through. As a request comes into the gateway, these filters can manipulate the incoming request.

If we follow the flow laid out in figure 8.11, we'll see everything start with a service client making a call to a service exposed through the service gateway. From there, the following activities take place:

1. Any pre-filters defined in the gateway will be invoked as a request enters the gateway. The pre-filters can inspect and modify a HTTP request before it gets to the actual service. A pre-filter cannot redirect the user to a different endpoint or service.
2. After the pre-filters are executed against the incoming request by the gateway, the gateway will determine the destination of where the service is heading.
3. After the target service has been invoked, the gateway Post filters will be invoked. A post-filter can inspect and modify the response back from the invoked service.

The best way to understand how to implement the gateway filters is to see them in use. To this end, in the next several sections, we'll build a pre- and post-filter and then run service client requests through them.

Figure 8.11 shows how these filters will fit together in processing requests to our O-stock services.

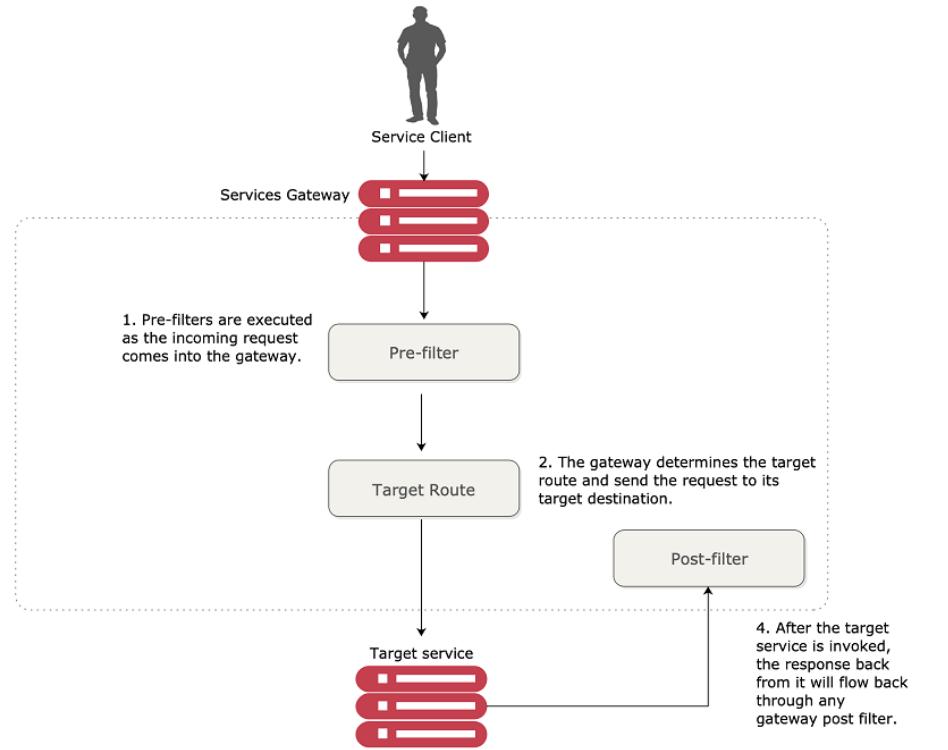


Figure 8.11 The gateway filters provide centralized tracking of service calls, logging. Gateway filters allow us to enforce custom rules and policies against microservice calls.

Following the flow of figure 8.11, we'll see the following filters being used:

- 1. Tracking Filter.** The TrackingFilter will be a pre-filter that will ensure that every request flowing from the gateway has a correlation ID associated with it. A correlation ID is a unique ID that gets carried across all the microservices that are executed when carrying out a customer request. A correlation ID allows us to trace the chain of events that occur as a call goes through a series of microservice calls.
- 2. Target Service.** The target service could either be the organization or the licensing service. Both services are going to receive the Correlation ID in the HTTP Request Header.
- 3. Response Filter.** The ResponseFilter is a post-filter that will inject the correlation ID associated with the service call into the HTTP response header being sent back to the client. This way, the client will have access to the correlation ID associated with the request they made.

8.5 Building the pre-filter

Building filters in the Spring Cloud Gateway is a straightforward activity. To begin, we'll build a pre-filter, called the TrackingFilter, that will inspect all incoming requests to the gateway and determine whether there's an HTTP header called tmx-correlation-id present in the request. The tmx-correlation-id header will contain a unique GUID (Globally Universal Id) that can be used to track a user's request across multiple microservices.

NOTE We discussed the concept of a correlation ID in chapter 7. Here we're going to walk through in more detail how to use the gateway to generate a correlation ID. If you skipped around in the book, I highly recommend you look at chapter 7 and read the section on Thread context. Your implementation of correlation IDs will be implemented using ThreadLocal variables, and there's extra work to do to have ThreadLocal variables work.

If the tmx-correlation-id isn't present on the HTTP header, our gateway TrackingFilter will generate and set the correlation ID. If there's already a correlation ID present, the gateway won't do anything with the correlation ID. The presence of a correlation ID means that this particular service call is part of a chain of service calls carrying out the user's request. In this case, our TrackingFilter class will do nothing.

Let's go ahead and look at the implementation of the TrackingFilter in the following code listing 8.8. This code can also be found in the book samples in /gatewayserver/src/main/java/com/optimagrowth/gateway/filters/TrackingFilter.java

Listing 8.8 Pre-filter for generating correlation IDs

```
package com.optimagrowth.gateway.filters;
//Removed other imports for conciseness
import org.springframework.http.HttpHeaders;
import reactor.core.publisher.Mono;
@Order(1)
@Component
public class TrackingFilter implements GlobalFilter { #A
private static final Logger logger = LoggerFactory.getLogger(TrackingFilter.class);
@Autowired
FilterUtils filterUtils; #B
@Override
public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) { #C
HttpHeaders requestHeaders = exchange.getRequest().getHeaders(); #D
if (isCorrelationIdPresent(requestHeaders)) {
logger.debug("tmx-correlation-id found in tracking filter: {}.", " " ,
filterUtils.getCorrelationId(requestHeaders));
} else {
String correlationID = generateCorrelationId();
exchange = filterUtils.setCorrelationId(exchange, correlationID);
logger.debug("tmx-correlation-id generated in tracking filter: {}.", " " ,
correlationID);
}
return chain.filter(exchange);
} #E
private boolean isCorrelationIdPresent(HttpHeaders requestHeaders) {
if (filterUtils.getCorrelationId(requestHeaders) != null) {
return true;
} else {
return false;
}
} #F
private String generateCorrelationId() {
return java.util.UUID.randomUUID().toString();
}
}
```

```

#A All global filters must implement the GlobalFilter interface and override the filter() method.
#B Commonly used functions that are used across all your filters have been encapsulated in the FilterUtils
    class.
#C The filter() method is the code that is executed every time a request passes through the filter.
#D Extract the HTTP headers from the request using the ServerWebExchange object passed by parameters to
    the filter() method.
#E The helper method that checks if there is a correlation ID in the request header.
#F The helper methods that actually check if the tmx-correlation-id is present and can also generate a
    correlation ID UUID value.

```

To implement a global filter in the Spring Cloud Gateway, we have to implement the GlobalFilter class and then override the following method: filter(). This method contains the business logic the filter is going to implement. Another critical point to highlight from the previous code is the way we obtain the HTTP Headers from the ServerWebExchange object. HttpHeaders requestHeaders = exchange.getRequest().getHeaders();

I've implemented a class called FilterUtils. This class is used to encapsulate common functionality used by all the filters. The FilterUtils class is located in the /gatewayserver/src/main/java/com/optimagrowth/gateway/filters/FilterUtils.java. We're not going to walk through the entire FilterUtils class, but the key methods we'll discuss here are the getCorrelationId(), and setCorrelationId() functions.

The following code listing 8.9 shows the code for the FilterUtils getCorrelationId() method.

Listing 8.9 Retrieving the tmx-correlation-id from the HTTP headers

```

public String getCorrelationId(HttpHeaders requestHeaders){
    if (requestHeaders.get(CORRELATION_ID) !=null) {
        List<String> header = requestHeaders.get(CORRELATION_ID);
        return header.stream().findFirst().get();
    }
    else{
        return null;
    }
}

```

The key thing to notice in listing 8.9 is that we first check to see if the tmx-correlation-ID is already set on the HTTP Headers for the incoming request. If it isn't there, we are just going to return null to create one later on. You may remember that earlier in the filter() method on our TrackingFilter class, we did exactly this with the following code snippet:

```

} else {
    String correlationID = generateCorrelationId();
    exchange = filterUtils.setCorrelationId(exchange, correlationID);
    logger.debug("tmx-correlation-id generated in tracking filter: {}.", correlationID);
}

```

The setting of the tmx-correlation-id occurs with the FilterUtils setCorrelationId() method as shown in the following code listing 8.10:

Listing 8.10 Set the tmx-correlation-id in the HTTP headers

```
public ServerWebExchange setRequestHeader(ServerWebExchange exchange, String name, String value) {  
    return exchange.mutate().request(  
        exchange.getRequest().mutate()  
            .header(name, value)  
            .build()  
            .build());  
}  
public ServerWebExchange setCorrelationId(ServerWebExchange exchange, String correlationId) {  
    return this.setRequestHeader(exchange, CORRELATION_ID, correlationId);  
}
```

In the FilterUtils setCorrelationId() method, when we want to add a value to the HTTP request headers, we use the Server Web Exchange's mutate() method. This method returns a builder to mutate properties of the exchange object by wrapping it with ServerWebExchangeDecorator and returning either mutated values or delegating back to this instance.

To test this call you can make a call to any organization or license service. Once the call is submitted, we should see a log message in the console writing out the passed in correlation ID as it flows through the filter.

```
gatewayserver_1 | 2020-04-14 22:31:23.835 DEBUG 1 --- [or-http-epoll-3] c.o.gateway.filters.TrackingFilter :  
tmx-correlation-id generated in tracking filter: 735d8a31-b4d1-4c13-816d-c31db20afb6a.
```

If you don't see the message on your console, just add the following code lines shown in code listing 8.11 on the bootstrap.yml configuration file of the gateway server.

Listing 8.11 Logger configuration on the gateway service bootstrap.yml file

```
//Some code removed for conciseness  
logging:  
level:  
com.netflix: WARN  
org.springframework.web: WARN  
com.optimagrowth: DEBUG
```

Then build again and execute your microservices. If you are using docker you can execute the following commands in the root directory where the parent pom.xml is located.

```
mvn clean package dockerfile:build  
docker-compose -f docker/docker-compose.yml up
```

8.5.1 Using the correlation ID in the services

Now that we've guaranteed that a correlation ID has been added to every microservice call flowing through the gateway, how do we ensure that

- The correlation-ID is readily accessible to the microservice that's being invoked.
- Any downstream service calls the microservice might make also propagate the correlation-ID on to the downstream call

To implement this, we're going to build a set of three classes into each of our microservices. These classes will work together to read the correlation ID (along with other information we'll add later) of the incoming HTTP request, map it to a class that's easily accessible and useable by the business logic in the application, and then ensure that the correlation ID is propagated to any downstream service calls.

Figure 8.12 demonstrates how these different pieces are going to be built out using our licensing service.

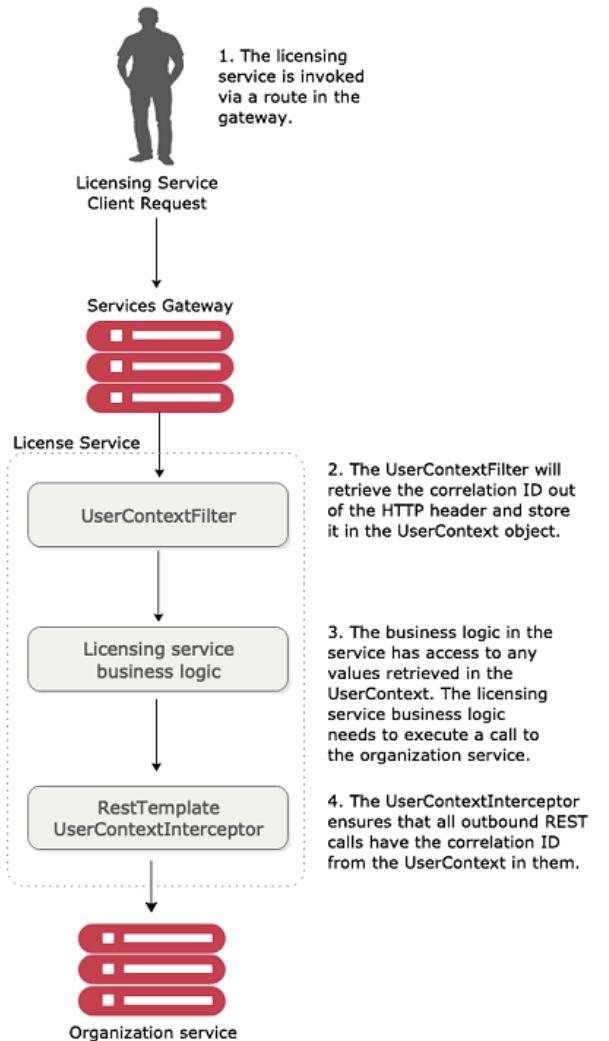


Figure 8.12 A set of common classes are used so that the correlation ID can be propagated to downstream service calls.

Let's walk through what's happening in figure 8.12:

1. When a call is made to the licensing service through the gateway, the TrackingFilter will inject a correlation ID into the incoming HTTP header for any calls coming into the gateway.
2. The UserContextFilter class is a custom HTTP ServletFilter. It maps a correlation ID to the UserContext class. The UserContext class is stored values in thread-local storage for use later in the call.
3. The licensing service business logic needs to execute a call to the organization service.

4. A RestTemplate is used to invoke the organization service. The RestTemplate will use a custom Spring Interceptor class UserContextInterceptor to inject the correlation ID into the outbound call as an HTTP header.

Repeated code vs. shared libraries

The subject of whether you should use common libraries across your microservices is a gray area in microservice design. Microservice purists will tell you that you shouldn't use a custom framework across your services because it introduces artificial dependencies in your services. Changes in business logic or a bug can introduce wide-scale refactoring of all your services. On the other side, other microservice practitioners will say that a purist approach is impractical because certain situations exist (like the previous UserContextFilter example) where it makes sense to build a common library and share it across services.

I think there's a middle ground here. Common libraries are fine when dealing with infrastructure-style tasks. If you start sharing business-oriented classes, you're asking for trouble because you're breaking down the boundaries between the services.

I seem to be breaking my own advice with the code examples in this chapter because if you look at all the services in the chapter, they all have their own copies of the UserContextFilter, UserContext, and UserContextInterceptor classes.

USERCONTEXTFILTER: INTERCEPTING THE INCOMING HTTP REQUEST

The first class we're going to build is the UserContextFilter class. This class is an HTTP servlet filter that will intercept all incoming HTTP requests coming into the service and map the correlation ID (and a few other values) from the HTTP request to the UserContext class. The following code listing 8.12 shows the code for the UserContext class. The source for this class can be found in licensing-service/src/main/java/com/optimagrowth/license/utils/ UserContextFilter.java.

Listing 8.12 Mapping the correlation ID to the UserContext class

```
package com.optimagrowth.license.utils;
//Remove the imports for conciseness
@Component
public class UserContextFilter implements Filter { #A
private static final Logger logger = LoggerFactory.getLogger(UserContextFilter.class);
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
throws IOException, ServletException {
HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
UserContextHolder.getContext().setCorrelationId( #B httpServletRequest.getHeader(UserContext.CORRELATION_ID) );
UserContextHolder.getContext().setUserId(
httpServletRequest.getHeader(UserContext.USER_ID));
UserContextHolder.getContext().setAuthToken(
httpServletRequest.getHeader(UserContext.AUTH_TOKEN));
UserContextHolder.getContext().setOrganizationId(
httpServletRequest.getHeader(UserContext.ORGANIZATION_ID));
logger.debug("UserContextFilter Correlation id: {}", UserContextHolder.getContext().getCorrelationId());
filterChain.doFilter(httpServletRequest, servletResponse);
}
// Not showing the empty init and destroy methods}
```

```
#A The filter is registered and picked up by Spring through the use of the Spring @Component annotation and  
by implementing a javax.servlet.Filter interface.  
#B The filter retrieves the correlation ID from the header and sets the value on the UserContext class.
```

Ultimately, the UserContextFilter is used to map the HTTP header values you're interested in into a Java class, UserContext.

USERCONTEXT: MAKING THE HTTP HEADERS EASILY ACCESSIBLE TO THE SERVICE

The UserContext class is used to hold the HTTP header values for an individual service client request being processed by our microservice. It consists of a getter/setter method that retrieves and stores values from java.lang.ThreadLocal. The following listing 8.13 shows the code from the UserContext class. The source for this class can be found in /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContext.java.

Listing 8.13 Storing the HTTP header values inside the UserContext class

```
//Remove the imports for conciseness  
@Component  
public class UserContext {  
    public static final String CORRELATION_ID = "tmx-correlation-id";  
    public static final String AUTH_TOKEN = "tmx-auth-token";  
    public static final String USER_ID = "tmx-user-id";  
    public static final String ORGANIZATION_ID = "tmx-organization-id";  
    private String correlationId= new String();  
    private String authToken= new String();  
    private String userId = new String();  
    private String organizationId = new String();  
}
```

Now the UserContext class is nothing more than a POJO holding the values scraped from the incoming HTTP request. You use a class called /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContextHolder.java to store the UserContext in a ThreadLocal variable that is accessible in any method being invoked by the thread processing the user's request. The code for UserContextHolder is shown in the following listing 8.14.

Listing 8.14 The UserContextHolder stores the UserContext in a ThreadLocal

```
public class UserContextHolder {  
    private static final ThreadLocal<UserContext> userContext = new ThreadLocal<UserContext>();  
    public static final UserContext getContext(){  
        UserContext context = userContext.get();  
        if (context == null) {  
            context = createEmptyContext();  
            userContext.set(context);  
        }  
        return userContext.get();  
    }  
    public static final void setContext(UserContext context) {
```

```

Assert.notNull(context, "Only non-null UserContext instances are permitted");
userContext.set(context);
}
public static final UserContext createEmptyContext(){
return new UserContext();
}
}

```

CUSTOM RESTTEMPLATE AND USERCONTEXTINTERCEPTOR: ENSURING THAT THE CORRELATION ID GETS PROPAGATED FORWARD

The last piece of code that we're going to look at is the UserContextInterceptor class. This class is used to inject the correlation ID into any outgoing HTTP-based service requests being executed from a RestTemplate instance. This is done to ensure that we can establish a linkage between service calls.

To do this, we're going to use a Spring Interceptor that's being injected into the RestTemplate class. Let's look at the UserContextInterceptor in the following listing 8.15

Listing 8.15 All outgoing microservice calls have the correlation ID injected into them

```

public class UserContextInterceptor implements ClientHttpRequestInterceptor { #A
private static final Logger logger = LoggerFactory.getLogger(UserContextInterceptor.class);
@Override
public ClientHttpResponse intercept( #B
HttpRequest request, byte[] body, ClientHttpRequestExecution execution)
throws IOException {
HttpHeaders headers = request.getHeaders();
headers.add(UserContext.CORRELATION_ID, UserContextHolder.getContext().getCorrelationId()); #C
headers.add(UserContext.AUTH_TOKEN, UserContextHolder.getContext().getAuthToken());
return execution.execute(request, body);
}
}

```

#A The UserContextIntercept implements the Spring frameworks ClientHttpRequestInterceptor.
#B The intercept() method is invoked before the actual HTTP service call occurs by the RestTemplate.
#C We take the HTTP request header that's being prepared for the outgoing service call and add the correlation-ID stored in the UserContext.

To use the UserContextInterceptor, we need to define a RestTemplate bean and then add the UserContextInterceptor to it. To do this, we're going to add our own RestTemplate bean definition to the /licensing-service/src/main/java/com/optimagrowth/license/ LicenseServiceApplication.java class. The following listing 8.16 shows the method that's added to this class.

Listing 8.16 Adding the UserContextInterceptor to the RestTemplate class

```

@LoadBalanced #A
@Bean

```

```

public RestTemplate getRestTemplate(){
    RestTemplate template = new RestTemplate();
    List interceptors = template.getInterceptors();
    if (interceptors==null){ #B
        template.setInterceptors(Collections.singletonList(new UserContextInterceptor()));
    }
    else{
        interceptors.add(new UserContextInterceptor());
        template.setInterceptors(interceptors);
    }
    return template;
}

```

#A The `@LoadBalanced` annotation indicates that this `RestTemplate` object is going to use Ribbon.
#B Adding the `UserContextInterceptor` to the `RestTemplate` instance that has been created.

With this bean definition in place, any time we use the `@Autowired` annotation and inject a `RestTemplate` into a class, we'll use the `RestTemplate` created in listing 8.18 with the `UserContextInterceptor` attached to it.

Log aggregation and authentication and more

Now that you have correlation ID's being passed to each service; it's possible to trace a transaction as it flows through all the services involved in the call. To do this, you need to ensure that each service logs to a central log aggregation point that captures log entries from all of your services into a single point. Each log entry captured in the log aggregation service will have a correlation ID associated to each entry. Implementing a log aggregation solution is outside the scope of this chapter, but in chapter 10, we'll see how to use Spring Cloud Sleuth. Spring Cloud Sleuth won't use the `TrackingFilter` that you built here, but it will use the same concepts of tracking the correlation ID and ensuring that it's injected in every call.

8.6 Building a post filter receiving correlation IDs

Remember, the gateway executes the actual HTTP call on behalf of the service client and has the opportunity to inspect the response back from the target service call and then alter the response or decorate it with additional information. When coupled with capturing data with the pre-filter, a gateway post filter is an ideal location to collect metrics and complete any logging associated with the user's transaction. We'll want to take advantage of this by injecting the correlation ID that we've been passing around to our microservices back to the user.

We're going to do this by using a gateway post filter to inject the correlation ID back into the HTTP response headers being passed back to the caller of the service. This way, we can pass the correlation ID back to the caller without ever having to touch the message body. The following listing 8.17 shows the code for building a post filter. This code can be found in `/gatewayserver/src/main/java/com/optimagrowth/gateway/filters/ResponseFilter.java`.

Listing 8.17 Inject the correlation ID into the HTTP response

```
@Configuration
public class ResponseFilter {
    final Logger logger = LoggerFactory.getLogger(ResponseFilter.class);
    @Autowired
    FilterUtils filterUtils;
    @Bean
    public GlobalFilter postGlobalFilter() {
        return (exchange, chain) -> {
            return chain.filter(exchange).then(Mono.fromRunnable(() -> {
                HttpHeaders requestHeaders = exchange.getRequest().getHeaders();
                String correlationId = filterUtils.getCorrelationId(requestHeaders); #A
                logger.debug("Adding the correlation id to the outbound headers. {}", correlationId);
                exchange.getResponse().getHeaders().add(FilterUtils.CORRELATION_ID, correlationId);
                logger.debug("Completing outgoing request for {}.", #C
                    exchange.getRequest().getURI());
            }));
        };
    }
}
```

#A Grab the correlation ID that was passed in on the original HTTP request.

#B Inject the correlation ID into the response.

#C Log the outgoing request URI so that you have “bookends” that will show the incoming and outgoing entry of the user’s request into the gateway.

Once the ResponseFilter has been implemented, we can fire up our service and call the licensing or organization service through it. Once the service has completed, you’ll see a tmx-correlation-id on the HTTP response header from the call. Figure 8.13 shows the tmx-correlation-id being sent back from the call.

The screenshot shows a Postman request for 'Get Organization - Gateway'. The method is 'GET' and the URL is 'http://localhost:8072/organization/v1/organization/958aa1bf-18dc-405c-b84a-b69f04d98d4f'. The 'Headers' tab is selected, showing 7 headers. The 'Body' tab is also visible. Below the table, there is a note: 'This request did not return any data.' The table under 'Headers' has 4 rows:

KEY	VALUE
transfer-encoding	chunked
Content-Type	application/json
Date	Wed, 15 Apr 2020 00:49:29 GMT
tmx-correlation-id	e3f6a72b-7d6c-41da-ac12-fb30fc1e547

An annotation with an arrow points to the last row ('tmx-correlation-id') with the text: 'The correlation ID returned in the HTTP response'.

Figure 8.13 The tmx-correlation-id has been added to the response headers sent back to the service client.

Also, you can see the log messages in the console as shown in Figure 8.14 writing out the passed in correlation ID e3f6a72b-7d6c-41da-ac12-fb30fcde547 as it flows through the pre- and post-filter.

```

gatewayserver_1 | 2020-04-15 00:49:28.500 DEBUG 1 --- [or-http-epoll-3] c.o.gateway.filters.TrackingFilter
: tmx-correlation-id generated in tracking filter: e3f6a72b-7d6c-41da-ac12-fb30fcde547.
organizationservice_1 | 2020-04-15 00:49:28.795 INFO 1 --- [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring DispatcherServlet 'dispatcherServlet'
: Initializing Servlet 'dispatcherServlet'
organizationservice_1 | 2020-04-15 00:49:28.816 INFO 1 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet
: Completed initialization in 20 ms
organizationservice_1 | Hibernate: select organizati0_.organization_id as organizati0_0_, organizati0_.contact_email a
s contact_2_0_0_, organizati0_.contact_name as contact_3_0_0_, organizati0_.contact_phone as contact_4_0_0_, organizati
0_.name as name5_0_0_ from organizations organizati0_ where organizati0_.organization_id=?
gatewayserver_1 | 2020-04-15 00:49:29.103 DEBUG 1 --- [or-http-epoll-4] c.o.gateway.filters.ResponseFilter
: Adding the correlation id to the outgoing headers: e3f6a72b-7d6c-41da-ac12-fb30fcde547
gatewayserver_1 | 2020-04-15 00:49:29.103 DEBUG 1 --- [or-http-epoll-4] c.o.gateway.filters.ResponseFilter
: Completing outgoing request for http://localhost:8072/v1/organization/958aa1bf-18dc-405c-b84a-b69ff04d98d4f.

```

The diagram illustrates the flow of log entries from the gateway server and organization service. Annotations highlight specific parts of the log output:

- Correlation ID added in the Pre-filter data.**: Points to the first log entry from `gatewayserver_1` where a correlation ID is generated.
- Organization service processing data.**: Points to the second log entry from `organizationservice_1` which shows the execution of a Hibernate query.
- Correlation ID in the Post-filter data.**: Points to the third log entry from `gatewayserver_1` where the correlation ID is added to the outgoing response headers.
- Request URI**: Points to the final log entry from `gatewayserver_1` which specifies the completed outgoing request URI.

Figure 8.14 Logger output that shows the pre-filter, the organization service processing data and the post-filter data.

Up until this point, all our filter examples have dealt with manipulating the service client calls before and after it has been routed to its target destination. Now, that we know how to create a Spring Cloud Gateway, let's move on with our next chapter.

8.7 Summary

- Spring Cloud makes it trivial to build a services gateway.
- Spring Cloud Gateway contains a set of built-in predicates and filters factories.
- Predicates are objects that allow us to check if the requests fulfill a or a set of given conditions before executing or processing a request.
- Filters allow us to modify the incoming and outgoing HTTP requests and responses.
- The Spring Cloud Gateway integrates with Netflix's Eureka server and can automatically map services registered with Eureka to a route.
- Using the Spring Cloud Gateway, you can manually define route mappings. These route mappings are manually defined in the applications configuration files.
- By using Spring Cloud Config Server, you can dynamically reload the route mappings without having to restart the gateway server.
- Spring Cloud Gateway allows you to implement custom business logic through filters. With Spring Cloud Gateway, you can create pre-and post-filters.
- Pre-filters can be used to generate a correlation ID that can be injected into every service flowing through the gateway.
- A post-filter can inject a correlation ID into every HTTP service response back to a service client.

9 Securing your microservices

This chapter covers

- Learning why security matters in a microservice environment
- Understanding OAuth2 and OpenID
- Setting up and configuring KeyCloak
- Performing user authentication and authorization with Keycloak
- Protecting your Spring microservice using KeyCloak
- Propagating your access token between services

Now that we have a robust microservices architecture, the task of covering security vulnerabilities becomes more and more urgent and essential. In this chapter, security and vulnerability go hand in hand. We'll define vulnerability as a weakness or flaw presented in an application. Of course, all systems have vulnerabilities, but the big difference lies in whether or not these vulnerabilities are exploited to cause harm. Mentioning security often causes an involuntary groan from developers. Among developers, we hear comments such as "It's obtuse, hard to understand, and even harder to debug." Yet we won't find any developer (except for maybe an inexperienced developer) who says that they don't worry about security.

Securing a microservices architecture is a complex and laborious task that involves multiple layers of protection, including:

- *Application layer*— Ensuring that the proper user controls are in place so that we can validate that a user is who they say they are and that they have permission to do what they're trying to do.
- *Infrastructure*— Keeping the infrastructure the service is running on patched and up to date to minimize the risk of vulnerabilities.
- *Network layer*— Implementing network access controls so that a service is only accessible through well-defined ports and accessible to a small number of authorized servers.

This chapter will only cover the first bullet point from this list: how to authenticate that the user calling our microservice is who they say they are and determine whether they're authorized to carry out the action they're requesting from our microservice. The other two topics are extremely broad security topics that are outside the scope of this book. There are also other tools that can help identify vulnerabilities, such as the OWASP dependency-check.

NOTE The OWASP Dependency-check is an OWASP Software Composition Analysis (SCA) tool that allows us to identify publicly disclosed vulnerabilities. If you want to find out more, I highly recommend you look at <https://owasp.org/www-project-dependency-check/>.

To implement authorization and authentication controls, we're going to use Spring Cloud security and Keycloak to secure our Spring-based services. Keycloak is an open-source identity and access management software for modern applications and services. This open-source software was

written in Java, and it supports SAML v2 and OpenID Connect (OIDC)/OAuth2 federated identity protocols.

9.1 What is OAuth2?

OAuth2 is a token-based security framework that describes patterns for granting authorization but does not define how to actually perform authentication. Instead, it allows users to authenticate themselves with a third-party authentication service called Identity Provider (IdP). If the user successfully authenticates, they will be presented with a token that must be sent with every request. The token can then be validated back to the authentication service. The main goal behind OAuth2 is that when multiple services are called to fulfill a user's request, the user can be authenticated by each service without having to present their credentials to each service processing their request.

OAuth2 allows us to protect our REST-based services across different scenarios through authentication schemes called grants. The OAuth2 specification has four types of grants:

- Password
- Client credential
- Authorization code
- Implicit

We aren't going to walk through each of these grant types or provide code examples for each grant type. That's too much material to cover in one chapter. Instead, I'll do the following:

- Discuss how our microservices service can use OAuth2 through one of the simpler OAuth2 grant types (the

password grant type).

- Use JSON web tokens to provide a more robust OAuth2 solution and establish a standard for encoding information in an OAuth2 token.
- Walk through other security considerations that need to be taken into account when building microservices.

I provide overview material on the other OAuth2 grant types in appendix B, “OAuth2 grant types.” If you’re interested in diving into more detail on the OAuth2 spec and how to implement all the grant types, I highly recommend Justin Richer and Antonio Sanso’s book, *OAuth2 in Action* (Manning, 2017), which is a comprehensive explanation of OAuth2.

The real power behind OAuth2 is that it allows application developers to easily integrate with third-party cloud providers and do user authentication and authorization with those services without having to pass the user’s credentials to the third-party service continually.

OpenID Connect (OIDC) is a layer on top of the OAuth2 framework that provides authentication and profile information about who is logged in to the application (Identity). When an authorization server supports OIDC, it is sometimes called an Identity Provider.

Before we get into the technical details of protecting our services, let’s walk through the KeyCloak architecture.

9.2 Introduction to KeyCloak

KeyCloak is an open-source Identity and Access management software aimed at current services and applications. The

main objective of Keycloak is to facilitate the protection of the services and applications with little or no code.

Some characteristics of Keycloak are:

- Centralizes authentication and enables single sign-on authentication (SSO).
- It allows developers to focus on business functionality instead of worrying about the security aspects such as authorization and authentication.
- It allows two-factor authentication.
- It is LDAP compliant.
- It offers several adapters to secure applications and servers easily.
- It allows you to customize password policies.

Keycloak security can be broken down into four components. These four components are protected resource, resource owner, application, and Authentication/Authorization server. The four components interact together, as shown in the following figure 9.1.

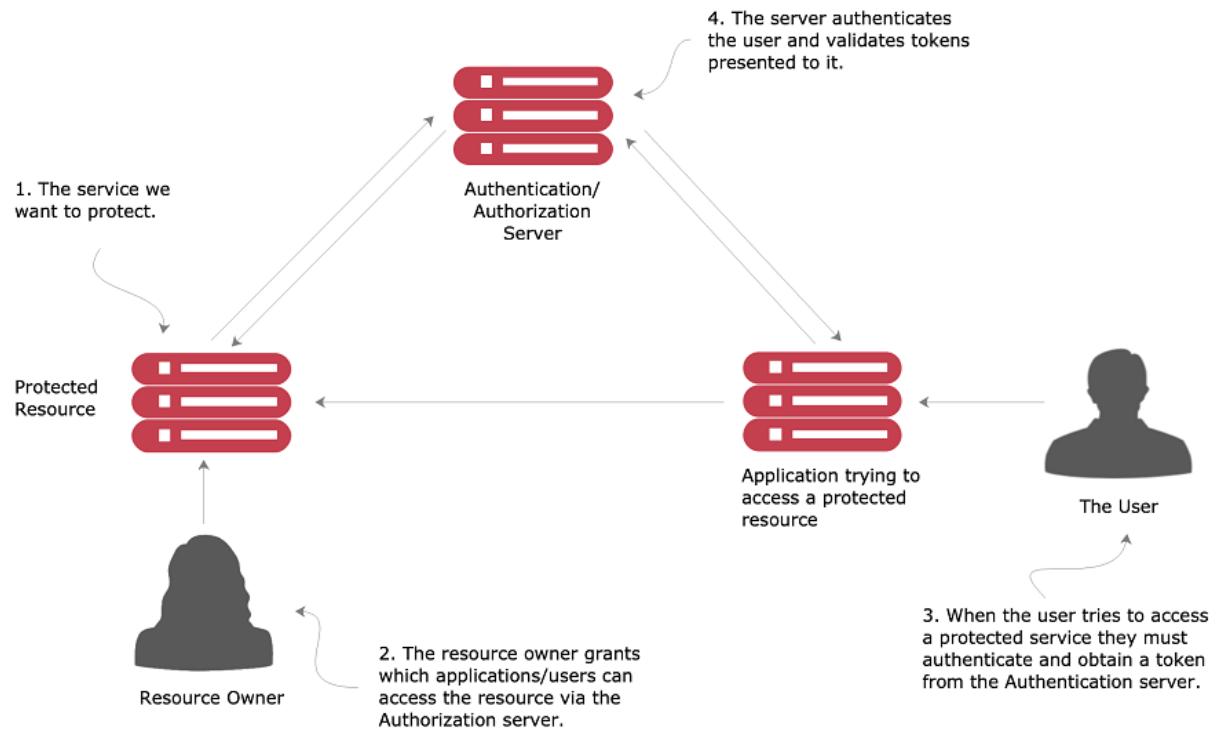


Figure 9.1 Keycloak allows a user to authenticate without constantly having to present credentials.

- **A protected resource.** This is the resource (in our case, a microservice) you want to protect and ensure that only authenticated users who have the proper authorization can access it.
- **A resource owner.** A resource owner defines what applications can call their service, which users are allowed to access the service, and what they can do with the service. Each application registered by the resource owner will be given an application name that identifies the application along with an application secret key. The combination of the application name and the secret key are part of the credentials that are passed when authenticating an access token.
- **An application.** This is the application that's going to call the service on behalf of a user. After all, users rarely

invoke a service directly. Instead, they rely on an application to do the work for them.

- **Authentication/Authorization server.** The authentication server is the intermediary between the application and the services being consumed. The Authentication server allows the user to authenticate themselves without having to pass their user credentials down to every service the application is going to call on behalf of the user.

As I previously mentioned, these components interact together to authenticate the user. The user only has to present their credentials. If they successfully authenticate, they're issued an authentication token that can be passed from service to service.

A user authenticates against the KeyCloak server by providing their credentials and the application/device they're using to access the resource. If the user's credentials are valid, the Keycloak server provides a token that can be presented every time a service is being used by the user accessing a protected resource (the microservice).

The protected resource can then contact the Keycloak server to determine the token's validity and retrieve what roles a user has assigned. Roles are used to group related users together and define what resources they can access. For this chapter's purposes, we're going to use Keycloak roles to determine what service endpoints and what HTTP verbs a user can call on an endpoint.

Web service security is an extremely complicated subject. We have to understand who's going to call our services (internal users to our corporate network, external users),

how they're going to call our service (internal web-based client, mobile device, web application outside our corporate network), and what actions they're going to take with our code.

On authentication vs. authorization

I've often found that developers "mix and match" the meaning of the term's authentication and authorization. Authentication is the act of a user proving who they are by providing credentials. Authorization determines whether a user is allowed to do what they're trying to do. For instance, the user Hillary could prove her identity by providing a user ID and password, but she may not be authorized to look at sensitive data such as payroll data. For the purposes of our discussion, a user must be authenticated before authorization takes place.

9.3 Starting small: using Spring and KeyCloak to protect a single endpoint

To understand how to set up the authentication and authorization pieces, we'll do the following:

- Add Keycloak service in docker.
- Set up a Keycloak and register Ostock application as an authorized application that can authenticate and authorize user identities.
- Use Spring Security to protect our Ostock services. We're not going to build a UI for Ostock, so instead, we'll simulate a user logging in to use POSTMAN to authenticate against our Keycloak service.
- Protect the licensing and organization service so that they can only be called by an authenticated user.

9.3.1 Adding Keycloak service in Docker

This section will explain to you how to add the Keycloak service in our Docker environment. To achieve this, let's start by adding the following code shown in code listing 9.1 to our docker-compose.yml file

NOTE In case you didn't follow the previous chapter's code listings, you can download the code created in chapter 8 from the following link:
<https://github.com/ihuaylupo/manning-smia/tree/master/chapter8>.

Listing 9.1 Keycloak service in docker-compose.yml

```
Rest of docker-compose.yml removed for conciseness
...
keycloak: #A
image: jboss/keycloak
restart: always
environment:
KEYCLOAK_USER: admin #B
KEYCLOAK_PASSWORD: admin #C
ports:
- "8080:8080"
networks:
backend:
aliases:
- "keycloak"
```

```
#A Keycloak Docker service name
#B KeyCloak administrative console's username
#C KeyCloak administrative console's password
```

It is essential to highlight that Keycloak can be used with several databases such as H2, PostgreSQL, MySQL, Microsoft SQL Server, Oracle, and MariaDB. For this specific example, I will be using the default embedded H2 database, but I highly recommend visiting the following link if you want to use

another database. <https://github.com/keycloak/keycloak-containers/tree/master/docker-compose-examples>

If you noticed the previous code listing 9.1, I'm using the 8080 port now for the Keycloak service, and if we go back a few chapters, we also exposed the licensing service in that port. I will be mapping the 8180 instead of the 8080 port to the licensing service in the docker-compose.yml file to make everything work. The following code shows you how.

```
licensingService:  
  image: ostock/licensing-service:0.0.3-SNAPSHOT  
  ports:  
    - "8180:8080"
```

NOTE To make Keycloak work in our local environment, we need to add the following host entry 127.0.0.1 Keycloak into our hosts' file. If you are using Windows, you need to add the host entry at C:\Windows\System32\drivers\etc\hosts, and if you are using Linux, the host file is at /etc/hosts. Why do we need that entry? The containers can talk to each other using the network aliases or MAC addresses, but our postman need to use localhost to invoke the services.

9.3.2 Setting up KeyCloak

Now that we have our Keycloak service in the docker-compose.yml let's run the following command at the root folder of our code.

```
docker-compose -f docker/docker-compose.yml up
```

Once the services are up, let's visit the following link <http://keycloak:8080/auth/> to open the Keycloak administrative console.

Configuring KeyCloak is a straightforward process. The first time we access KeyCloak, a welcome page will be displayed. This page will display different options, such as visit the Administration Console, documentation, report issues, and more. In our case, we are going to click the Administration console link. Figure 9.2 shows the welcome page.

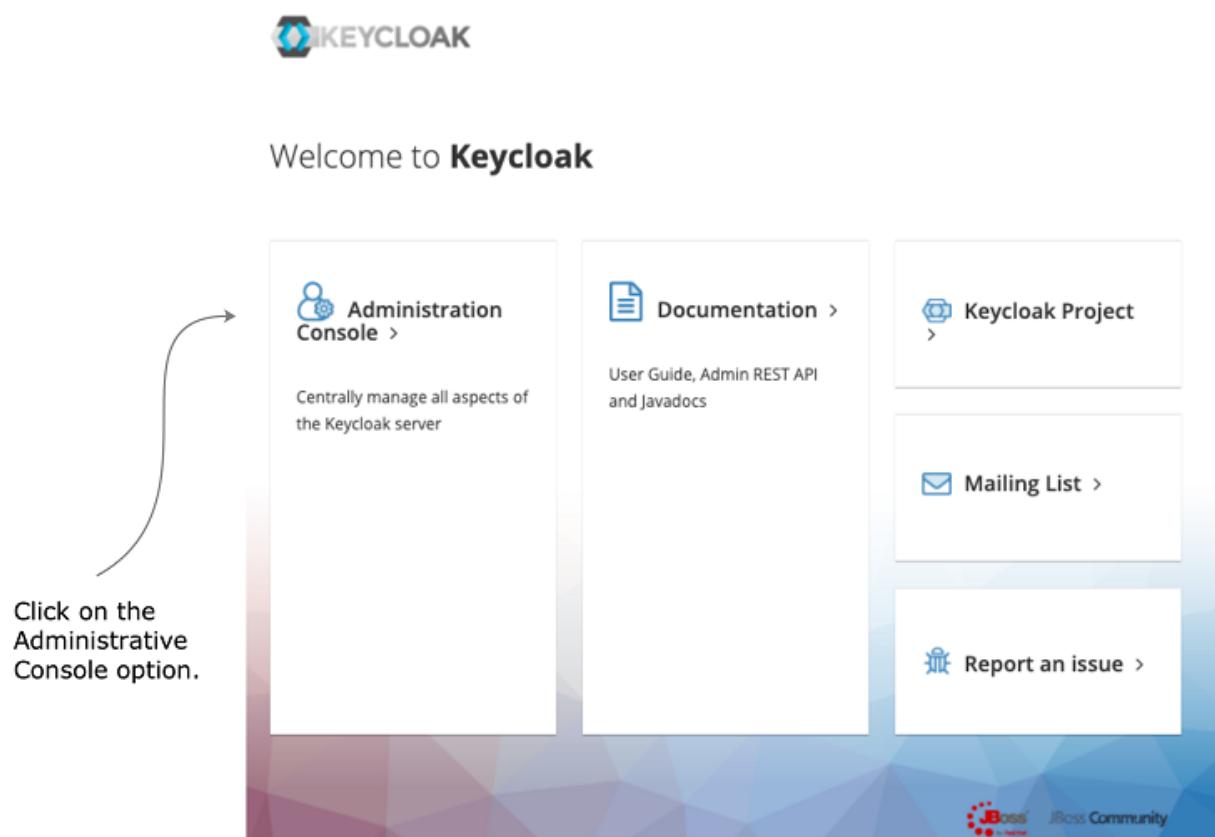


Figure 9.2 KeyCloak's welcome page offers several options: Administration console, Documentation, Report issues and more.

The next step is to enter the username and password data that we previously defined in the docker-compose.yml file. Figure 9.3 shows this step.

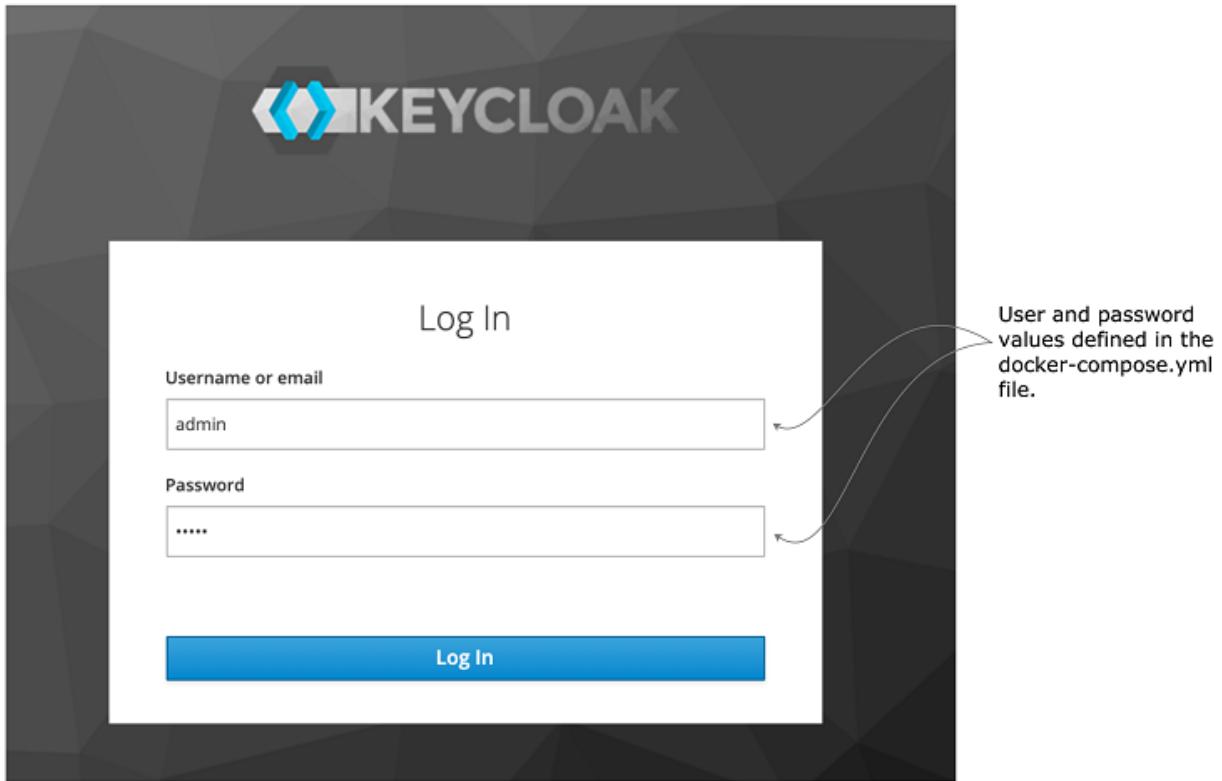


Figure 9.3 KeyCloak's login page.

To continue configuring our data, let's create our Realm. A realm is a concept that Keycloak uses to refer to an object that manages a set of users, credentials, roles, and groups.

To create our Realm, Click the Add realm option that is shown in the Master drop-down menu. Figure 9.4 will show you how to create the spmia-realm in our Keycloak service.

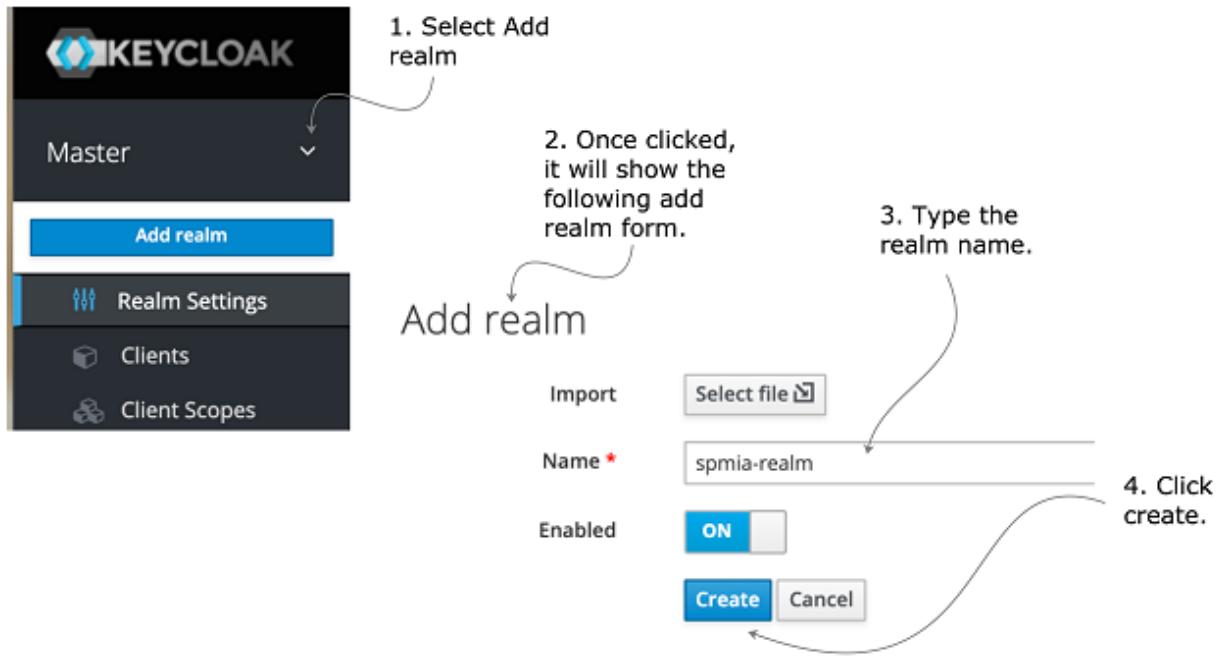


Figure 9.4 KeyCloak Realm creation page displays a form that allows the user to enter the realm name.

Once the Realm is created, you will see the Realm Main page with the configuration shown in Figure 9.5.

The screenshot shows the Keycloak interface for managing a realm named 'Spmia-realm'. The left sidebar has a dark theme with white icons and text. It includes sections for 'Configure' (with 'Realm Settings' selected), 'Manage' (with 'Groups', 'Users', 'Sessions', 'Events', 'Import', and 'Export' listed), and a dropdown for 'Spmia-realm'. The main content area is titled 'Spmia-realm' and shows the 'General' tab selected. The 'General' tab contains fields for 'Name' (set to 'spmia-realm'), 'Display name', 'HTML Display name', 'Frontend URL', and two toggle switches: 'Enabled' (set to 'ON') and 'User-Managed Access' (set to 'OFF'). Below these are 'Endpoints' for 'OpenID Endpoint Configuration' and 'SAML 2.0 Identity Provider Metadata'. At the bottom are 'Save' and 'Cancel' buttons.

Figure 9.5 KeyCloak spmia-realm Configuration.

9.3.3 Registering a client application

The next step in our configuration is to create a Client. Clients in KeyCloak are entities that can request user authentication. The clients are often the applications or services that we want to secure and provide a single sign-on solution. To create a Client, let's click on the Clients option in the left menu. Once clicked, you will see the page shown in Figure 9.6.

The screenshot shows the Keycloak interface for managing clients in the 'Spmia-realm'. The left sidebar has a dropdown for 'Spmia-realm' and links for 'Configure', 'Realm Settings', 'Clients' (which is selected), 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main area is titled 'Clients' with a 'Lookup' button. Below is a table with columns 'Client ID', 'Enabled', 'Base URL', and 'Actions' (Edit, Export, Delete). The table contains six rows corresponding to the clients listed. A red box highlights the 'Create' button in the top right corner of the table header, with a callout line pointing to it labeled 'Click create.'

Client ID	Enabled	Base URL	Actions		
account	True	http://localhost:8180/auth/realms/spmia-realm/account/	Edit	Export	Delete
account-console	True	http://localhost:8180/auth/realms/spmia-realm/account/	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	http://localhost:8180/auth/admin/spmia-realm/console/	Edit	Export	Delete

Figure 9.6 KeyCloak Spmia-realm clients.

Once the Client's list is shown, click on the Create button displayed on the table's top right corner, as shown in the previous figure. Once clicked, you will see an add client form asking for the following information:

- Client ID
- Client Protocol
- Root URL

Please enter the information, as shown in figure 9.7.

Add Client

Import

Client ID *

Client Protocol

Root URL

Ostock Client ID.

Select the openid-connect protocol.

Click Save.

Figure 9.7 Ostock KeyCloak Client.

After saving the client, you will be presented with the client configuration page shown in figure 9.8. On that page, we are going to select the type and select the following data:

- **Access Type:** Confidential
- **Service Accounts Enabled:** On
- **Authorization Enabled:** On
- **Valid Redirect URLs:** http://localhost:80*
- **Web Origins:** *

The screenshot shows the configuration page for a client named 'Ostock' in Keycloak. The configuration fields and their current values are:

- Client Protocol**: openid-connect
- Access Type**: confidential (marked with a red asterisk)
- Standard Flow Enabled**: ON
- Implicit Flow Enabled**: OFF
- Direct Access Grants Enabled**: ON
- Service Accounts Enabled**: ON
- Authorization Enabled**: ON
- Root URL**: (empty field)
- * Valid Redirect URIs**: http://localhost:80* (with a plus sign for adding more)
- Base URL**: (empty field)
- Admin URL**: (empty field)
- Web Origins**: * (with a plus sign for adding more)

Annotations with arrows point to specific fields:

1. Select the confidential access type.
2. Enable Authorization.
3. Valid Request URIs
4. Web Origins
5. Click the save button located at the end of the page.

Figure 9.8 Ostock KeyCloak Client additional configuration.

For purposes of this example, I will only create a global client called Ostock, but here you can have as many clients as you want.

The next step is to set up the Client Roles, so let's click the Roles tab. To better understand the client roles concept, let's imagine that our application will have two types of users: Admins and Regular Users. The admin users will be able to

execute all of the application services, and the regular users are only going to be allowed to execute some services.

Once the Roles page is loaded, you will be seeing a list of pre-defined client roles. Let's click the Add Role button displayed in the top right corner of the Roles table. Once clicked, you will see the add Role form shown in figure 9.9.

Clients > ostock > Roles > Add Role

Add Role

Role Name *

Description

Save **Cancel**

Figure 9.9 KeyCloak Client Add Roles page.

At this step, you need to create the following Client Roles:

- USER
- ADMIN

So, please add one and then repeat the step. In the end, you will have a list similar to the one shown in figure 9.10.

Search...		View all roles	Add Role	
Role Name	Composite	Description	Actions	
ADMIN	False		Edit	Delete
USER	False		Edit	Delete
uma_protection	False		Edit	Delete

Figure 9.10 KeyCloak Ostock Client Roles.

Now that we have finished our basic Client configuration, let's make a pause and go to the Credentials tab. The credentials page will show the Client Secret required in the authentication process. The following Figure 9.11 will show how the credential page looks like.

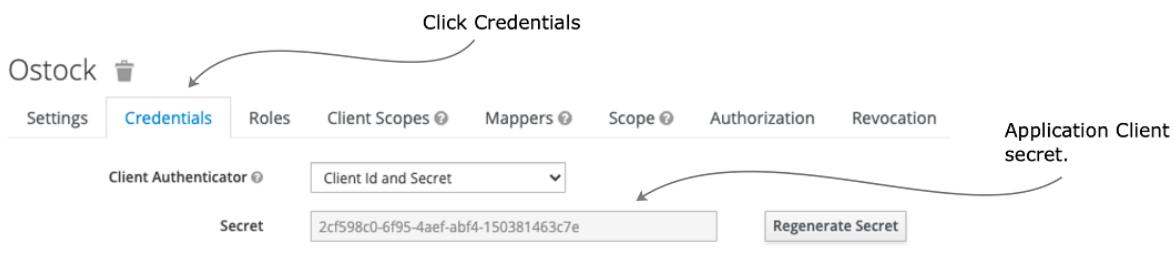


Figure 9.11 KeyCloak Ostock Client Secret

The next step in our configuration is to create Realm Roles. The Realm roles will allow us to have better control of what roles are being set for each user. This is an optional step. If you don't want to create these roles, you can go ahead and create the users directly, but later on, it can be harder to identify and maintain the roles for each user.

To create the Realm Roles, let's click the Roles option shown in the left menu and then click the Add Role button in the

Roles table's top right. Same as the client roles, we will be creating two types of Realm roles, the ostock-user, and the ostock-admin. Figures 9.12 and 9.13 will show you how to create a Realm Role.

[Roles](#) > Add Role

Add Role

* Role Name

Description

Save Cancel

Figure 9.12 Creating the ostock-admin realm role.

Roles > ostock-admin

Ostock-admin

- Details**
- Attributes
- Users in Role

Role Name: ostock-admin

Description:

Composite Roles: ON

Save Cancel

Composite Roles

Realm Roles	Available Roles	Associated Roles
	offline_access uma_authorization	
	Add selected >	< Remove selected

Client Roles	Available Roles	Associated Roles
ostock	uma_protection USER	ADMIN
	Add selected >	< Remove selected

Figure 9.13 Specifying additional configuration for the ostock-admin realm role.

Now that we have the ostock-admin Realm role configured let's repeat the same steps to create the ostock-user role. Once you are done, you should have a similar list to the one shown in figure 9.14.

Roles

Realm Roles		Default Roles		
Search... <input type="text"/>		<input type="button" value="View all roles"/>	<input type="button" value="Add Role"/>	
Role Name	Composite	Description	Actions	
offline_access	False	\$(role_offline-access)	Edit	Delete
ostock-admin	True		Edit	Delete
ostock-user	True		Edit	Delete
uma_authorization	False	\$(role_uma_authorization)	Edit	Delete

Figure 9.14 Spmia-realm Roles.

9.3.4 Configuring Ostock users

Now that we've defined application-level and realm-level roles, names, and secrets. We're now going to set up individual user credentials and the roles that they belong to. To create the users, let's click on the Users option shown in the left menu.

For the examples in this chapter, we will define two user accounts: illary.huaylupo and john.carnell. The john.carnell account will have the role of ostock-user and the illary.huaylupo account will have the role of ostock-admin. Figure 9.15 shows the user creation page.

Add user

ID

Created At

Username *

Email

First Name

Last Name

User Enabled ON

Email Verified ON

Required User Actions

Save **Cancel**

Figure 9.15 Spmia-realm User creation page.

On this page, let's type the username and enable the email verified option, as shown in the previous figure.

NOTE Keycloak allows us to also add additional attributes to the users such as first name, last name, email, address, birth date, phone number, and more. But for the purposes of this example, I will only be setting up the required attributes.

Once it is saved, click the Credentials tab. You need to type the user's password, disable the temporary option, and click the set password button on the credentials page. Figure 9.16 shows this step.

Manage Credentials

Position	Type	User Label	Data	Actions
----------	------	------------	------	---------

Set Password

Password

Password Confirmation

Temporary 

OFF 

 Turn Off

Figure 9.16 Set the user's password and disable the temporary option.

Once the password is set, let's click the Role Mappings tab and assign the user's specific role. Figure 9.17 shows this step.

The screenshot shows the 'Role Mappings' tab selected for the user 'Hillary.huaylupo'. The interface is divided into four main sections: 'Realm Roles', 'Available Roles', 'Assigned Roles', and 'Effective Roles'. In the 'Available Roles' section, there is one role: 'ostock-user'. In the 'Assigned Roles' section, three roles are listed: 'offline_access', 'ostock-admin' (which is circled in red), and 'uma_authorization'. In the 'Effective Roles' section, the same three roles are listed. Below these sections, there is a 'Client Roles' section with a dropdown menu labeled 'Select a client...'. At the top of the page, there are tabs for 'Details', 'Attributes', 'Credentials', 'Role Mappings' (which is highlighted in blue), 'Groups', 'Consents', and 'Sessions'.

Figure 9.17 Map the realm roles to the created user.

To finish our configuration, let's repeat the same steps for the other user in this example for the john.carnell user.

9.3.5 Authenticating the user

At this point, we have enough of our base Keycloak server functionality in place to perform application and user authentication for the password grant flow. Now, let's start our authentication service. To achieve this, let's click the Realm settings option in the left menu and click the OpenID Endpoint Configuration link to see the list of endpoints available in our Realm. These steps are shown in the following figures 9.18 and 9.19.

1. Click the realm settings option.

Spmia-realm

Configure

Realm Settings

Clients

Client Scopes

Roles

Identity

Providers

User Federation

Authentication

Manage

Groups

Users

Sessions

General Login Keys Email Themes Cache

* Name spmia-realm

Display name

HTML Display name

Frontend URL

Enabled ON

User-Managed Access OFF

Endpoints

OpenID Endpoint Configuration

SAML 2.0 Identity Provider Metadata

Save Cancel

Figure 9.18 Realm OpenID endpoints link.

```
{
  "issuer": "http://keycloak:8080/auth/realm/spmia-realm",
  "authorization_endpoint": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/auth",
  "token_endpoint": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/token",
  "introspection_endpoint": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/token/introspect",
  "userinfo_endpoint": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/userinfo",
  "end_session_endpoint": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/logout",
  "jwks_uri": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/certs",
  "check_session_iframe": "http://keycloak:8080/auth/realm/spmia-realm/protocol/openid-connect/login-status-iframe.html",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials"
  ],
}
```

Copy the token endpoint.

Figure 9.19 Map the realm roles to the created user.

Now we'll simulate a user acquiring an access token by using POSTMAN to POST to the token endpoint <http://keycloak:8080/auth/realms/spmia-realm/protocol/openid-connect/token> and provide the application, secret key, user ID, and password.

NOTE Remember, we use the 8080 port because it is the port we previously defined in the docker-compose.yml file for the KeyCloak service.

To simulate a user acquiring an authentication token first, we need to set up POSTMAN with the application name and secret key. We're going to pass these elements to our Authentication server endpoint using basic authentication. Figure 9.20 shows how POSTMAN is set up to execute a basic authentication call. Remember, we will use the application name we previously defined and the secret application key as a password.

The screenshot shows the POSTMAN interface with the following details:

- Header Bar:** Shows "POST" method and URL <http://keycloak:8080/auth/realms/spmia-realm/protocol/openid-connect/token>.
- Authorization Tab:** Selected tab, showing "Basic Auth" selected under "TYPE". A note says: "The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)".
- Params Tab:** Not selected.
- Headers Tab:** Contains 9 items.
- Body Tab:** Not selected.
- Tests Tab:** Not selected.
- Settings Tab:** Not selected.
- Authorization Fields:** "Username" field contains "ostock", "Password" field contains "2cf598c0-6f95-4aef-abf4-150381463c7e", and a checked "Show Password" checkbox.
- Notes:**
 - A callout points to the "Username" field with the text: "Type the client ID defined previously in Keycloak."
 - A callout points to the "Password" field with the text: "Type Client Secret shown in the client credential's tab."
- Footnote:** A note at the bottom left of the Authorization tab area states: "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaboration, use variables".

Figure 9.20 Setting up basic authentication using the application key and secret

However, we're not ready to make the call to get the token yet. Once the application name and secret key are configured, we need to pass in the following information in the service as HTTP form parameters:

- **grant_type**. The grant type you're executing. In this example, we will use a password grant.
- **username**. Name of the user logging in.
- **password**. Password of the user logging in.

The following figure 9.21 shows how HTTP form parameters are configured for our Authentication call.

The screenshot shows the Postman interface with a POST request to `http://keycloak:8080/auth/realms/spmia-realm/protocol/openid-connect/token`. The 'Body' tab is selected, showing the following form parameters:

KEY	VALUE
<input checked="" type="checkbox"/> grant_type	password
<input checked="" type="checkbox"/> username	illary.huaylupo
<input checked="" type="checkbox"/> password	password1

Annotations on the right side of the interface:

- An annotation points to the 'grant_type' value with the text: "1. Type password as the grant_type".
- An annotation points to the 'username' and 'password' values with the text: "2. Type the username and password".

Figure 9.21 When requesting an access token, the user's credentials are passed in as HTTP form parameters to the /openid-connect/token endpoint.

Unlike other REST calls in this book, this list's parameters will not be passed in as a JSON body. The authentication

standard expects all parameters passed to the token generation endpoint to be HTTP form parameters.

Figure 9.22 shows the JSON payload that's returned from the /openid-connect/token call.

```
Body Cookies Headers (13) Test Results
Pretty Raw Preview Visualize JSON 🔍
1 {
2   "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJSLUzoZGZQNI eyJleHAiOjE2MDI4NzксImlhdCI6MTYwMjg30Dg0NiwanRpIjoi0TjlNzQzMGYtNj00MS0t YWxtIiwiYXVkIjoiYWNjb3VudCIsInN1YiI6ImZlNDZjYWIwLWNj0TYtNDY1YS1i0Dg3LTyYzg; LTFl0GMtNDY5MS04NDlilWY40TBjMWIyYjYyNSiImFjciI6IjEiLCJhbGxvd2VklW9yaWdpbnM: bwLuLWFwcCIsInVtYV9hdXR0b3JpemF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3Mi0nsiZ2F0ZxdI Z2UtYWNjb3VudC1saW5rcyIsInZpZXctcHJvZmlsZSJdfX0sInNb3BlIjoiChJvZmlsZSBlbWFj ozq0bCq9xQJ1sERDrBm20dcYuzIAAnq0825-XXk-EVe05BrLYp5oXu1ULIXtbz2cELD2H5luXU02i o9pkZ0bIxWAICw9votmbDCe2zw0RZV1QGu6MbFrTkZKDWF9MB2r8YHu0Zler72a00VrPEP5xdj
3   "expires_in": 300,
4   "refresh_expires_in": 1800,
5   "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIwZGU1MmMyZ: eyJleHAiOjE2MDI40DA2NDYsImlhdCI6MTYwMjg30Dg0NiwanRpIjoiMjM4NDdkMDktYjY10C0t YWxtIiwiYXVkIjoiHR0cDovL2xvY2FsaG9zdDo4MTgwL2F1dGgvcvvhbG1zL3NwbWlhLXJlyWx: dGV3YXktc2Vydmyiic2Vzc2lvbl9zdgF0ZSI6IjI2YjBiMDRmLTFl0GMtNDY5MS04NDlilWY-
6   "token_type": "bearer",
7   "not-before-policy": 0,
8   "session_state": "26b0b04f-1e8c-4691-849b-f890c1b2b625",
9   "scope": "profile email"
10 }
```

The number of seconds before the access token expires

The type of access token being generated

This is the key field. The access_token is the authentication token presented with each call.

The defined scope for which the token is valid

The token that is presented when the access token expires and needs to be refreshed

Figure 9.22 Payload returned after a successful client credential validation

The payload returned contains five attributes:

- **access_token**. The access token that will be presented with each service call the user makes to a protected resource.
- **token_type**. The type of token. The Authorization specification allows us to define multiple token types. The most common token type used is the bearer token.

We won't cover any of the other token types in this chapter.

- **refresh_token**. Contains a token that can be presented back to the Authorization server to reissue a token after it has been expired.
- **expires_in**. This is the number of seconds before the access token expires. The default value for authorization token expiration in Spring is 12 hours.
- **Scope**. The scope that this access token is valid for.

Now that we have retrieved a valid access token from the Authorization server, we can decode the JWT with

<https://jwt.io> to retrieve all the access token information.

Figure 9.23 shows what the results of the decoded JWT.

The screenshot shows the JJWT online token decoder interface. On the left, under 'Encoded', is a long base64-encoded JWT token. On the right, under 'Decoded', is the JSON representation of the token.

```

HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "w-ravex6cnZkTeArgGS2zhCUJPwbgE2bct_vXyUrg_U"
}

PAYLOAD: DATA
{
  "exp": 1683066302,
  "iat": 1683066002,
  "jti": "4dd325a8-9651-408e-b6c9-7411828d444e",
  "iss": "http://keycloak:8080/auth/realms/spmia-realm",
  "aud": "account",
  "sub": "b337eb52-4ed3-4738-ba69-ef9519d7f967",
  "typ": "Bearer",
  "azp": "ostock",
  "session_state": "1d3ec929-59b4-4f0f-9528-3f4ab27d5ae6",
  "acr": "1",
  "allowed_origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "offline_access",
      "uma_authorization",
      "ostock-admin"
    ]
  },
  "resource_access": {
    "ostock": {
      "roles": [
        "ADMIN"
      ]
    }
  },
  "account": {
    "roles": [
      "manage-account",
      "manage-account-links",
      "view-profile"
    ]
  }
},
  "scope": "profile email",
  "email_verified": true,
  "preferred_username": "illary.huaylupo"
}

```

Annotations with arrows point from specific fields in the decoded payload to their corresponding definitions in the Keycloak realm configuration:

- An arrow points from the 'realm_access/roles' field to the 'realm_access/roles' section in the realm configuration, which includes 'offline_access', 'uma_authorization', and 'ostock-admin'.
- An arrow points from the 'resource_access/ostock/roles' field to the 'resource_access/ostock/roles' section in the realm configuration, which includes 'ADMIN'.
- An arrow points from the 'account/roles' field to the 'account/roles' section in the realm configuration, which includes 'manage-account', 'manage-account-links', and 'view-profile'.
- An arrow points from the 'preferred_username' field to the 'users' section in the realm configuration, specifically pointing to the user 'illary.huaylupo'.

Figure 9.23 Looking up user information based on the issued access token

9.4 Protecting the organization service using KeyCloak

Once you've registered a client in our KeyCloak server and set up individual user accounts with roles, we can begin exploring how to protect a resource using Spring Security and the Keycloak Spring Boot Adapter. While the creation

and management of access tokens is the KeyCloak server's responsibility, in Spring, the definition of what user roles have permissions to do what actions occur at the individual service level.

To set up a protected resource, we need to take the following actions:

- Add the appropriate Spring Security and KeyCloak jars to the service we're protecting
- Configure the service to point to our KeyCloak server
- Define what and who can access the service

Let's start with one of the simplest examples of setting up a protected resource by taking our organization service and ensuring that it can only be called by an authenticated user.

9.4.1 Adding the Spring Security and KeyCloak jars to the individual services

As usual with Spring microservices, we have to add a couple of dependencies to the organization service's Maven organization-service/pom.xml file. The following code listing 9.2 shows the new dependencies.

Listing 9.2 Configuring KeyCloak and Spring security dependencies

```
//Rest of pom.xml removed for conciseness
<dependency>
<groupId>org.keycloak</groupId>
<artifactId>keycloak-spring-boot-starter</artifactId> #A
</dependency>
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId> #B
</dependency>
</dependencies>
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>org.keycloak.bom</groupId>
<artifactId>keycloak-adapter-bom</artifactId> #C
<version>11.0.2</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

#A KeyCloak Spring Boot dependency.

#B Spring security starter dependency.

#C KeyCloak Spring Boot Dependency Management

9.4.2 Configuring the service to point to our KeyCloak server

Remember that once we set up the organization service as a protected resource, every time a call is made to the service, the caller must include the Authentication HTTP header containing a Bearer access token to the service. Our protected resource then has to call back to the KeyCloak server to see if the token is valid.

The following code listing 9.3 shows the required KeyCloak configuration. You need to add this configuration to the organization service application properties located in the configuration server repository.

Listing 9.3 KeyCloak configuration in the organization-service.properties file

```
//Rest of properties removed for conciseness
keycloak.realm = spmia-realm #A
keycloak.auth-server-url = http://keycloak:8080/auth #B
keycloak.ssl-required = external
keycloak.resource = ostock #C
keycloak.credentials.secret = 5988f899-a5bf-4f76-b15f-f1cd0d2c81ba #D
keycloak.use-resource-role-mappings = true
keycloak.bearer-only = true
```

#A The created realm name.

#B The KeyCloak server URL Auth endpoint.

http://<KEYCLOAK_SERVER_URL>/auth

#C The created Client ID.

#D The created Client Secret.

NOTE For purposes of making more straightforward the examples of this book, I'm using the classpath repository, and you can find the configuration file in the following path:
`/configserver/src/main/resources/config/organization-service.properties.`

9.4.3 Defining who and what can access the service

We're now ready to begin defining the access control rules around the service. To define access control rules, we need to extend a KeycloakWebSecurityConfigurerAdapter class and override the configure(), configureGlobal(), sessionAuthenticationStrategy() and KeycloakConfigResolver methods. In the organization service, our SecurityConfig class is located in `/organization-service/src/main/java/com/optimagrowth/organization/config/SecurityConfig.java`. The following code listing 9.4 shows the SecurityConfig.java code.

Listing 9.4 Security Config Code

```
//Rest of properties removed for conciseness
@Configuration #A
@EnableWebSecurity #B
@EnableGlobalMethodSecurity(jsr250Enabled = true) #C
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter { #D
@Override
protected void configure(HttpSecurity http) throws Exception { #E
super.configure(http);
http.authorizeRequests()
.anyRequest()
.permitAll();
http.csrf().disable();
}
@.Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) #F
throws Exception {
KeycloakAuthenticationProvider keycloakAuthenticationProvider =
keycloakAuthenticationProvider();
keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new
SimpleAuthorityMapper());
auth.authenticationProvider(keycloakAuthenticationProvider);
}
@Bean
@Override
protected SessionAuthenticationStrategy sessionAuthenticationStrategy() { #G
return new RegisterSessionAuthenticationStrategy(new SessionRegistryImpl());
}
@Bean
public KeycloakConfigResolver KeycloakConfigResolver() { #H
return new KeycloakSpringBootConfigResolver();
}
}
```

#A The class must be marked with the @Configuration annotation.

#B Applies the configuration to the global WebSecurity.

#C Annotation that allow us to use the @RoleAllowed annotation.

#D The SecurityConfig class needs to extend

KeycloakWebSecurityConfigurerAdapter.

#E Registers they KeyCloakAuthentication provider.

#F Defines the session authentication strategy.

#G Defines the session authentication strategy.

#H By default the Spring Security Adapter will look for a keycloak.json file.

Access rules can range from extremely coarse-grained (any authenticated user can access the entire service) to fine-

grained (only the application with this role, accessing this URL through a DELETE is allowed).

We can't discuss every permutation of Spring Security's access control rules, but we can look at several of the more common examples. These examples include protecting a resource so that

- Only authenticated users can access a service URL
- Only users with a specific role can access a service URL

PROTECTING A SERVICE BY AN AUTHENTICATED USER

The first thing we're going to do is protect the organization service so that it can only be accessed by an authenticated user. The following code listing 9.5 shows how you can build this rule into the SecurityConfig.java class.

Listing 9.5 Restricting access to only authenticated users

```
package com.optimagrowth.organization.security;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {#A
super.configure(http);
http.authorizeRequests()
.anyRequest().authenticated();
}
```

```
#A All-access rules are configured off the HttpSecurity object passed into the  
method.
```

All-access rules are going to be defined inside the `configure()` method. We'll use the `HttpSecurity` class passed in by Spring to define our rules. In this example, we will restrict all access to any URL in the organization service to authenticated users only.

Suppose we were to access the organization service without an access token present in the HTTP header. In that case, we'd get a 401 HTTP response code along with a message indicating that a full authentication to the service is required.

Figure 9.24 shows the output of a call to the organization service without the Authentication HTTP header.

The screenshot shows a REST API testing interface. At the top, there are tabs for 'Body', 'Cookies (1)', 'Headers (12)', and 'Test Results'. The 'Body' tab is selected, indicated by an orange underline. To the right, the status is shown as 'Status: 401 Unauthorized'. Below the tabs, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and a dropdown set to 'JSON'. A red error icon is visible next to the JSON button. The JSON response body is displayed in a code editor-like format with line numbers 1 through 7. The response content is:

```
1 {  
2   "timestamp": "2020-10-19T00:56:59.823+0000",  
3   "status": 401,  
4   "error": "Unauthorized",  
5   "message": "Unauthorized",  
6   "path": "/v1/organization/1445c05d-d90b-442f-b555-57b9d2ab305c"  
7 }
```

A callout bubble points from the 'Status: 401 Unauthorized' text to the line 'HTTP Status code 401 is returned.' located near the bottom right of the JSON response area.

Figure 9.24 Trying to call the organization service will result in a failed call.

Next, we'll call the organization service with an access token. To get an access token, see section 9.2.5, "Authenticating the user," on how to generate the token. We want to cut and paste the `access_token` field's value from the returned JSON

call out to the /openid-connect/token endpoint and use it in our call to the organization service. Remember, when we call the organization service, we need to set the authorization type to Bearer Token with the value access_token value.

Figure 9.25 shows the callout to the organization service, but this time with an access token passed to it.

The screenshot shows a Postman request configuration for a GET request to `http://localhost:8072/organization/v1/organization/e6a625cc-718b-48c2-ac76-1dfdff9a531e`. The 'Authorization' tab is selected, showing 'Bearer Token' as the type. A note warns about sensitive data. The 'Token' field contains a JWT token. The 'Body' tab shows a JSON response with an organization's details. An annotation points to the token in the 'Token' field with the text 'Access token is passed in the header.'

GET http://localhost:8072/organization/v1/organization/e6a625cc-718b-48c2-ac76-1dfdff9a531e

Params Authorization ● Headers (8) Body Pre-request Script Tests Settings

TYPE
Bearer Token

Bearer Token Authorization Type
The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Token eyJhbGciOiJSUzI1NiIsInR5cC IgOIAiSlldUliwi a2

Access token is passed in the header.

Body Cookies (1) Headers (11) Test Results

Pretty Raw Preview Visualize JSON ↴

```
1 {  
2   "id": "e6a625cc-718b-48c2-ac76-1dfdff9a531e",  
3   "name": "Ostock",  
4   "contactName": "Hillary Huaylupo",  
5   "contactEmail": "illaryhs@gmail.com",  
6   "contactPhone": "8888888888"  
7 }
```

Status: 200 OK Time: 272 ms

Figure 9.25 Passing in the access token on the call to the organization service

This is probably one of the most straightforward use cases for protecting an endpoint using Json Web Tokens (JWT). Next, we'll build on this and restrict access to a specific endpoint to a specific role.

PROTECTING A SERVICE VIA A SPECIFIC ROLE

In the next example, we're going to lock down the DELETE call on our organization service to only those users with ADMIN access. As we'll remember from section 9.2.4, "Configuring Ostock users," we created two user accounts that could access Ostock services: illary.huaylupo and john.carnell. The john.carnell account had the role of USER assigned to it. The illary.huaylupo account had the USER role and the ADMIN role.

We can achieve this by using the @RolesAllowed annotation in the controller. The following code listings 9.6 will show you how.

Listing 9.6 Using the @RolesAllowedAnnotation in the OrganizationController.java

```
package com.optimagrowth.organization.security;
//Imports removed for conciseness
@RestController
@RequestMapping(value="v1/organization")
public class OrganizationController {
@.Autowired
private OrganizationService service;
@RolesAllowed({ "ADMIN", "USER" })
@RequestMapping(value="/{organizationId}",method = RequestMethod.GET)
public ResponseEntity<Organization> getOrganization(
@PathVariable("organizationId") String organizationId) {
return ResponseEntity.ok(service.findById(organizationId));
}
@RolesAllowed({ "ADMIN", "USER" })
@RequestMapping(value="/{organizationId}",method = RequestMethod.PUT)
public void updateOrganization( @PathVariable("organizationId") String id,
@RequestBody Organization organization) {
service.update(organization);
}
@RolesAllowed({ "ADMIN", "USER" })
@PostMapping
public ResponseEntity<Organization> saveOrganization(@RequestBody Organization
organization) {
return ResponseEntity.ok(service.create(organization));
}
```

```
    }
    @RolesAllowed("ADMIN") #B
    @DeleteMapping(value="/{organizationId}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteLicense(@PathVariable("organizationId") String organizationId) {
        service.delete(organizationId);
    }
}
```

#A RolesAllowed annotation indicating that only users containing the USER and ADMIN role can execute that action.

#B RolesAllowed annotation indicating that only users containing the ADMIN role can execute that action.

Now, to obtain the token for john.carnell (password: password1) we need to execute the openid-connect/token post request again. Once we have the new access token, we need to call the DELETE endpoint for the organization service (<http://localhost:8072/organization/v1/organization/dfd13002-57c5-47ce-a4c2-a1fda2f51513>). We'd get a 403 Forbidden HTTP status code on the call and an error message indicating that the access was denied for this service. The JSON text returned by our call would be

```
{
    "timestamp": "2020-10-19T01:19:56.534+0000",
    "status": 403,
    "error": "Forbidden",
    "message": "Forbidden",
    "path": "/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c7"
}
```

NOTE Remember, we use the 8072 port because it is the port we defined for the Spring Cloud Gateway in the previous chapter. Specifically, in the gateway-server.yml configuration file located in the Spring Cloud Configuration service repository.

If we tried the same call using the illary.huaylupo user account (password: password1) and its access token, we'd

see a successful call that returns no content and an HTTP Status Code 204 – No Content, that would delete that organization.

At this point, we've looked at how to call and protect a single service (the organization service) with Keycloak. However, often in a microservices environment, we will have multiple service calls used to carry out a single transaction. In these types of situations, we need to ensure that the access token is propagated from service call to service call.

9.4.4 Propagating the access token

To demonstrate propagating a token between services, we're now going to see how to protect our licensing service with Keycloak. Remember, the licensing service calls the organization service to lookup information. The question becomes, how do we propagate the token from one service to another?

We're going to set up a simple example where we're going to have the licensing service call the organization service. If you followed the examples we have been building in the previous chapters, you'd see that both services are running behind a gateway.

Figure 9.26 shows the basic flow of how an authenticated user's token will flow through the gateway, the licensing service, and then down to the organization service.

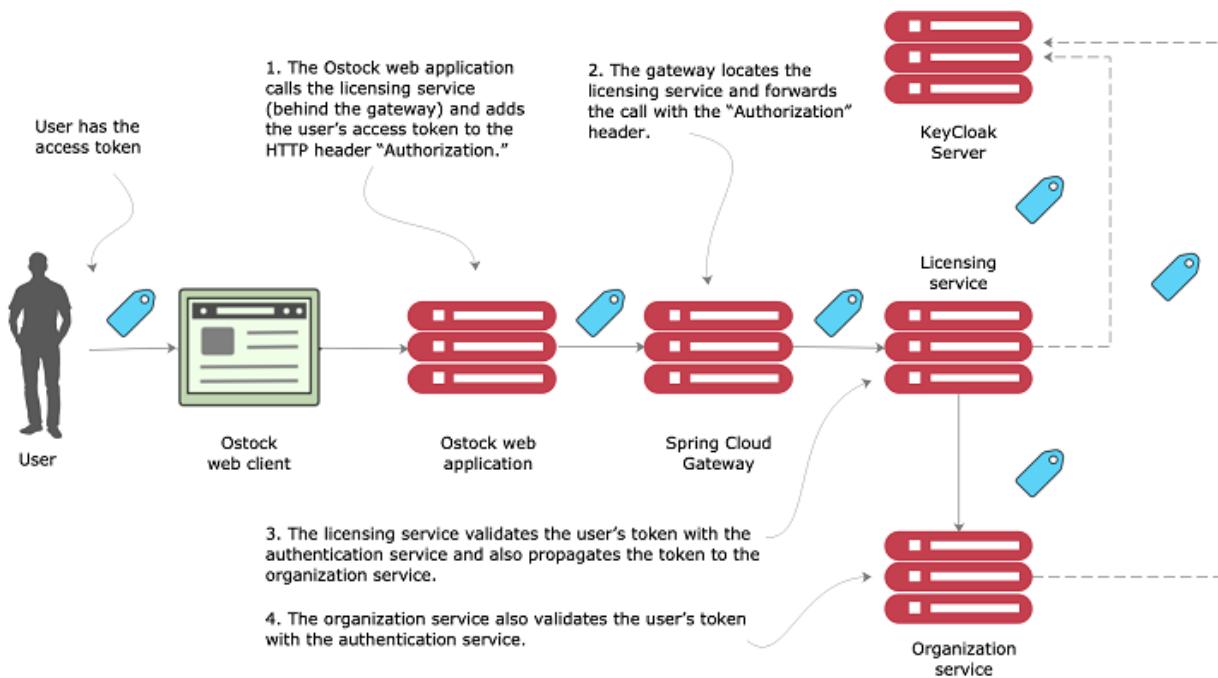


Figure 9.26 The access token has to be carried throughout the entire call chain.

The following activity occurs in figure 9.26:

1. The user has already authenticated against the Keycloak server and places a call to the Ostock web application. The user's access token is stored in the user's session. The Ostock web application needs to retrieve some licensing data and call the licensing service REST endpoint. As part of the call to the licensing REST endpoint, the Ostock web application will add the access token via the HTTP Header "Authorization." The licensing service is only accessible behind a Spring Cloud services gateway.
2. The gateway will look up the licensing service endpoint and then forward the call onto one of the licensing service's servers. The services gateway will need to

copy the "Authorization" HTTP header from the incoming call and ensure that the "Authorization" HTTP header is forwarded onto the new endpoint.

3. The licensing service will receive the incoming call. Because the licensing service is a protected resource, the licensing service will validate the token with the KeyCloak server and then check the user's roles for the appropriate permissions. As part of its work, the licensing service invokes the organization service. In doing this call, the licensing service needs to propagate the user's access token to the organization service.
4. When the organization service receives the call, it will again take the "Authorization" HTTP header token and validate the token with the KeyCloak server.

To implement these steps, we need to make several changes in our code. If we don't do them, we will get the following error while retrieving the organization info from the licensing service.

```
message": "401 : {[status:401,  
error: Unauthorized  
message: Unauthorized,  
path: /v1/organization/d898a142-de44-466c-8c88-9ceb2c2429d3}]
```

The first step is to modify the gateway to propagate the access token to the licensing service. By default, the gateway won't forward sensitive HTTP headers such as Cookie, Set-Cookie, and Authorization to downstream services. To allow the propagation of the "Authorization" HTTP header, we need to add the following filter for each

route on the gateway-server.yml configuration file located in the Spring Cloud Config repository:

- ```
- RemoveRequestHeader= Cookie,Set-Cookie
```

This configuration is a blacklist of the sensitive headers that the gateway will keep from being propagated to a downstream service. The absence of the Authorization value in the previous list means that it will allow it through. If we don't set this configuration property at all, the gateway will automatically block all three values from being propagated (Cookie, Set-Cookie, and Authorization).

Next, we need to configure our licensing service to have the Keycloak and the Spring Security dependencies and set up any authorization rules we want for the service and finally add the Keycloak properties to the application properties file in the configuration server.

## ***CONFIGURING THE LICENSING SERVICE***

The first step will be adding the following maven dependencies shown in code listing 9.7 to our licensing service pom.xml file.

### **Listing 9.7 Configuring Keycloak and Spring security dependencies**

```
//Rest of pom.xml removed for conciseness
<dependency>
<groupId>org.keycloak</groupId>
```

```

<artifactId>keycloak-spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
</dependencies>
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>org.keycloak.bom</groupId>
<artifactId>keycloak-adapter-bom</artifactId>
<version>11.0.2</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

```

The next step is to protect the licensing service so that it can only be accessed by an authenticated user. The following code listing 9.8 shows the /licensing-service/src/main/java/com/optimagrowth/license/config/SecurityConfig.java class.

### **Listing 9.8 Restricting access to the authenticated users only**

```

//Rest of class removed for conciseness
@Configuration
@EnableWebSecurity
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {
super.configure(http);
http.authorizeRequests()
.anyRequest().authenticated();
http.csrf().disable();
}
@Override
protected void configure(HttpSecurity http) throws Exception {
super.configure(http);
http.authorizeRequests()
.anyRequest().authenticated();
}

```

```

 http.csrf().disable();
}
@.Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
 KeycloakAuthenticationProvider keycloakAuthenticationProvider =
 keycloakAuthenticationProvider();
 keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new
 SimpleAuthorityMapper());
 auth.authenticationProvider(keycloakAuthenticationProvider);
}
@Bean
@Override
protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
 return new RegisterSessionAuthenticationStrategy(new SessionRegistryImpl());
}
@Bean
public KeycloakConfigResolver KeycloakConfigResolver() {
 return new KeycloakSpringBootConfigResolver();
}
}

```

The final step in this process will be adding the KeyCloak configuration in our licensing service's licensing-service.properties file. The following code listing 9.9 shows how the properties file of the licensing service should look like

### **Listing 9.9 Configure KeyCloak in licensing-service.properties file**

```

//Rest of properties removed for conciseness
keycloak.realm = spmia-realm
keycloak.auth-server-url = http://keycloak:8080/auth
keycloak.ssl-required = external
keycloak.resource = ostock
keycloak.credentials.secret = 5988f899-a5bf-4f76-b15f-f1cd0d2c81ba
keycloak.use-resource-role-mappings = true
keycloak.bearer-only = true

```

**NOTE** Remember, I explained these steps the previous section 9.3 "Protecting the organization service using KeyCloak".

Now that we have the gateway changes to propagate the authorization header and the licensing service set up, we can

move on with our final step to propagate the access token.

All we need to do for this final step is to modify how the code in the licensing service calls the organization service. We need to ensure that the “Authorization” HTTP header is injected into the application call out to the Organization service. Without Spring Security, we’d have to write a servlet filter to grab the HTTP header of the incoming licensing service call and then manually add it to every outbound service call in the licensing service. Keycloak provides a new Rest Template class that supports these calls. The class is called KeycloakRestTemplate. To use the KeycloakRestTemplate class, we first need to expose it as a bean that can be auto-wired into a service calling another protected service. We do this in the /licensing-service/src/main/java/com/optimagrowth/license/config/SecurityConfig.java by adding the following code show in code listing 9.10:

### **Listing 9.10 Exposing the KeycloakRestTemplate in the SecurityConfig.java class.**

```
package com.optimagrowth.license.service.client;
//Rest of class removed for conciseness
@ComponentScan(basePackageClasses = KeycloakSecurityComponents.class)
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {
 ...
 @Autowired
 public KeycloakClientRequestFactory keycloakClientRequestFactory;
 @Bean #A
 @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
 public KeycloakRestTemplate keycloakRestTemplate() {
 return new KeycloakRestTemplate(keycloakClientRequestFactory);
 }
 ...
}
```

To see the KeycloakRestTemplate class in action, we can look in the /licensing-service/src/main/java/com/optimagrowth/license/service/client/OrganizationRestTemplateClient.java class. The following code listing 9.11 shows how KeycloakRestTemplate is auto-wired into this class.

### **Listing 9.11 Using the KeycloakRestTemplate to propagate the access token**

```
package com.optimagrowth.license.service.client;
//Imports removed for conciseness
@Component
public class OrganizationRestTemplateClient {
@.Autowired
KeycloakRestTemplate restTemplate; #A
public Organization getOrganization(String organizationId){
ResponseEntity<Organization> restExchange = #B
restTemplate.exchange(
"http://gateway:8072/organization/v1/organization/{organizationId}",
HttpMethod.GET,
null, Organization.class, organizationId);
return restExchange.getBody();
}
}
```

#A The KeycloakRestTemplate is a drop-in replacement for the standard RestTemplate and handles the propagation of the access token.

#B The invocation of the organization service is done in the exact same manner as a standard RestTemplate. In this particular scenario, we are going to point to the gateway server.

To test this code, you can request a service in the licensing service that calls over the organization service to retrieve the data. For example, the following service retrieves the data of a specific license and retrieves the associated organization's information.

```
http://localhost:8072/license/v1/organization/d898a142-de44-466c-8c88-9ceb2c2429d3/license/f2a9c9d4-d2c0-44fa-97fe-724d77173c62
```

Figure 9.27 shows the output of that call.

The screenshot shows the Postman interface with a successful API call. The top bar indicates a GET request to `http://localhost:8072/license/v1/organization/d898a142-de44-466c-8c88-9ceb2c2429d3/license/f2a9c9d4-d2...`. The 'Authorization' tab is selected, showing a 'Bearer Token' dropdown with the value `eyJhbGciOiJSUzI1NiIsInR5cClgOiAiSldeUliwia2IkliA6ICJ3LXjhdmV4NmNuW...`. A note says: 'Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. [Learn more about variables](#)'.

The 'Body' tab is selected, showing the JSON response:

```
1 {
2 "licenseId": "f2a9c9d4-d2c0-44fa-97fe-724d77173c62",
3 "description": "Software Product",
4 "organizationId": "d898a142-de44-466c-8c88-9ceb2c2429d3",
5 "productName": "Ostock",
6 "licenseType": "complete",
7 "comment": "I AM DEV",
8 "organizationName": "OptimaGrowth",
9 "contactName": "Admin",
10 "contactPhone": "888888888",
11 "contactEmail": "illaryhs@gmail.com",
12 "_links": {
13 "self": {
14 "href": "http://172.20.0.7:8080/v1/organization/d898a142-de44-466c-8c88-9ceb2c2429d3/license/
15 f2a9c9d4-d2c0-44fa-97fe-724d77173c62"
16 },
17 "createLicense": {
18 "href": "http://172.20.0.7:8080/v1/organization/{organizationId}/license",
19 "templated": true
20 },
21 "updateLicense": {
22 "href": "http://172.20.0.7:8080/v1/organization/{organizationId}/license",
23 "templated": true
24 }
25 }
26}
```

**Figure 9.27 Pass the access token from the licensing service to the organization service.**

## 9.4.5 Parsing a custom filled out of a JWT

We're going to turn to our gateway for an example of how to parse out a custom field in the JWT token. Specifically, we're going to modify the TrackingFilter class we introduced in chapter 8 to decode the preferred\_username field out of the JWT token flowing through the gateway.

To do this, we're going to pull in a JWT parser library and add to the gateway server's pom.xml file. Multiple token parsers are available, and I chose apache commons-codec and the org.json.json to parse the json body.

```
<dependency>
<groupId>commons-codec</groupId>
<artifactId>commons-codec</artifactId>
</dependency>
<dependency>
<groupId>org.json</groupId>
<artifactId>json</artifactId>
<version>20190722</version>
</dependency>
```

Once the library is added, we can add a new method to our /gatewayserver/src/main/java/com/optimagrowth/gateway/filters/TrackingFilter.java class called getUsername(). The following listing 9.12 shows this new method.

### **Listing 9.12 Parsing the preferred\_username out of our JWT Token**

```
//Rest of the code removed for conciseness
private String getUsername(HttpServletRequest requestHeaders){
String username = "";
if (filterUtils.getAuthToken(requestHeaders)!=null){
String authToken = filterUtils.getAuthToken(requestHeaders).replace("Bearer ","");
#A
JSONObject jsonObj = decodeJWT(authToken);
try {
username = jsonObj.getString("preferred_username"); #B
}catch(Exception e) {logger.debug(e.getMessage());}
}
return username;
```

```
 }
 private JSONObject decodeJWT(String JWTToken) {
 String[] split_string = JWTToken.split("\\.");
 String base64EncodedBody = split_string[1]; #C
 Base64 base64Url = new Base64(true);
 String body = new String(base64Url.decode(base64EncodedBody));
 JSONObject jsonObj = new JSONObject(body); #D
 return jsonObj;
 }
```

#A Parse out the token out of the Authorization HTTP header.  
#B Pull the preferred\_username out of the JWT.  
#C Use Base64 class to parse out the token, passing in the signing key used to sign the token.  
#D Parse the JWT body into a JSON Object to retrieve the preferred\_username.

To make this example work, we have to make sure the /gatewayserver/src/main/java/com/optimagrowth/gateway/filters/FilterUtils.java AUTH\_TOKEN variable is set to “Authorization”. Like the following code.

```
public static final String AUTH_TOKEN = "Authorization";
```

Once the getUsername() function is implemented, we added a System.out.println to the filter() method on the TrackingFilter to print out the preferred\_username parsed from our JWT token that's flowing through the gateway. So, whenever we make a call to the gateway, we are going to see the preferred\_username in the console output.

**NOTE** Remember, when you make this call, you still need to set up all the HTTP form parameters and the HTTP authorization header to include the Authorization header and the JWT token

If everything was successful, you should see the following System.out.println in your console log.

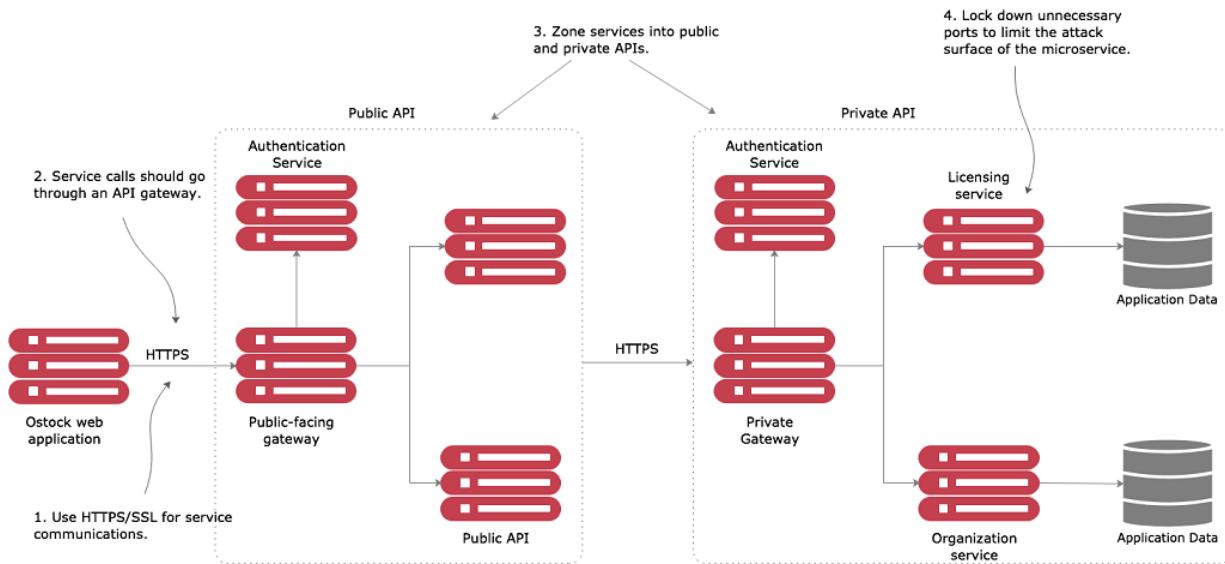
```
tmx-correlation-id found in tracking filter: 26f2b2b7-51f0-4574-9d84-07e563577641.
The authentication name from the token is : illary.huaylupo
```

## 9.5 Some closing thoughts on microservice security

While this chapter has introduced you to the OpenID, OAuth2, KeyCloak specification and how you can use Spring Cloud security, and KeyCloak to implement an authentication and authorization service, KeyCloak is only one piece of the microservice security puzzle. As you build your microservices for production use, you should be building your microservices security around the following practices:

1. Use HTTPS/Secure Sockets Layer (SSL) for all service communication.
2. All service calls should go through an API gateway.
3. Zone your services into a public API and private API.
4. Limit the attack surface of your microservices by locking down unneeded network ports.

Figure 9.15 shows how these different pieces fit together. Each of the bulleted items in the list maps to the numbers in figure 9.28.



**Figure 9.28 A microservice security architecture is more than implementing Authentication and Authorization.**

Let's examine each of the topic areas enumerated in the previous list and diagrams in more detail.

## ***USE HTTPS/SECURE SOCKETS LAYER (SSL) FOR ALL SERVICE COMMUNICATION***

In all the code examples in this book, you've been using HTTP because HTTP is a simple protocol and doesn't require setup on every service before you can start using the service.

In a production environment, your microservices should communicate only through the encrypted channels provided

through HTTPS and SSL. The configuration and setup of the HTTPS can be automated through your DevOps scripts.

## ***USE A SERVICES GATEWAY TO ACCESS YOUR MICROSERVICES***

The individual servers, service endpoints, and ports your services are running on should never be directly accessible to the client. Instead, use a services gateway to act as an entry point and gatekeeper for your service calls. Configure the network layer on the operating system or container your microservices are running in to only accept traffic from the services gateway.

Remember, the services gateway can act as a policy enforcement point (PEP) that can be enforced against all services. Putting service calls through a services gateway allows you to be consistent in how you're securing and auditing your services. A service gateway also allows you to lock down what port and endpoints you're going to expose to the outside world.

## ***ZONE YOUR SERVICES INTO A PUBLIC API AND PRIVATE API***

Security, in general, is all about building layers of accessing and enforcing the concept of least privilege. Least privilege is the concept that a user should have the bare minimum network access and privileges to do their day-to-day job. To this end, you should implement least-privilege by separating your services into two distinct zones: public and private.

The public zone contains the public APIs that will be consumed by clients (Ostock application). Public API microservices should carry out narrow tasks that are workflow-oriented. Public API microservices tend to be service aggregators, pulling data, and carrying out tasks across multiple services.

Public microservices should be behind their own services gateway and have their own authentication service for performing authentication and authorization. Access to public services by client applications should go through a single route protected by the services gateway. Also, the public zone should have its own authentication service.

The private zone acts as a wall to protect your core application functionality and data. The private zone should only be accessible through a single well-known port and should be locked down to only accept network traffic from the network subnet that the private services are running. The private zone should have its own services gateway and authentication service. Public API services should authenticate against the private zone's authentication service. All application data should at least be in the private zone's network subnet and only accessible by microservices residing in the private zone.

## ***LIMIT THE ATTACK SURFACE OF YOUR MICROSERVICES BY LOCKING DOWN UNNEEDED NETWORK PORTS***

Many developers don't take a hard look at the absolute minimum number of ports they need to open for their services to function. Configure the operating system your service is running on to only allow the inbound and outbound access to ports needed by your service or a piece of infrastructure needed by your service (monitoring, log aggregation).

Don't focus only on inbound access ports. Many developers forget to lock down their outbound ports. Locking down your outbound ports can prevent data from being leaked off your service if an attacker has compromised the service itself. Also, make sure you look at network port access in both your public and private API zones.

## 9.6 Summary

- OAuth2 is a token-based authorization framework.
- OAuth2 offers different mechanisms for protecting web services calls. These mechanisms are called grants.
- OpenID Connect (OIDC) is a layer on top of the OAuth2 framework that provides authentication and profile information about who is logged in to the application (Identity).
- KeyCloak is an open-source Identity and Access management software aimed at current services and applications.
- The main objective of Keycloak is to facilitate the protection of the services and applications with little or no code.
- Each application can have its own application name and secret key in Keycloak.
- Each service must define what actions a role can take.

- Spring Cloud Security supports the JSON Web Token (JWT) specification.
- With JWT, we can inject custom fields into the specification.
- Securing our microservices involves more than just using Authentication and Authorization.
- We should use HTTPS to encrypt all calls between services.
- Use a services gateway to narrow the number of access points a service can be reached through.
- Limit the attack surface for a service by limiting the number of inbound and outbound ports on the operating system that the service is running on.

# 10 Event-driven architecture with Spring Cloud Stream

This chapter covers

- Understanding event-driven architecture processing and its relevance to microservices
- Using Spring Cloud Stream to simplify event processing in your microservices
- Configuring Spring Cloud Stream
- Publishing messages with Spring Cloud Stream and Kafka
- Consuming messages with Spring Cloud Stream and Kafka
- Implementing distributed caching with Spring Cloud Stream, Kafka, and Redis

Human beings are always in a state of motion, interacting with their environment while sending out and receiving information from the things around them. Typically, these conversations aren't synchronous, linear, or narrowly defined to a request-response model. It's something more like a message-driven, where we're constantly sending and receiving messages. As we receive messages, we react to

those messages, while often interrupting the primary task that we're working on.

This chapter is about how to design and implement our Spring-based microservices to communicate with other microservices using asynchronous messages. Using asynchronous messages to communicate between applications isn't new. What's new is the concept of using messages to communicate events representing changes in state. This concept is called Event-Driven Architecture (EDA). It's also known as Message Driven Architecture (MDA). What an EDA-based approach allows us to do is to build highly decoupled systems that can react to changes without being tightly coupled to specific libraries or services. When combined with microservices, EDA allows us to quickly add new functionality into our application by merely having the service listen to the stream of events (messages) being emitted by our application.

The Spring Cloud project has made it trivial to build messaging-based solutions through the Spring Cloud Stream sub-project. Spring Cloud Stream allows us to easily implement message publication and consumption while shielding our services from the implementation details associated with the underlying messaging platform.

## **10.1 The case for messaging, EDA, and microservices**

Why is messaging important in building microservice-based applications? To answer that question, let's start with an

example. We're going to use the two services we've been using throughout the book: our licensing and organization services. Let's imagine that after these services are deployed to production, we find that the licensing service calls are taking an exceedingly long time when doing a lookup of organization information from the organization service. When we look at the usage patterns of the organization data, we find that the organization data rarely changes and that most of the data reads from the organization service are done by the primary key of the organization record. If we could cache the reads for the organization data without having to incur the cost of accessing a database, we could significantly improve the response time of the licensing service calls.

To implement a caching solution, we need to consider the following three core requirements.

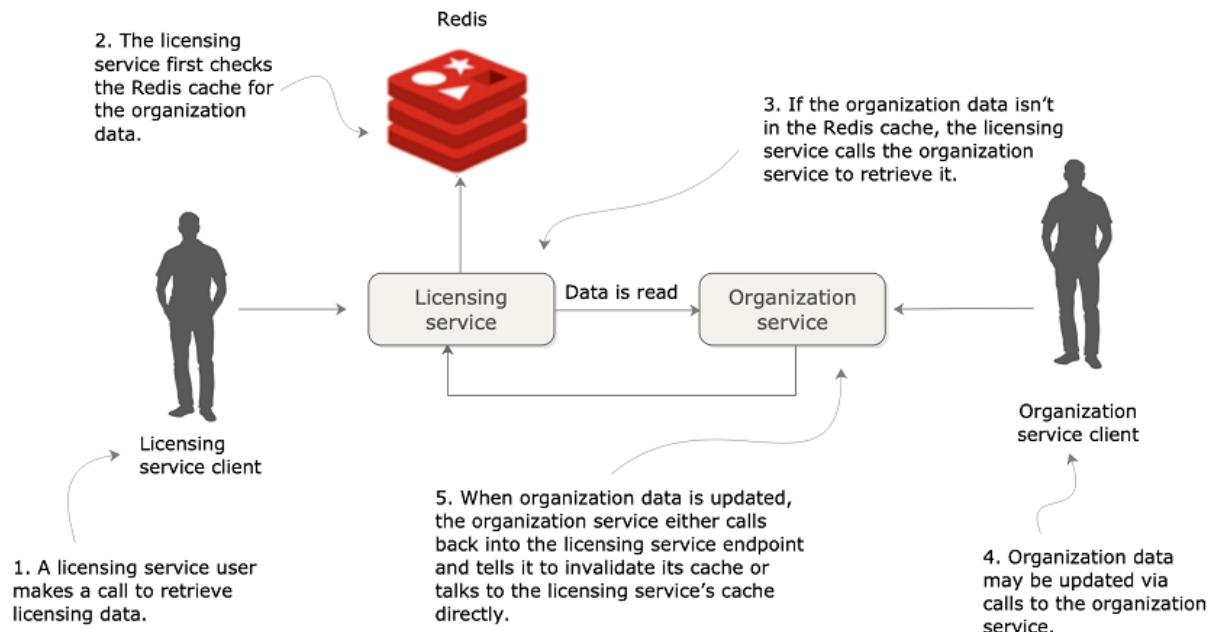
- 1. The cached data needs to be consistent across all instances of the licensing service.** This means that we can't cache the data locally within the licensing service because we want to guarantee that the same organization data is read regardless of the service instance hitting it.
- 2. We cannot cache the organization data within the memory of the container hosting the licensing service.** The run-time container hosting our service is often restricted in size and can access data using different access patterns. A local cache can introduce complexity because we have to guarantee our local cache is synced with all of the other services in the cluster.

3. When an organization record changes via an update or delete, we want the licensing service to recognize that there has been a state change in the organization service. The licensing service should then invalidate any cached data it has for that specific organization and evict it from the cache.

Let's look at two approaches to implement these requirements. The first approach will implement the above requirements using a synchronous request-response model. When the organization state changes, the licensing and organization services communicate back and forth via their REST endpoints. The second approach will have the organization service emit an asynchronous event (message) that will communicate that the organization service data has changed. With the second approach, the organization service will publish a message to a queue that an organization record has been updated or deleted. The licensing service will listen with the intermediary, see that an organization event has occurred, and clear the organization data from the cache.

### **10.1.1 Using synchronous request-response approach to communicate state change**

For our organization data cache, we're going to use Redis (<https://redis.io/>) a distributed key-value store database. Figure 10.1 provides a high-level overview of how to build a caching solution using a traditional synchronous, request-response programming model.



**Figure 10.1 In a synchronous request-response model, tightly coupled services introduce complexity and brittleness.**

In figure 10.1, when a user calls the licensing service, the licensing service will also need to look up the organization data. The licensing service will first check to retrieve the desired organization by its ID from the Redis cluster. If the licensing service can't find the organization data, it will call the organization service using a REST-based endpoint and then store the data returned in Redis, before returning the organization data back to the user. Now, if someone updates or deletes the organization record using the organization service's REST endpoint, the organization service will need to call an endpoint exposed on the licensing service, telling it to invalidate the organization data in its cache. In figure 10.1, if

we look at where the organization service calls back into the licensing service to tell it to invalidate the Redis cache, we can see at least three problems:

1. The organization and licensing services are tightly coupled.
2. The coupling has introduced brittleness between the services. If the licensing service endpoint for invalidating the cache changes, the organization service has to change.
3. The approach is inflexible because we can't add new consumers of the organization data even without modifying the code on the organization service to know that it has called the other service to let it know about the change.

## ***TIGHT COUPLING BETWEEN SERVICES***

In figure 10.1, we can see tight coupling between the licensing and the organization service. The licensing service always had a dependency on the organization service to retrieve data. However, by having the organization service directly communicate back to the licensing service whenever an organization record has been updated or deleted, we've introduced coupling back from the organization service to the licensing service.

For the data in the Redis cache to be invalidated, the organization service either needs an endpoint on the licensing service exposed that can be called to invalidate its Redis cache, or the organization service has to talk directly

to the Redis server owned by the licensing service to clear the data in it.

Having the organization service talk to Redis has its own problems because we're talking to a data store owned directly by another service. In a microservice environment, this is a big no-no. While one can argue that the organization data rightly belongs to the organization service, the licensing service is using it in a specific context and could be potentially transforming the data or have built business rules around it. Having the organization service talking directly to the Redis service can accidentally break rules the team owning the licensing service has implemented.

## ***BRITTLENESS BETWEEN THE SERVICES***

The tight coupling between the licensing service and the organization service has also introduced brittleness between the two services. If the licensing service is down or running slowly, the organization service can be impacted because the organization service is now communicating directly with the licensing service. Again, if we had the organization service talk directly to the licensing service's Redis data store, we've now created a dependency between the organization service and Redis. In this scenario, any problems with the shared Redis server now have the potential to take down both services.

## ***INFLEXIBLE IN ADDING NEW CONSUMERS TO CHANGES IN THE***

## **ORGANIZATION SERVICE**

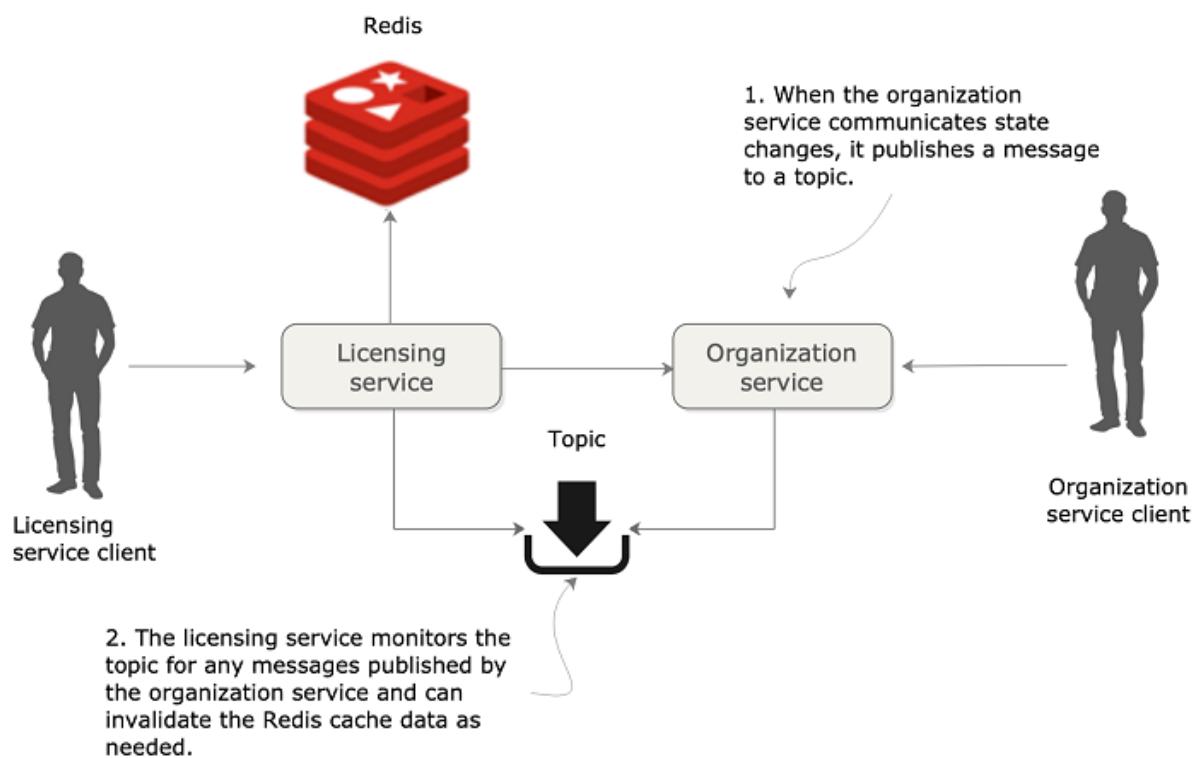
The last problem with this architecture is that it's inflexible. With the model in figure 10.1, if we had another service that was interested in when the organization data changes, we'd need to add another call from the organization service to that other service. This means a code change and redeployment of the organization service. If we use the synchronous, request-response model for communicating state change, we start to see almost a web-like pattern of dependency between our core services in our application and other services. The centers of these webs become our major points of failure within our application.

### **Another kind of coupling**

While messaging adds a layer of indirection between our services, we can still introduce tight coupling between two services using messaging. Later in the chapter, we're going to send messages between the organization and licensing service. These messages are going to be serialized and de-serialized to a Java object using JSON formatting for the message. Changes to the structure of the JSON message can cause problems when converting back and forth to Java if the two services don't gracefully handle different versions of the same message type. JSON doesn't natively support versioning. However, we can use Apache Avro (<https://avro.apache.org/>) if we need versioning. Avro is a binary protocol that has versioning built into it. Spring Cloud Stream does support Apache Avro as a messaging protocol. However, using Avro is outside the scope of this book, but I did want to make you aware that it does help if you truly need to worry about message versioning.

### **10.1.2 Using messaging to communicate state changes between services**

We're going to inject a topic between the licensing and organization service with a messaging approach. This topic won't be used to read data from the organization service but will instead be used by the organization service to publish when any state changes within the organization data managed by the organization service occur. Figure 10.2 demonstrates this approach.



**Figure 10.2 As organization state changes, messages will be written to a message queue that sits between the two services.**

In the model in figure 10.2, every time organization data changes, the organization service publishes a message out to a topic. The licensing service is monitoring the topic for

messages, and when a message comes in, it clears the appropriate organization record out of the Redis cache. When it comes to communicating state, the message queue acts as an intermediary between the licensing and organization service. This approach offers four benefits:

- Loose coupling
- Durability
- Scalability
- Flexibility

## ***LOOSE COUPLING***

A microservices application can be composed of dozens of small and distributed services that have to interact with each other and are interested in the data managed by one another. As we saw with the synchronous design proposed earlier, a synchronous HTTP response creates a hard dependency between the licensing and organization service. We can't eliminate these dependencies completely, but we can try to minimize dependencies by only exposing endpoints that directly manage the data owned by the service. A messaging approach allows us to decouple the two services because when it comes to communicating state changes, neither service knows about each other. When the organization service needs to publish a state change, it writes a message to a queue. The licensing service only knows that it gets a message; it has no idea who has published the message.

## ***DURABILITY***

The presence of the queue allows us to guarantee that a message will be delivered even if the consumer of the service is down. The organization service can keep publishing messages even if the licensing service is unavailable. The messages will be stored in the queue and will stay there until the licensing service is available. Conversely, with the combination of a cache and the queuing approach, if the organization service is down, the licensing service can degrade gracefully because at least part of the organization data will be in its cache. Sometimes old data is better than no data.

## **SCALABILITY**

Since messages are stored in a queue, the sender of the message doesn't have to wait for a response back from the consumer of the message. They can go on their way and continue their work. Likewise, if a consumer reading a message of the queue isn't processing messages fast enough, it's a trivial task to spin up more consumers and have them process those messages of the queue. This scalability approach fits well within a microservices model because one of the things I've been emphasizing through this book is that it should be trivial to spin up new instances of a microservice and have that additional microservice become another service that can process work of the message queue holding the messages. This is an example of scaling horizontally. Traditional scaling mechanisms for reading messages of a queue involved increasing the number of threads that a message consumer could process at one time. Unfortunately, with this approach, we were ultimately limited by the number of CPUs available to the

message consumer. A microservice model doesn't have this limitation because we're scaling by increasing the number of machines hosting the service consuming the messages.

## **FLEXIBILITY**

The sender of a message has no idea who is going to consume it. This means we can easily add new message consumers (and new functionality) without impacting the original sending service. This is an extremely powerful concept because new functionality can be added to an application without having to touch existing services. Instead, the new code can listen for events being published and react to them accordingly.

### **10.1.3 Downsides of a messaging architecture**

Like any architectural model, a messaging-based architecture has tradeoffs. A messaging-based architecture can be complicated and requires the development team to pay close attention to several key things, including

- Message handling semantics
- Message visibility
- Message choreography

## **MESSAGE HANDLING SEMANTICS**

Using messages in a microservice-based application requires more than understanding how to publish and consume

messages. It requires us to understand how our application will behave based on the order messages are consumed and what happens if a message is processed out of order. For example, if we have strict requirements that all orders from a single customer must be processed in the order they are received, we're going to have to set up and structure our message handling differently than if every message can be consumed independently of one another.

It also means that if we're using messaging to enforce strict state transitions of our data, we need to think about designing our applications to take into consideration scenarios where a message throws an exception, or an error is processed out of order. If a message fails, do we retry processing the error, or do we let it fail? How do we handle future messages related to that customer if one of the customer messages fails? Again, these are all topics to think through.

## **MESSAGE VISIBILITY**

Using messages in our microservices often means a mix of synchronous service calls and processing in services asynchronously. The asynchronous nature of messages means they might not be received or processed in close proximity to when the message is published or consumed. Also, having things like a correlation ID for tracking a user's transactions across web service invocations and messages is critical to understanding and debugging what's going on in our application. As you may remember from chapter 8, a correlation ID is a unique number that's generated at the start of a user's transaction and passed along with every

service call. It should also be passed with every message that's published and consumed.

## **MESSAGE CHOREOGRAPHY**

As alluded to in the section on message visibility, messaging-based applications make it more difficult to reason through the business logic of their applications because their code is no longer being processed in a linear fashion with a simple block request-response model. Instead, debugging message-based applications can involve wading through the logs of several different services where the user transactions can be executed out of order and at different times.

**NOTE** Messaging can be complicated but powerful, and with the previous sections, I didn't mean to scare you away from using messaging in your applications. Instead, my goal is to highlight that using messaging in your services requires forethought. A positive side of using messaging is that business themselves work asynchronously, so in the end we are modeling the business more closely.

## **10.2 Introducing Spring Cloud Stream**

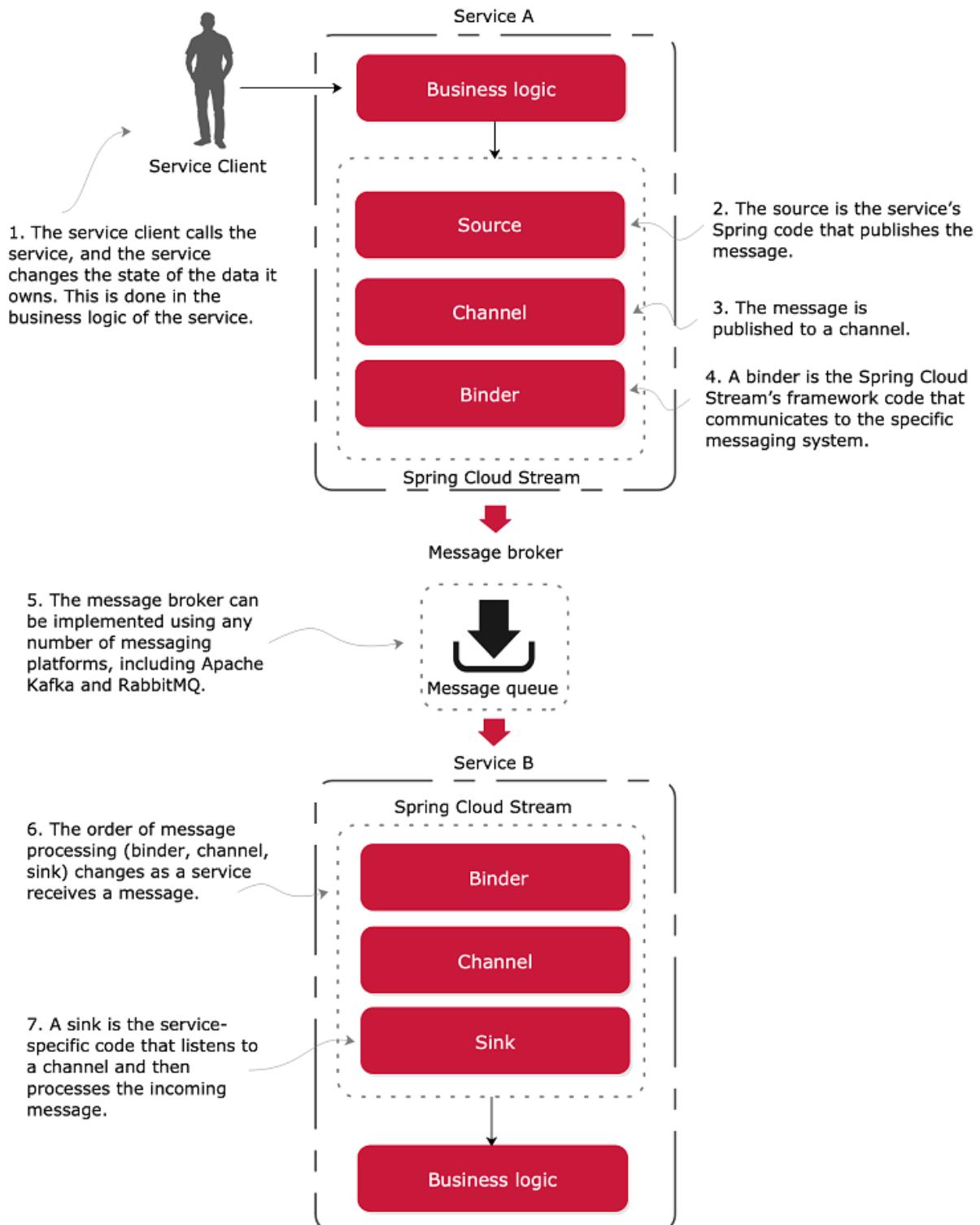
Spring Cloud makes it easy to integrate messaging into our Spring-based microservices. It does this through the Spring Cloud Stream project (<https://spring.io/projects/spring-cloud-stream>). The Spring Cloud Stream project is an annotation-driven framework that allows us to easily build message publishers and consumers in our Spring application.

Spring Cloud Stream also allows us to abstract away the implementation details of the messaging platform we're using. Multiple message platforms can be used with Spring Cloud Stream (including the Apache Kafka project and RabbitMQ), and the platform's implementation-specific details are kept out of the application code. The implementation of message publication and consumption in your application is done through platform-neutral Spring interfaces.

**NOTE** For this chapter, we're going to use a message bus called Kafka (<https://kafka.apache.org/>). Kafka is a highly performant message bus that allows us to asynchronously send streams of messages from one application to one or more other applications. Written in Java, Kafka has become the de facto message bus for many cloud-based applications because it's highly reliable and scalable. Spring Cloud Stream also supports the use of RabbitMQ as a message bus. Both Kafka and RabbitMQ are messaging platforms.

To understand Spring Cloud Stream, let's begin with a discussion of the Spring Cloud Stream architecture and familiarize ourselves with the terminology of Spring Cloud Stream. The new terminology involved can be somewhat overwhelming if you've never worked with a messaging-based platform before.

Let's begin our discussion by looking at the Spring Cloud Stream architecture through the lens of two services communicating via messaging. One service will be the message publisher, and one service will be the message consumer. Figure 10.3 shows how Spring Cloud Stream is used to facilitate this message passing.



**Figure 10.3 As a message is published and consumed, it flows through a series of Spring Cloud Stream components that abstract away the underlying messaging platform.**

With the publication and consumption of a message in Spring Cloud, four components are involved in publishing and consuming the message:

- Source
- Channel
- Binder
- Sink

## **SOURCE**

When a service gets ready to publish a message, it will publish the message using a source. A source is a Spring annotated interface that takes a Plain Old Java Object (POJO) that represents the message to be published. A source takes the message, serializes it (the default serialization is JSON), and publishes the message to a channel.

## **CHANNEL**

A channel is an abstraction over the queue that's going to hold the message after it has been published by the message producer or consumed by a message consumer. In other words, we can describe a channel as a queue used to send and receive messages. A channel name is always associated with a target queue name. However, that queue name is never directly exposed to the code. Instead, the

channel name is used in the code, which means that we can switch the queues the channel reads or writes from by changing the application's configuration, not the application's code.

## ***BINDER***

The binder is part of the Spring Cloud Stream framework. It's the Spring code that talks to a specific message platform. The binder part of the Spring Cloud Stream framework allows us to work with messages without having to be exposed to platform-specific libraries and APIs for publishing and consuming messages.

## ***SINK***

In Spring Cloud Stream, when a service receives a message from a queue, it does it through a sink. A sink listens to a channel for incoming messages and de-serializes the message back into a plain old Java object (POJO). From there, the message can be processed by the business logic of the Spring service.

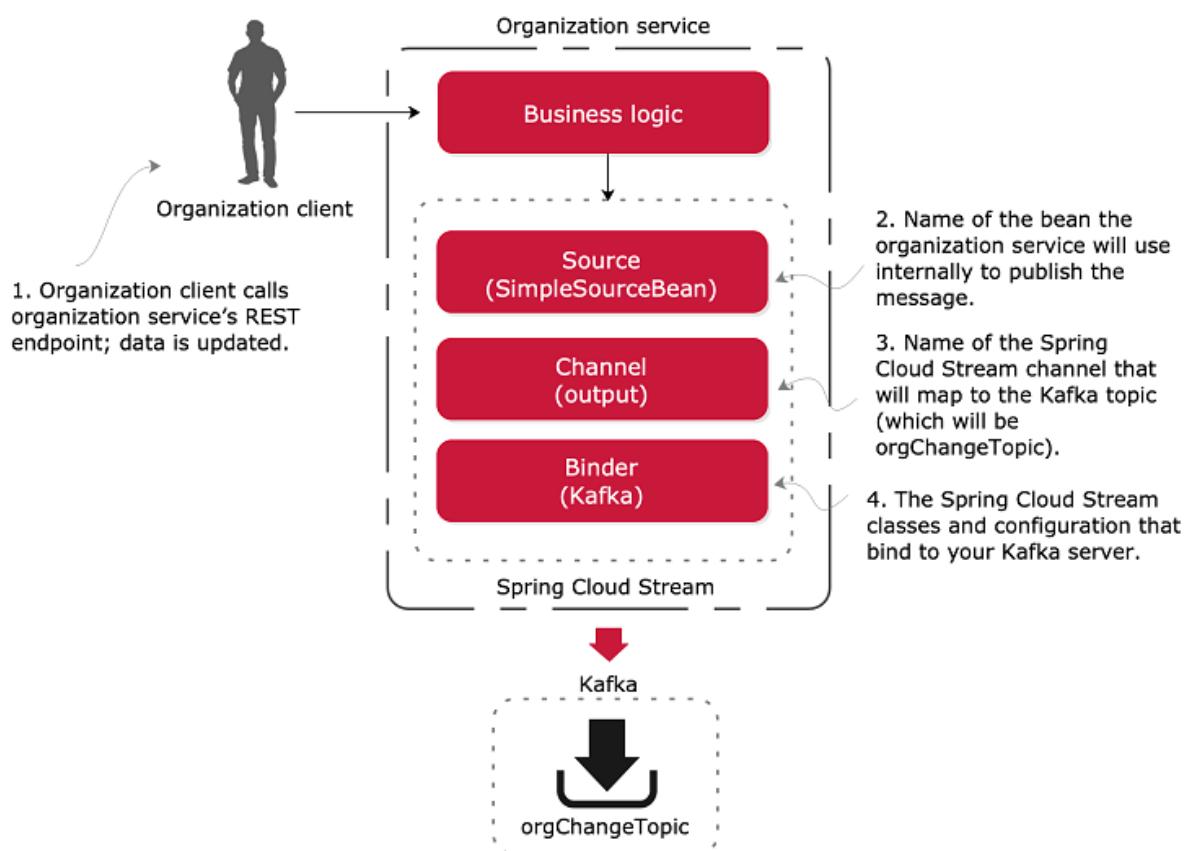
## **10.3 Writing a simple message producer and consumer**

Now that we've walked through the essential components in Spring Cloud Stream, let's look at a simple Spring Cloud Stream example. For the first example, we're going to pass a message from our organization service to our licensing

service. The only thing we'll do with the message in the licensing service is to print a log message to the console.

In addition, because we're only going to have one Spring Cloud Stream source (the message producer) and sink (message consumer) in this example, we're going to start the example with a few simple Spring Cloud shortcuts that will make setting up the source in the organization service and a sink in the licensing service trivial.

Figure 10.4 highlights the message producer and builds on the general Spring Cloud Stream architecture from figure 10.3.



**Figure 10.4 When organization service data changes it will publish a message to orgChangeTopic.**

### 10.3.1 Configuring Apache Kafka and Redis in Docker

In this section, I will explain you how to add the Kafka and Redis service in our Docker environment. To achieve this let's start by adding the following code shown in code listing 9.1 to our docker-compose.yml file.

#### **Listing 10.1 Kafka and Redis service in docker-compose.yml**

```
Rest of docker-compose.yml removed for conciseness
...
zookeeper:
 image: wurstmeister/zookeeper:latest
 ports:
 - 2181:2181
 networks:
 backend:
 aliases:
 - "zookeeper"
kafkaserver:
 image: wurstmeister/kafka:latest
 ports:
 - 9092:9092
 environment:
 - KAFKA_ADVERTISED_HOST_NAME=kafka
 - KAFKA_ADVERTISED_PORT=9092
 - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
 - KAFKA_CREATE_TOPICS=dresses:1:1,ratings:1:1
 volumes:
 - "/var/run/docker.sock:/var/run/docker.sock"
 depends_on:
 - zookeeper
 networks:
 backend:
 aliases:
 - "kafka"
redisserver:
 image: redis:alpine
 ports:
 - 6379:6379
 networks:
 backend:
```

```
aliases:
- "redis"
```

## 10.3.2 Writing the message producer in the organization service

To highlight how to use topics in our architecture, we're going to begin by modifying the organization service so that every time organization data is added, updated, or deleted, the organization service will publish a message to a Kafka topic indicating that the organization change event has occurred.

The published message will include the organization ID associated with the change event and will also include what action occurred (Add, Update, or Delete).

The first thing we need to do is set up our Maven dependencies in the organization service's Maven pom.xml file. The pom.xml file can be found in the organization service root directory. In the pom.xml, we need to add two dependencies: one for the core Spring Cloud Stream libraries and the other to include the Spring Cloud Stream Kafka libraries:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

**NOTE** It's important to highlight that I'm using Docker to run all the examples, if you want to run this locally you need to be sure to have Apache Kafka installed on your computer. If you are using Docker, you can find the docker-compose.yml file up to date with the Kafka and Zookeper containers at <https://github.com/ihuaylupo/manning-smia/tree/master/chapter10/docker>. Remember to execute the services, you can execute the following commands in the root directory where the parent pom.xml is located: mvn clean package dockerfile:build && docker-compose -f docker/docker-compose.yml up

Once the Maven dependencies have been defined, we need to tell our application that it's going to bind to a Spring Cloud Stream message broker. We do this by annotating the organization service's bootstrap class /organization-service/src/main/java/com/optimagrowth/organization/OrganizationServiceApplication.java with an @EnableBinding annotation. The following code listing 10.2 shows the organization service's OrganizationServiceApplication.java source code.

## **Listing 10.2 The annotated Application.java class**

```
package com.optimagrowth.organization;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
@SpringBootApplication
@RefreshScope
@EnableResourceServer
@EnableBinding(Source.class) #A
public class OrganizationServiceApplication {
 public static void main(String[] args) {
 SpringApplication.run(OrganizationServiceApplication.class, args);
 }
}
```

#A The `@EnableBinding` annotation tells Spring Cloud Stream to bind the application to a message broker.

In listing 10.2, the `@EnableBinding` annotation tells Spring Cloud Stream that we want to bind the service to a message broker. The use of `Source.class` in the `@EnableBinding` annotation tells Spring Cloud Stream that this service will communicate with the message broker via a set of channels defined on the `Source` class. Remember, channels sit above a message queue. Spring Cloud Stream has a default set of channels that can be configured to speak to a message broker.

At this point, we haven't told Spring Cloud Stream what message broker we want the organization service to bind to. We'll get to that shortly. Now, we can go ahead and implement the code that will publish a message.

The first step to create our example to publish the message is to change the `/organization-service/src/main/java/com/optimagrowth/organization/utils/UserContext.java` to make our variables `ThreadLocal`, the following code listing 10.3 shows the code for this class.

### **Listing 10.3 Making our UserContext variables ThreadLocal**

```
package com.optimagrowth.organization.utils;
//Imports removed for conciseness
@Component
public class UserContext {
 public static final String CORRELATION_ID = "tmx-correlation-id";
 public static final String AUTH_TOKEN = "Authorization";
 public static final String USER_ID = "tmx-user-id";
 public static final String ORG_ID = "tmx-org-id";
 private static final ThreadLocal<String> correlationId= new ThreadLocal<String>();
 private static final ThreadLocal<String> authToken= new ThreadLocal<String>();
 private static final ThreadLocal<String> userId = new ThreadLocal<String>();
```

```

private static final ThreadLocal<String> orgId = new ThreadLocal<String>(); #A
public static HttpHeaders getHttpHeaders(){
HttpHeaders httpHeaders = new HttpHeaders();
httpHeaders.set(CORRELATION_ID, getCorrelationId());
return httpHeaders;
}
}

```

#A By defining our variables as ThreadLocal it will give us the ability to store data individually for the current thread. The information set here can only be read by the threat that set the value.

The next step is to create the logic to publish the message. This message publication code can be found in the /organization-service/src/main/java/com/optimagrowth/organization/events/source/SimpleSourceBean.java class. The following code listing 10.4 shows the code for this class.

#### **Listing 10.4 Publishing a message to the message broker**

```

package com.optimagrowth.organization.events.source;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;
import com.optimagrowth.organization.events.model.OrganizationChangeModel;
import com.optimagrowth.organization.utils.UserContext;
@Component
public class SimpleSourceBean {
private Source source;
private static final Logger logger =
LoggerFactory.getLogger(SimpleSourceBean.class);
public SimpleSourceBean(Source source){ #A
this.source = source;
}
public void publishOrganizationChange(ActionEvent action, String organizationId){
logger.debug("Sending Kafka message {} for Organization Id: {}", action,
organizationId);
OrganizationChangeModel change = new OrganizationChangeModel(
OrganizationChangeModel.class.getTypeName(),
action.toString(),
organizationId,
UserContext.getCorrelationId()); #B
source.output().send(MessageBuilder.withPayload(change).build()); #C
}
}

```

```
#A Spring Cloud Stream will inject a Source interface implementation for use by
the service.
#B The message to be published is a Java POJO.
#C When you're ready to send the message, use the send() method from a
channel defined on the Source class.
```

In listing 10.4, we inject the Spring Cloud Source class into our code. Remember, all communication to a specific message topic occurs through a Spring Cloud Stream construct called a channel. A channel is represented by a Java interface class. In this listing, we're using the Source interface. The Source interface is a Spring Cloud defined interface that exposes a single method called output(). The Source interface is a convenient interface to use when our service only needs to publish to a single channel. The output() method returns a class of type MessageChannel. The MessageChannel is how we'll send messages to the message broker. Later in this chapter, I'll show you how to expose multiple messaging channels using a custom interface.

The ActionEnum passed by parameters in the previous method contains the following actions:

```
public enum ActionEnum {
 GET,
 CREATED,
 UPDATED,
 DELETED
}
```

The actual publication of the message occurs in the publishOrganizationChange() method. This method builds a

Java POJO called OrganizationChangeModel. The following code listing 10.5 shows the code for this class.

## **Listing 10.5 Publish Object OrganizationChangeModel**

```
package com.optimagrowth.organization.events.model;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
@Getter @Setter @ToString
public class OrganizationChangeModel {
 private String type;
 private String action;
 private String organizationId;
 private String correlationId;
 public OrganizationChangeModel(String type, String action, String organizationId,
 String correlationId) {
 this.type = type;
 this.action = action;
 this.organizationId = organizationId;
 this.correlationId = correlationId;
 }
}
```

The OrganizationChangeModel is nothing more than a POJO class around three data elements:

- **Action.** This is the action that triggered the event. I've included the action in the message to give the message consumer more context on how it should process an event.
- **Organization ID.** This is the organization ID associated with the event.
- **Correlation ID.** This is the correlation ID the service call that triggered the event. We should always include a correlation ID in our events as it helps greatly with tracking and debugging the flow of messages through our services.

If we go back to the SimpleSourceBean class, we can see that when we're ready to publish the message, we use the

send() method on the MessageChannel class returned from the source.output() method.

```
source.output().send(MessageBuilder.withPayload(change).build());
```

The send() method takes a Spring Message class. We use a Spring helper class called MessageBuilder to take the contents of the OrganizationChangeModel class and convert it to a Spring Message class.

This is all the code we need to send a message. However, at this point, everything should feel a little bit like magic because we haven't seen how to bind our organization service to a specific message queue, let alone the actual message broker. This is all done through configuration. The following code listing 10.6 shows the configuration that does the mapping of our service's Spring Cloud Stream Source to a Kafka message broker and a message topic in Kafka. This configuration information can be localized in our Spring Cloud Config entry for the organization service.

**NOTE** For purposes of this example, I'm using the classpath repository on the Spring Cloud Config, and the configuration for the organization service is found at /configserver/src/main/resources/config/organization-service.properties.

## **Listing 10.6 The Spring Cloud Stream configuration for publishing a message**

```
#Rest removed for conciseness
spring.cloud.stream.bindings.output.destination=orgChangeTopic #A
spring.cloud.stream.bindings.output.content-type=application/json #B
spring.cloud.stream.kafka.binder.zkNodes=localhost #C
spring.cloud.stream.kafka.binder.brokers=localhost #C
```

#A orgChangeTopic is the name of the message queue (or topic) you're going to write messages to.

#B The content-type gives a hint to Spring Cloud Stream of what type of message is going to be sent and received (in this case JSON).

#C The zknodes and brokers property tells Spring Cloud Stream the network location of your Kafka and ZooKeeper.

**NOTE** Zookeeper is a product build by Apache that is used to maintain configuration and naming data and to provide flexible synchronization in distributed systems. In Apache Kafka acts like a centralized service that keeps track of the Kafka cluster nodes and topics configuration.

The configuration in code listing 10.6 looks dense, but it's straightforward. The `spring.cloud.stream.bindings` is the start of the configuration needed for our service to publish to a Spring Cloud Stream message broker. The configuration property `spring.cloud.stream.bindings.output` in the listing maps the `source.output()` channel in listing 10.4 to the `orgChangeTopic` on the message broker we're going to communicate with. It also tells Spring Cloud Stream that messages being sent to this topic should be serialized as JSON. Spring Cloud Stream can serialize messages in multiple formats, including JSON, XML, and the Apache Foundation's Avro format (<https://avro.apache.org/>).

Now that we have the code written that will publish a message via Spring Cloud Stream and the configuration to tell Spring Cloud Stream that it's going to use Kafka as a message broker let's look at where the publication of the message in our organization service actually occurs. This work will be done in the `/organization-service/src/main/java/com/optimagrowth/organization/service/OrganizationService.java` class. The following code listing 10.7 shows the code for this class.

## **Listing 10.7 Publishing a message in the organization service**

```
package com.optimagrowth.organization.service;
//Imports removed for conciseness
@Service
public class OrganizationService {
 private static final Logger logger =
 LoggerFactory.getLogger(OrganizationService.class);
 @Autowired
 private OrganizationRepository repository;
 @Autowired
 SimpleSourceBean simpleSourceBean; #A
 public Organization create(Organization organization){
 organization.setId(UUID.randomUUID().toString());
 organization = repository.save(organization);
 simpleSourceBean.publishOrganizationChange(ActionEnum.CREATED,
 organization.getId()); #B
 return organization;
 }
 //Rest of the code removed for conciseness
}
```

#A Spring autowiring is used to inject the SimpleSourceBean into your organization service.

#B For each method in the service that changes organization data, call simpleSourceBean.publish OrgChange().

## **What data should I put in the message?**

One of the most common questions I get from teams when they're first embarking on their message journey is exactly how much data should go in their messages. My answer is, it depends on your application. As you may notice, in all my examples, I only return the organization ID of the organization record that has changed. I never put a copy of the changes to the data in the message. Also, I used messages based on system events to tell other services that data state has changed, but I always force the other services to go back to the master (the service that owns the data) to retrieve a new copy of the data. This approach is costlier in terms of execution time, but it also guarantees I always have the latest copy of the data to work with. A chance still exists that the data you're working with could change right after you've read it from the source system, but that's much less likely than blindly consuming the information right off the queue. Think carefully about how much data you're passing around. Sooner or later, you'll run into a situation where the data you passed is stale. It could be stale

because a problem caused it to sit in a message queue too long, or a previous message containing data failed, and the data you’re passing in the message now represents data that’s in an inconsistent state (because your application relied on the message’s state rather than the actual state in the underlying data store). If you’re going to pass state in your message, make sure to include a date-time stamp or version number in your message so that the services that are consuming the data can inspect the data being passed and ensure that it’s not older than the copy of the data they already have.

### 10.3.3 Writing the message consumer in the licensing service

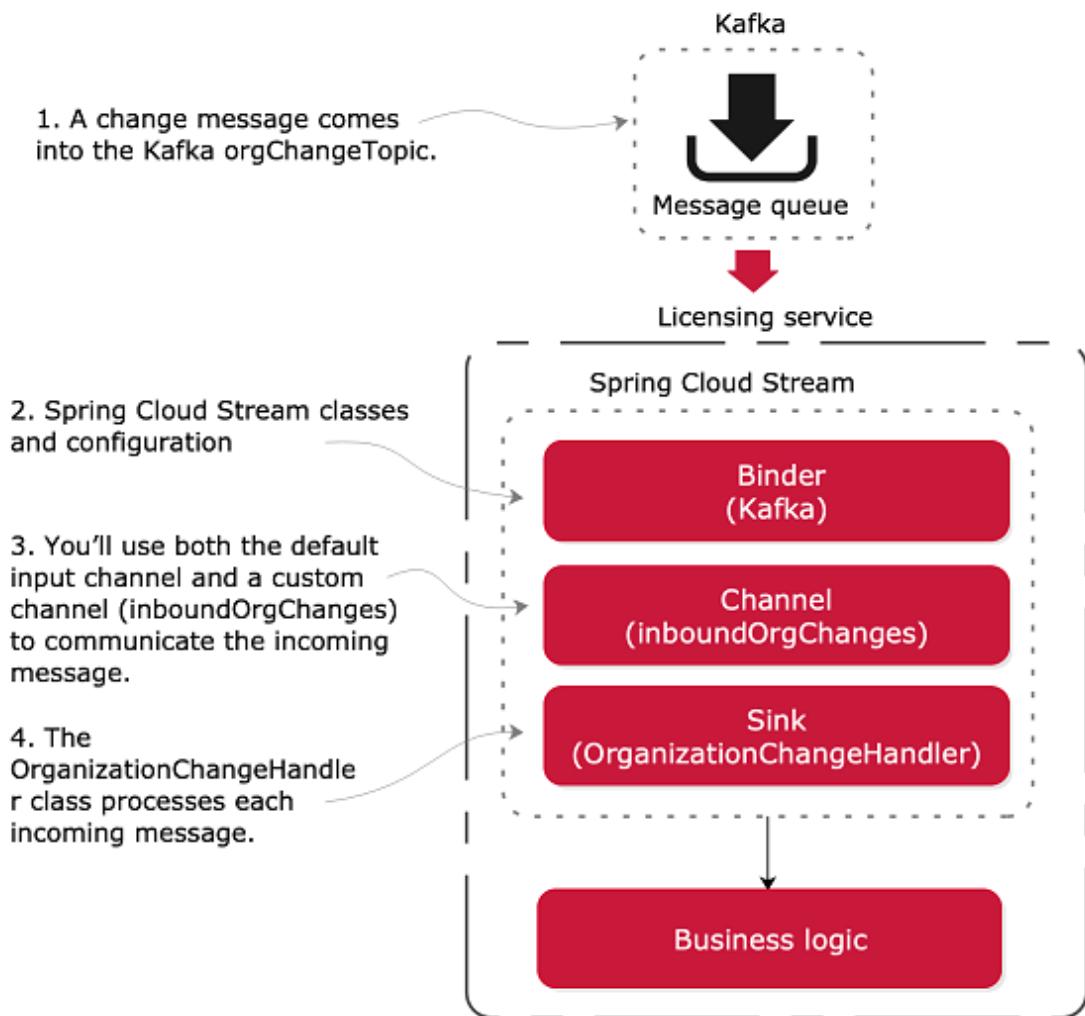
At this point, we’ve modified the organization service to publish a message to Kafka every time the organization service changes organization data. Anyone who’s interested can react without having to be explicitly called by the organization service. It also means we can easily add new functionality that can react to the changes in the organization service by having them listen to messages coming in on the message queue. Let’s now switch directions and look at how a service can consume a message using Spring Cloud Stream.

For this example, we’re going to have the licensing service consume the message published by the organization service.

To begin, we again need to add our Spring Cloud Stream dependencies to the licensing services pom.xml file. This pom.xml file can be found in the licensing-service root directory of the source code for the book. Similar to the organization-service pom.xml file you saw earlier, we add the following two dependency entries:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

Figure 10.5 shows where the licensing service will fit into the Spring Cloud architecture, first shown in figure 10.3.



## **Figure 10.5 When a message comes into the Kafka orgChangeTopic, the licensing service will respond.**

Then we need to tell the licensing service that it needs to use Spring Cloud Stream to bind to a message broker. Like the organization service, we're going to annotate the licensing services bootstrap class (/licensing-service/src/main/java/com/optimagrowth/license/LicenseServiceApplication.java) with the @EnableBinding annotation. The difference between the licensing service and the organization service is the value we're going to pass to the @EnableBinding annotation, as shown in the following code listing 10.8.

### **Listing 10.8 Consuming a message using Spring Cloud Stream**

```
package com.optimagrowth.license;
//Imports and rest of annotations removed for conciseness
@EnableBinding(Sink.class) #A
public class LicenseServiceApplication {
@StreamListener(Sink.INPUT) #B
public void loggerSink(OrganizationChangeModel orgChange) {
logger.debug("Received an {} event for organization id {}", orgChange.getAction(),
orgChange.getOrganizationId());
}
//Rest of the code removed for conciseness
}
```

#A The @EnableBinding annotation tells the service to use the channels defined in the Sink interface to listen for incoming messages.

#B Spring Cloud Stream will execute this method every time a message is received off the input channel.

Because the licensing service is a consumer of a message, we're going to pass the @EnableBinding annotation the value Sink.class. This tells Spring Cloud Stream to bind to a

message broker using the default Spring Sink interface. Similar to the Spring Cloud Stream Source interface described in section 10.3.1, Spring Cloud Stream exposes a default channel on the Sink interface. The channel on the Sink interface is called input and is used to listen for incoming messages on a channel.

Once we've defined that we want to listen for messages via the `@EnableBinding` annotation, we can write the code to process a message coming off the Sink input channel. To do this, we use the Spring Cloud Stream `@StreamListener` annotation.

The `@StreamListener` annotation tells Spring Cloud Stream to execute the `loggerSink()` method every time a message is received off the input channel. Spring Cloud Stream will automatically deserialize the message coming off the channel to a Java POJO called `OrganizationChangeModel`.

Once again, the actual mapping of the message broker's topic to the input channel is done in the licensing service's configuration. For the licensing service, its configuration is shown in the following code listing 10.9 and can be found at the Spring Cloud config repository, for this example at the `/configserver/src/main/resources/config/licensing-service.properties` file.

### **Listing 10.9 Mapping the licensing service to a message topic in Kafka**

```
#Rest of properties removed for conciseness
spring.cloud.stream.bindings.input.destination=orgChangeTopic #A
spring.cloud.stream.bindings.input.content-type=application/json
spring.cloud.stream.bindings.input.group= licensingGroup #B
spring.cloud.stream.kafka.binder.zkNodes= localhost
```

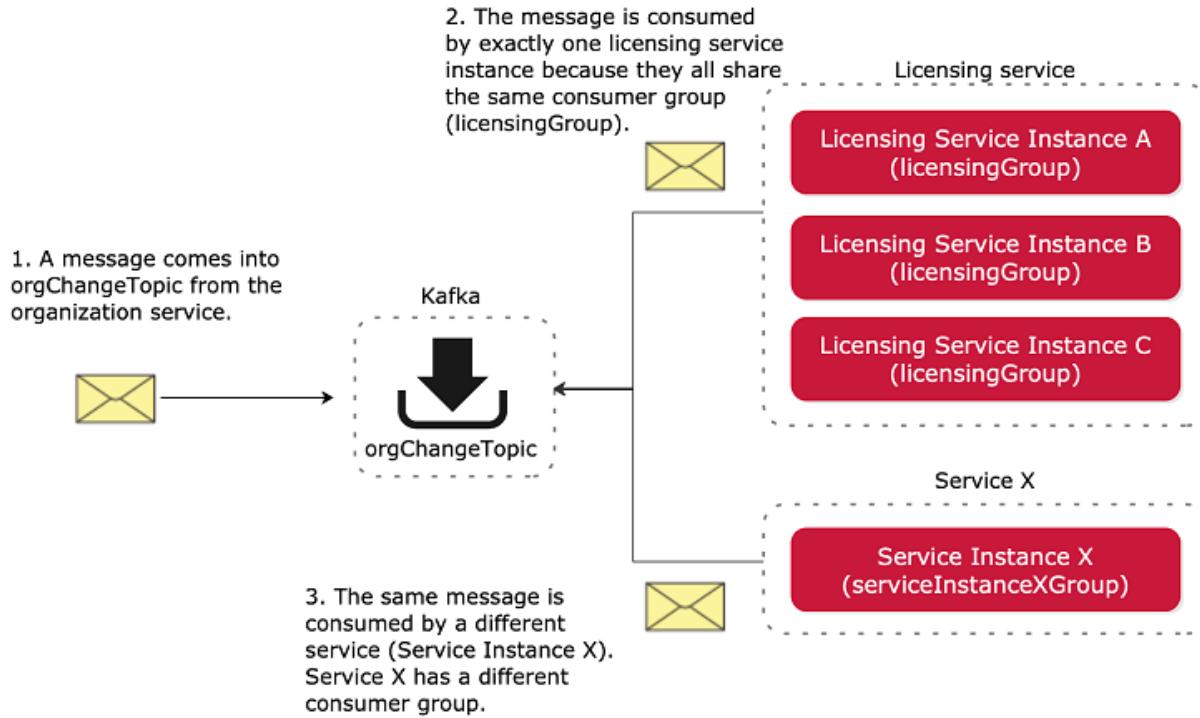
```
spring.cloud.stream.kafka.binder.brokers=localhost
```

- #A The `spring.cloud.stream.bindings.input` property maps the input channel to the `orgChangeTopic` queue.
- #B The `group` property is used to guarantee process-once semantics for a service.

The configuration in this listing looks like the configuration for the organization service. It has, however, two key differences. First, we now have a channel called `input` defined under the `spring.cloud.stream.bindings` property. This value maps to the `Sink.INPUT` channel defined in the code from listing 10.8. This property maps the input channel to the `orgChangeTopic`. Second, we see the introduction of a new property called `spring.cloud.stream.bindings.input.group`. The `group` property defines the name of the consumer group that will be consuming the message.

The concept of a consumer group is this: We might have multiple services with each service having multiple instances listening to the same message queue. We want each unique service to process a copy of a message, but we only want one service instance within a group of service instances to consume and process a message. The `group` property identifies the consumer group that the service belongs to. As long as all the service instances have the same group name, Spring Cloud Stream and the underlying message broker will guarantee that only one copy of the message will be consumed by a service instance belonging to that group. In the case of our licensing service, the `group` property value will be called `licensingGroup`.

Figure 10.6 illustrates how the consumer group is used to help enforce consume once semantics for a message being consumed across multiple services.



**Figure 10.6 The consumer group guarantees a message will only be processed once by a group of service instances.**

### 10.3.4 Seeing the message service in action

At this point, we have the organization service publishing a message to the `orgChangeTopic` every time a record is added, updated, or deleted and the licensing service receiving the message of the same topic. Now we'll see this

code in action by creating an organization service record and watching the console to see the corresponding log message appear from the licensing service.

To create the organization service record, we're going to issue a POST on the organization service. The endpoint we're going to use is

<http://localhost:8072/organization/v1/organization/>. The body we're going to send on the POST call to the endpoint is

```
{
 "name": "Ostock",
 "contactName": "Illary Huaylupo",
 "contactEmail": "illaryhs@gmail.com",
 "contactPhone": "888888888"
}
```

**NOTE** Remember; first, we need to make the authentication to retrieve the tokens and pass the access token via the authorization header as a Bearer token. In the previous chapter, we discuss this if you didn't follow the code samples you can download the code from (<http://github.com/ihuaylupo/manning-smia/tree/master/chapter9>). Second, we are pointing to the Spring Cloud Gateway, that's why the endpoint has the 8072 port and the /organization/v1/organization path instead of just /v1/organization.

Figure 10.7 shows the returned output from this POST call.

POST ▼ http://localhost:8072/organization/v1/organization/

Params Authorization (1) Headers (10) **Body (1)** Pre-request

none  form-data  x-www-form-urlencoded  raw  binary

```

1 {

2 "name": "Ostock",

3 "contactName": "Hillary Huaylupo",

4 "contactEmail": "hillaryhs@gmail.com",

5 "contactPhone": "888888888"

6 }

```

## Figure 10.7 Creating a new organization service record using the organization service.

Once the organization service call has been made, we should see the following output in the console window running the services. Figure 10.8 shows this output.

Log message from the organization service indicating it sent a Kafka message

Sending Kafka message SAVE for Organization Id: 25b9ad32-b4fd-4c4b-b65f-f27b5f1ab362  
 Adding the correlation id to the outbound headers. acd8ebf9-04dd-4135-8842-40913227b0d2  
 Completing outgoing request for http://localhost:8072/v1/organization/.  
 Received SAVE event for the organization id 25b9ad32-b4fd-4c4b-b65f-f27b5f1ab362

Log message from the licensing service indicating that it received a message for a SAVE event

## **Figure 10.8 The console shows the message from the organization service being sent and then received.**

Now we have two services communicating with each other using messages. Spring Cloud Stream is acting as the middleman for these services. From a messaging perspective, the services know nothing about each other. They're using a messaging broker to communicate as an intermediary and Spring Cloud Stream as an abstraction layer over the messaging broker.

## **10.4 A Spring Cloud Stream use case: distributed caching**

At this point, we have two services communicating with messaging, but we're not really doing anything with the messages. Now we'll build the distributed caching example we discussed earlier in the chapter. We'll have the licensing service always check a distributed Redis cache for the organization data associated with a particular license. If the organization data exists in the cache, we'll return the data from the cache. If it doesn't, we'll call the organization service and cache the results of the call in a Redis hash.

When data is updated in the organization service, the organization service will issue a message to Kafka. The licensing service will pick up the message and issue a delete against Redis to clear out the cache.

## Cloud caching and messaging

Using Redis as a distributed cache is very relevant to microservices development in the cloud. Nowadays, you can use Redis to

Improve performance for lookup of commonly held data—By using a cache, you can significantly improve the performance of several key services by avoiding reads to the database.

Reduce the load (and cost) on the database tables holding your data—Accessing data in the database can be a costly proposition. Every read you make is a chargeable event. Using a Redis server is significantly cheaper for reads by a primary key than a database read.

Increase resiliency so that your services can degrade gracefully if your primary data store (Database) is having performance problems— If your database is having problems, using a cache such as Redis can help your service degrade gracefully. Depending on how much data you keep in your cache, a caching solution can help reduce the number of errors you get from hitting your data store.

Redis is far more than a caching solution, but it can fill that role if you need a distributed cache.

### 10.4.1 Using Redis to cache lookups

Now we're going to begin by setting up the licensing service to use Redis. Fortunately, Spring Data already makes it simple to introduce Redis into our licensing service. To use Redis in the licensing service we need to do four things:

1. Configure the licensing service to include the Spring Data Redis dependencies
2. Construct a database connection to Redis
3. Define the Spring Data Redis Repositories that our code will use to interact with a Redis hash
4. Use Redis and the licensing service to store and read organization data

## ***CONFIGURE THE LICENSING SERVICE WITH SPRING DATA REDIS DEPENDENCIES***

The first thing we need to do is include the spring-data-redis dependencies, along with the jedis into the licensing service's pom.xml file. The dependencies to include are shown in the following code listing 10.10.

### ***Listing 10.10 Adding the Spring Redis Dependencies***

```
//Some code removed for conciseness
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-redis</artifactId>
</dependency>
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
<type>jar</type>
</dependency>
```

## ***CONSTRUCTING THE DATABASE CONNECTION TO A REDIS SERVER***

Now that we have the dependencies in Maven, we need to establish a connection out to our Redis server. Spring uses the open-source project Jedis (<https://github.com/xetorthio/jedis>) to communicate with a Redis server. To communicate with a specific Redis instance, we're going to expose a JedisConnectionFactory in the /licensing-service/src/main/java/com/optimagrowth/license/LicenseServiceApplication.java class as a Spring Bean. Once we have a

connection out to Redis, we're going to use that connection to create a Spring RedisTemplate object. The RedisTemplate object will be used by the Spring Data repository classes that we'll implement shortly to execute the queries and saves of organization service data to our Redis service. The following code listing 10.11 shows this code.

### **Listing 10.11 Establishing how our licensing service will communicate with Redis**

```
package com.optimagrowth.license;
import org.springframework.data.redis.connection.RedisPassword;
import org.springframework.data.redis.connection.RedisStandaloneConfiguration;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
//Most of the imports and annotations have been removed for conciseness
@SpringBootApplication
@EnableBinding(Sink.class)
public class LicenseServiceApplication {
@.Autowired
private ServiceConfig serviceConfig;
//All other methods in the class have been removed for consistency
@Bean #A
JedisConnectionFactory jedisConnectionFactory() {
String hostname = serviceConfig.getRedisServer();
int port = Integer.parseInt(serviceConfig.getRedisPort());
RedisStandaloneConfiguration redisStandaloneConfiguration = new
RedisStandaloneConfiguration(hostname, port);
return new JedisConnectionFactory(redisStandaloneConfiguration);
}
@Bean #B
public RedisTemplate<String, Object> redisTemplate() {
RedisTemplate<String, Object> template = new RedisTemplate<>();
template.setConnectionFactory(jedisConnectionFactory());
return template;
}
//Rest of the code removed for conciseness
}
```

#A The jedisConnectionFactory() method sets up the actual database connection to the Redis server.

#B The redisTemplate() method creates a RedisTemplate that will be used to carry out actions against your Redis server.

The foundational work for setting up the licensing service to communicate with Redis is complete. Let's now move over to writing the logic that will get, add, update, and delete data from Redis.

The ServiceConfig is a simple class that contains, the logic to retrieve custom parameters we define in the configuration file for the licensing service, in this particular scenario the Redis host and port. The following code 10.12 listing shows the code for the class.

### **Listing 10.12 Setting up the ServiceConfig class with Redis data**

```
package com.optimagrowth.authentication.config;
//Imports removed for conciseness
@Component @Getter
public class ServiceConfig{
//Rest of code removed for conciseness
@Value("${redis.server}")
private String redisServer="";
@Value("${redis.port}")
private String redisPort="";
}
```

The host and the port will be defined in the Spring Cloud Config service repository (/configserver/src/main/resources/config/licensing-service.properties).

```
redis.server = localhost
redis.port = 6379
```

**NOTE** It's important to highlight that I'm using Docker to run all the examples, if you want to run this locally you need to be sure to have Redis installed on your computer. If you are using Docker, you can find the

docker-compose.yml file up to date with the Redis container at <https://github.com/ihuaylupo/manning-smia/tree/master/chapter10/docker>.

## ***DEFINING THE SPRING DATA REDIS REPOSITORIES***

Redis is a key-value store data store that acts like a big, distributed, in-memory HashMap. In the simplest case, it stores data and looks up data by a key. It doesn't have any sophisticated query language to retrieve data. Its simplicity is its strength and one of the reasons why so many projects have adopted it for use in their projects.

Because we're using Spring Data to access our Redis store, we need to define a repository class. As may you remember from early on in the first chapters, Spring Data uses user-defined repository classes to provide a simple mechanism for a Java class to access our Postgres database without having to write low-level SQL queries.

For the licensing service, we're going to define two files for our Redis repository. The first file we'll write will be a Java interface that's going to be injected into any of the licensing service classes that are going to need to access Redis. This interface (/licensing-service/src/main/java/com/optimagrowth/license/repository/OrganizationRedisRepository.java) is shown in the following code listing 10.13.

### **Listing 10.13 OrganizationRedisRepository defines methods used to call Redis**

```
package com.optimagrowth.license.repository;
```

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.optimagrowth.license.model.Organization;
@Repository
public interface OrganizationRedisRepository extends
CrudRepository<Organization, String>{
}
```

The OrganizationRedisRepository, by extending from CrudRepository contains all the CRUD (Create, Read, Update, Delete) logic used for storing and retrieving data from in this case for Redis.

The second file is the model we are going to use for our repository, this class is a POJO containing the data we are going to store in our Redis cache. The /licensing-service/src/main/java/com/optimagrowth/license/model/Organization.java class is shown in the following code listing 10.14.

### **Listing 10.14 Organization Model Redis Hash**

```
package com.optimagrowth.license.model;
import org.springframework.data.redis.core.RedisHash;
import org.springframework.hateoas.RepresentationModel;
import javax.persistence.Id;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
@Getter @Setter @ToString
@RedisHash("organization") #A
public class Organization extends RepresentationModel<Organization> {
@Id
String id;
String name;
String contactName;
String contactEmail;
String contactPhone;
}
```

#A @RedisHash annotation sets the name of the hash in the Redis server where the organization data is stored.

One important thing to note from the code in listing 10.14 is that a Redis server can contain multiple hashes and data structures within it. So, we need to tell Redis the name of the data structure we're performing the operation against in every operation against the Redis server.

## ***USING REDIS AND THE LICENSING SERVICE TO STORE AND READ ORGANIZATION DATA***

Now that we have the code in place to perform operations against Redis, we can modify our licensing service so that every time the licensing service needs the organization data, it will check the Redis cache before calling out to the organization service. The logic for checking Redis will occur in the /licensing-service/src/main/java/com/optimagrowth/license/service/client/OrganizationRestTemplateClient.java class. The code for this class is shown in the following code listing 10.15.

### **Listing 10.15 OrganizationRestTemplateClient class will implement cache logic**

```
package com.optimagrowth.license.service.client;
//Imports removed for conciseness
@Component
public class OrganizationRestTemplateClient {
@Autowired
RestTemplate restTemplate;
@Autowired
OrganizationRedisRepository redisRepository; #A
private static final Logger logger =
LoggerFactory.getLogger(OrganizationRestTemplateClient.class);
private Organization checkRedisCache(String organizationId) {
try {
return redisRepository.findById(organizationId).orElse(null); #B
}catch (Exception ex){
logger.error("Error encountered while trying to retrieve organization {}")
```

```

check Redis Cache. Exception {}", organizationId, ex);
return null;
}
}
private void cacheOrganizationObject(Organization organization) {
try {
redisRepository.save(organization); #C
}catch (Exception ex){
logger.error("Unable to cache organization {} in Redis. Exception {}",
organization.getId(), ex);
}
}
public Organization getOrganization(String organizationId){
logger.debug("In Licensing Service.getOrganization: {}", organizationId);
UserContext.getCorrelationId();
Organization organization = checkRedisCache(organizationId);
if (organization != null){ #D
logger.debug("I have successfully retrieved an organization {} from the
redis cache: {}", organizationId, organization);
return organization;
}
logger.debug("Unable to locate organization from the redis cache: {}.", organizationId);
ResponseEntity<Organization> restExchange =
restTemplate.exchange(
"http://gateway:8072/organization/v1/organization/{organizationId}",
HttpMethod.GET,
null, Organization.class, organizationId);
organization = restExchange.getBody();
if (organization != null) {
cacheOrganizationObject(organization);
}
return restExchange.getBody();
}
}

```

#A The OrganizationRedisRepository class is auto-wired in the OrganizationRestTemplateClient.  
#B Trying to retrieve an Organization class with its organization ID from Redis  
#C Saving the organization in Redis  
#D If you can't retrieve data from Redis, you'll call out the organization service to retrieve the data from the source database to later save it in Redis.

The getOrganization() method is where the call to the organization service takes place. Before we make the actual REST call, we attempt to retrieve the Organization object associated with the call from Redis using the checkRedisCache() method. If the organization object in question is not in Redis, the code will return a null value. If a null value is returned from the checkRedisCache() method,

the code will invoke the organization service's REST endpoint to retrieve the desired organization record. If the organization service returns an organization, the returned organization object will be cached using the `cacheOrganizationObject()` method.

**NOTE** Pay close attention to exception handling when interacting with the cache. To increase resiliency, we never let the entire call fail if we cannot communicate with the Redis server. Instead, we log the exception and let the call go out to the organization service. In this particular use case, caching is meant to help improve performance, and the absence of the caching server shouldn't impact the success of the call.

With the Redis caching code in place, we should hit the licensing service and see the logging messages in the following code snippet. If we were to make two back-to-back GET requests on the following licensing service endpoint, <http://localhost:8072/license/v1/organization/e839ee96-28de-4f67-bb79-870ca89743a0/license/279709ff-e6d5-4a54-8b55-a5c37542025b>, we should see the following two output statements in our logs:

```
licensingservice_1 | DEBUG 1 --- [nio-8080-exec-4]
c.o.l.s.c.OrganizationRestTemplateClient : Unable to locate organization from the
redis cache: e839ee96-28de-4f67-bb79-870ca89743a0.
licensingservice_1 | DEBUG 1 --- [nio-8080-exec-7]
c.o.l.s.c.OrganizationRestTemplateClient : I have successfully retrieved an
organization e839ee96-28de-4f67-bb79-870ca89743a0 from the redis cache:
Organization(id=e839ee96-28de-4f67-bb79-870ca89743a0, name=Ostock,
contactName=Illary Huaylupo, contactEmail=illaryhs@gmail.com,
contactPhone=8888888888)
```

The first line from the console shows the first time we tried to hit the licensing service endpoint for organization `e839ee96-28de-4f67-bb79-870ca89743a0`. The licensing service first checked the Redis cache and couldn't find the organization record it was looking for. The code then calls the

organization service to retrieve the data. The second line that was printed from the console shows that when you hit the licensing service endpoint a second time, the organization record is now cached.

## 10.4.2 Defining custom channels

Previously we built our messaging integration between the licensing and organization services to use the default output and input channels that come packaged with the Source and Sink interfaces in the Spring Cloud Stream project. However, if we want to define more than one channel for our application or we want to customize the names of our channels, we can define our own interface and expose as many input and output channels as our application needs.

To create a custom channel, call inboundOrgChanges in the licensing service. You can define the channel in the /licensing-service/src/main/java/com/optimagrowth/license/events/CustomChannels.java interface, as shown in the following code 10.16 listing.

### **Listing 10.16 Defining a custom input channel for the licensing service**

```
package com.optimagrowth.license.service.client;
//Imports removed for conciseness
package com.optimagrowth.license.events;
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;
public interface CustomChannels {
 @Input("inboundOrgChanges") #A
 SubscribableChannel orgs(); #B
}
```

- #A The @Input annotation is a method-level annotation that defines the name of the channel.
- #B Each channel exposed through the @Input annotation must return a SubscribableChannel class.

The key takeaway from listing 10.16 is that for each custom input channel we want to expose, we mark a method that returns a SubscribableChannel class with the @Input annotation. We will use the @OutputChannel above the method that will be called if we want to define output channels for publishing messages. In the case of an output channel, the defined method will return a MessageChannel class instead of the SubscribableChannel class used with the input channel:

```
@OutputChannel("outboundOrg")
MessageChannel outboundOrg();
```

Now that we have a custom input channel defined, we need to modify two more things to use it in the licensing service. First, we need to modify the licensing service to map our Kafka topic's custom input channel name in the licensing configuration file. The following code listing 10.17 shows this.

### **Listing 10.17 Modifying the licensing service to use our custom input channel**

```
//Rest removed for conciseness
spring.cloud.stream.bindings.inboundOrgChanges.destination= orgChangeTopic
spring.cloud.stream.bindings.inboundOrgChanges.content-type= application/json
spring.cloud.stream.bindings.inboundOrgChanges.group= licensingGroup
spring.cloud.stream.kafka.binder.zkNodes= localhost
spring.cloud.stream.kafka.binder.brokers=localhost
```

To use our custom input channel, we need to inject the CustomChannels interface we defined into a class that's going to use it to process messages. For the distributed caching example, I've moved the code for handling an incoming message to the following licensing-service class: /licensing-service/src/main/java/com/optimagrowth/license/events/handler/OrganizationChangeHandler.java. The following code listing 10.18 shows the message handling code that we'll use with the inboundOrgChanges channel we defined.

### **Listing 10.18 Processing an organization change with the new custom channel**

```
package com.optimagrowth.license.events.handler;
//Imports removed for conciseness
@EnableBinding(CustomChannels.class) #A
public class OrganizationChangeHandler {
 private static final Logger logger =
 LoggerFactory.getLogger(OrganizationChangeHandler.class);
 private OrganizationRedisRepository organizationRedisRepository; #B
 @StreamListener("inboundOrgChanges") #C
 public void loggerSink(OrganizationChangeModel organization) { #D
 logger.debug("Received a message of type " + organization.getType());
 logger.debug("Received a message with an event {} from the organization service
 for the organization id {} ",
 organization.getType(), organization.getType());
 }
}
```

#A Move the @EnableBindings out of the Application.java class and into the OrganizationChangeHandler class. This time instead of using Sink.class, use your CustomChannels class as the parameter to pass.

#B The OrganizationRedisRepository class that we use to interact with Redis is injected into the OrganizationChangeHandler. We can use it to do any CRUD operations.

#C With the @StreamListener annotation, we passed in the name of our channel, "inboundOrgChanges", instead of using Sink.INPUT.

#D When we receive a message, inspect the action that was taken with the data and then react accordingly.

Now, let's create an organization and then let's find that organization, we can do this using the following endpoints:  
<http://localhost:8072/organization/v1/organization/> and  
<http://localhost:8072/organization/v1/organization/d989f37d-9a59-4b59-b276-2c79005ea0d9>.

Figure 10.9 shows the console output of these calls with the OrganizationChangeHandler.



## Figure 10.9 The console shows the message from the organization service being sent and then received.

Now that we know how to use Spring Cloud Stream and Redis let's continue with the next chapter in which we will see several techniques and technologies to create a distributed tracing using Spring Cloud.

## 10.5 Summary

- Asynchronous communication with messaging is a critical part of microservices architecture.

- Using messaging within our applications allows our services to scale and become more fault-tolerant.
- Spring Cloud Stream simplifies the production and consumption of messages by using simple annotations and abstracting away platform-specific details of the underlying message platform.
- A Spring Cloud Stream message source is an annotated Java method that's used to publish messages to a message broker's queue.
- A Spring Cloud Stream message sink is an annotated Java method that receives messages off a message broker's queue.
- Redis is a key-value store that can be used as both a database and a cache.

# 11 Distributed tracing with Spring Cloud Sleuth and Zipkin

This chapter covers

- Using Spring Cloud Sleuth to inject tracing information into service calls
- Using log aggregation to see logs for distributed transaction
- Transforming, Searching, analyzing and visualizing log data in real-time with ELK stack
- Using OpenZipkin to visually understand a user's transaction as it flows across multiple service class
- Customizing tracing information with Spring Cloud Sleuth and Zipkin

The microservices architecture is a powerful design paradigm for breaking down complex monolithic software systems into smaller, more manageable pieces. These manageable pieces can be built and deployed independently of each other; however, this flexibility comes at a price: complexity. Because microservices are distributed by nature, trying to debug where a problem is occurring can be maddening. The distributed nature of the services means that we have to trace one or more transactions across

multiple services, physical machines, and different data stores, and try to piece together what exactly is going on.

This chapter lays out several techniques and technologies for making distributed debugging possible. In this chapter, we look at the following:

- Using correlation IDs to link together transactions across multiple services
- Aggregating log data from various services into a single searchable source
- Visualize the flow of a user transaction across multiple services and understand each part of the transaction's performance characteristics.
- Analyzing, searching, and visualizing log data in real-time using the Elasticsearch Logstash and Kibana (ELK) Stack

To accomplish the previous points, we are going to use the following technologies:

- **Spring Cloud Sleuth** (<https://cloud.spring.io/spring-cloud-sleuth/reference/html/>). Spring Cloud Sleuth is a Spring Cloud project that instruments our incoming HTTP requests with trace IDs (correlation ID). It does this by adding the filters and interacting with other Spring components to let the correlation IDs being generated pass through to all the system calls.
- **Zipkin** (<https://zipkin.io/>). Zipkin is an open-source data-visualization tool that can show the flow of a transaction across multiple services. Zipkin allows us to break a transaction down into its component pieces and visually identify where there might be performance hotspots.
- **ELK Stack** (<https://www.elastic.co/what-is/elk-stack>). ELK stack combines three open-source tools (Elasticsearch, Logstash, and Kibana) that allows us to analyze, search, and visualize logs in real-time. Elasticsearch is a distributed analytics engine for all types of data,

structured, non-structured, numeric, text, and more. Logstash is a server-side data processing pipeline that allows us to ingest data from multiple sources simultaneously and transform it before it is indexed in Elasticsearch. Logstash, is used to add and process data and send it to Elasticsearch. Kibana is the visualization and data management tool for Elasticsearch. It provides charts, maps, and real-time histograms.

To begin this chapter, we start with the simplest of tracing tools, the correlation ID.

**NOTE** Parts of this chapter rely on material covered in chapter 8 (mainly the material on the Spring Gateway pre-, response, and post-filters). If you haven't read chapter 8 yet, I recommend that you do so before you read this chapter.

## 11.1 Spring Cloud Sleuth and the correlation ID

We first introduced the concept of correlation IDs in chapters 7 and 8. A correlation ID is a randomly generated, unique number, or string assigned to a transaction when a transaction is initiated. As the transaction flows across multiple services, the correlation ID is propagated from one service call to another. In the context of chapter 8, we used a Spring Cloud Gateway filter to inspect all incoming HTTP requests and inject a correlation ID if one wasn't present.

Once the correlation ID was present, we used a custom Spring HTTP filter on every one of our services to map the incoming variable to a custom UserContext object. With the UserContext object in place, we could now manually add the

correlation ID to any of our log statements by making sure we appended the correlation ID to the log statement, or, with a little work, add the correlation ID directly to Spring's Mapped Diagnostic Context (MDC). Remember, MDC is a map that stores a set of key-value pairs provided by the application that are going to be inserted in the log messages. We also wrote a Spring Interceptor that would ensure that all HTTP calls from a service would propagate the correlation ID by adding the correlation ID to the HTTP headers on any outbound calls.

Fortunately, Spring Cloud Sleuth manages all this code infrastructure and complexity for us. By adding Spring Cloud Sleuth to our Spring Microservices, we can

- Transparently create and inject a correlation ID into our service calls if one doesn't exist.
- Manage the propagation of the correlation ID to outbound service calls so that the correlation ID for a transaction is automatically added to outbound calls.
- Add the correlation information to Spring's MDC logging so that the generated correlation ID is automatically logged by Spring Boot's default SL4J and Logback implementation.
- Optionally, publish the tracing information in the service call to the Zipkin distributed tracing platform.

**NOTE** With Spring Cloud Sleuth, if we use Spring Boot's logging implementation, we'll automatically get correlation IDs added to the log statements we put in our microservices.

Let's go ahead and add Spring Cloud Sleuth to our licensing and organization services.

## 11.1.1 Adding Spring Cloud sleuth to licensing and organization

To start using Spring Cloud Sleuth in our two services (licensing and organization), we need to add a single Maven dependency to the pom.xml files in both services:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

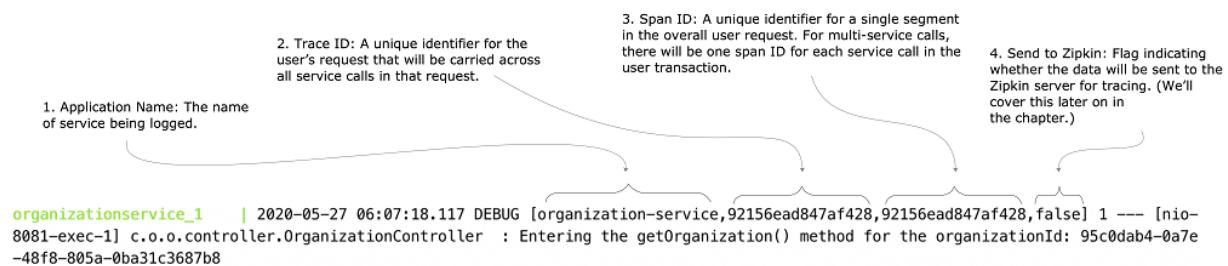
This dependency will pull in all the core libraries needed for Spring Cloud Sleuth. That's it. Once this dependency is pulled in, our service will now

1. Inspect every incoming HTTP service and determine whether or not Spring Cloud Sleuth tracing information exists in the incoming call. If the Spring Cloud Sleuth tracing data does exist, the tracing information passed into our microservice will be captured and made available to our service for logging and processing.
2. Add Spring Cloud Sleuth tracing information to the Spring MDC so that every log statement created by our microservice will be added to the logs.
3. Inject Spring Cloud tracing information into every outbound HTTP call and Spring messaging channel message our service makes.

## 11.1.2 Anatomy of a Spring Cloud Sleuth trace

If everything is set up correctly, any log statements written within our service application code will now include Spring Cloud Sleuth trace information. For example, figure 11.1 shows what the service's output would look like if we were to do an HTTP GET

<http://localhost:8072/organization/v1/organization/95c0dab4-0a7e-48f8-805a-0ba31c3687b8> on the organization service.



**Figure 11.1 Spring Cloud Sleuth adds four pieces of tracing information to each log entry written by your service. This data helps tie together service calls for a user's request.**

Spring Cloud Sleuth will add four pieces of information to each log entry. These four pieces (numbered to correspond with the numbers in figure 11.1) are

- **Application name of the service.** This is going to be the name of the application the log entry is being made in. By default, Spring Cloud Sleuth uses the name of the application (`spring.application.name`) as the name that gets written in the trace.

- **Trace ID.** Trace ID is the equivalent term for correlation ID. It's a unique number that represents an entire transaction.
- **Span ID.** A span ID is a unique ID that represents part of the overall transaction. Each service participating within the transaction will have its own span ID. Span IDs are particularly relevant when you integrate with Zipkin to visualize your transactions.
- **Export.** Whether trace data was sent to Zipkin. In high-volume services, the amount of trace data generated can be overwhelming and not add a significant amount of value. Spring Cloud Sleuth lets us determine when and how to send a transaction to Zipkin. The true/false indicator at the end of the Spring Cloud Sleuth tracing block tells us whether the tracing information was sent to Zipkin.

**NOTE** It's important to highlight that by default, any application flow starts with the same Trace Id and Span Id.

Up to now, we've only looked at the logging data produced by a single service call. Let's look at what happens when you make a call to the licensing service. Figure 11.2 shows the logging output from the two service calls.

The diagram shows a vertical stack of log entries from four different services: organizationservice\_1, gatewayserver\_1, organizationservice\_1, and licensingservice\_1. Annotations with arrows point to specific parts of the logs:

- An arrow points to the first log entry from organizationservice\_1, which contains a trace ID: "The two calls have the same trace ID".
- An arrow points to the second log entry from organizationservice\_1, which contains a span ID: "The span IDs for the two service calls are different".

```

organizationservice_1 | 2020-05-27 06:07:47.261 DEBUG [organization-service,85f4c6e4a1738e77,85f4c6e4a1738e77,false] 1 --- [nio-8081-exec-3] c.o.o.controller.OrganizationController : Entering the getOrganization() method for the organizationId: 95c0dab4-0a7e-48f8-805a-0ba31c3687b8
organizationservice_1 | Hibernate: select organizati0_.organization_id as organizati0_0_, organizati0_.contact_email as contact_2_0_0_, organizati0_.contact_name as contact_3_0_0_, organizati0_.contact_phone as contact_4_0_0_, organizati0_.name as name5_0_0_ from organizations organizati0_ where organizati0_.organization_id=?
organizationservice_1 | 2020-05-27 06:07:47.265 DEBUG [organization-service,85f4c6e4a1738e77,85f4c6e4a1738e77,false] 1 --- [nio-8081-exec-3] c.o.o.events.source.SimpleSourceBean : Sending Kafka message GET for Organization Id: 95c0dab4-0a7e-48f8-805a-0ba31c3687b8
gatewayserver_1 | 2020-05-27 06:07:47.271 DEBUG 1 --- [or-http-epoll-4] c.o.gateway.filters.ResponseFilter : Adding the correlation id to the outbound headers. 18d930d2-bd37-4a1c-8cbf-e5115707696c
gatewayserver_1 | 2020-05-27 06:07:47.272 DEBUG 1 --- [or-http-epoll-4] c.o.gateway.filters.ResponseFilter : Completing outgoing request for http://localhost:8072/v1/organization/95c0dab4-0a7e-48f8-805a-0ba31c3687b8.
licensingservice_1 | 2020-05-27 06:07:47.273 DEBUG [licensing-service,85f4c6e4a1738e77,382ce00e427adf7b,false] 1 --- [container-0-C-1] c.o.l.e.h.OrganizationChangeHandler : Received a message of type com.optimagrowth.organization.events.model.OrganizationChangeModel

```

## **Figure 11.2 With multiple services involved in a transaction, we can see that they share the same trace ID.**

By looking at figure 11.2, we can see that both the licensing and organization services have the same trace ID 85f4c6e4a1738e77. However, the organization service has a span ID of 85f4c6e4a1738e77 (the same value as the transaction ID). The licensing service has a span ID of 382ce00e427adf7b.

By adding nothing more than a few POM dependencies, we've replaced all the correlation ID infrastructure that you built out in chapters 7 and 8.

## **11.2 Log aggregation and Spring Cloud Sleuth**

In a large-scale microservice environment (especially in the cloud), logging data is a critical tool for debugging problems. However, because the functionality of a microservice-based application is decomposed into small, granular services and we can have multiple service instances for a single service type, trying to tie it to log data from multiple services to resolve a user's problem can be extremely difficult.

Developers trying to debug a problem across multiple servers often have to try the following:

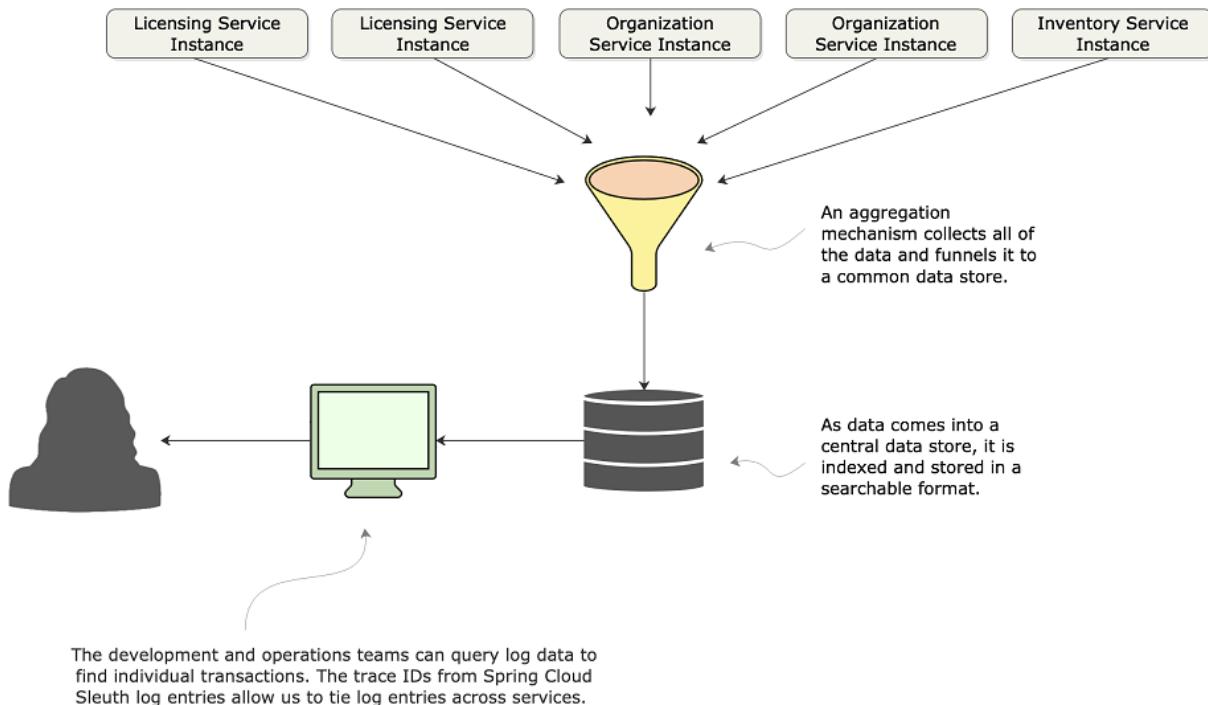
- Log into multiple servers to inspect the logs present on each server. This is an extremely laborious task, especially if the services in question have different

transaction volumes that cause logs to roll over at different rates.

- Write home-grown query scripts that will attempt to parse the logs and identify the relevant log entries. Because every query might be different, we often end up with a large proliferation of custom scripts for querying data from our logs.
- Prolong the recovery of a downgrade service process because the developer needs to back up the logs residing on a server. If a server hosting a service crashes completely, the logs are usually lost.

Each of the problems listed are real problems that we've run into. Debugging a problem across distributed servers is ugly work and often significantly increases the amount of time it takes to identify and resolve an issue.

A much better approach is to stream, real-time, all the logs from all of our service instances to a centralized aggregation point where the log data can be indexed and made searchable. Figure 11.3 shows at a conceptual level how this "unified" logging architecture would work.



**Figure 11.3 The combination of aggregated logs and a unique transaction ID across service log entries makes debugging distributed transactions more manageable.**

Fortunately, there are multiple open source and commercial products that can help us implement the previously described logging architecture. Also, multiple implementation models exist that will allow us to choose between an on-premise, locally managed solution, or a cloud-based solution. Table 11.1 summarizes several of the choices available for logging infrastructure.

**Table 11.1 Log Aggregation Solutions to use with Spring Boot**

Product	Implementation Notes
---------	----------------------

Name	Models	
Elasticsearch, Logstash, Kibana (ELK)	Open source Commercial Typically implemented on-premise	<a href="https://www.elastic.co/what-is/elk-stack">https://www.elastic.co/what-is/elk-stack</a> General-purpose search engine Can do log-aggregation through the ELK-stack
Graylog	Open source Commercial On-premise	<a href="https://www.graylog.org/">https://www.graylog.org/</a> An open-source platform that's designed to be installed on-premise
Splunk	Commercial only On-premise and cloud-based	<a href="https://www.splunk.com/">https://www.splunk.com/</a> Oldest and most comprehensive of the log management and aggregation tools Initially an on-premise solution, but have since offered a cloud offering
Sumo Logic	Freemium Commercial Cloud-based	<a href="https://www.sumologic.com/">https://www.sumologic.com/</a> Freemium/tiered pricing model Runs only as a cloud service Requires a corporate work account to signup (no Gmail or Yahoo accounts)

Papertrail	Freemium Commercial Cloud-based	<a href="https://www.papertrail.com/">https://www.papertrail.com/</a> <a href="#">Freemium/tiered pricing</a> <a href="#">model Runs only as a cloud service</a>
------------	---------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------

With all these choices, it might be challenging to choose which one is the best. Every organization is going to be different and have different needs.

For the purposes of this chapter, we're going to look at ELK as an example of how to integrate Spring Cloud Sleuth-backed logs into a unified logging platform. I've chosen ELK because

- ELK is open source.
- ELK It's straightforward to set up, simple to use, and user friendly.
- ELK is a complete tool that allows us to search, analyze, and visualize real-time logs generated from different services.
- ELK allows us to centralize all the logging to identify server and application issues.

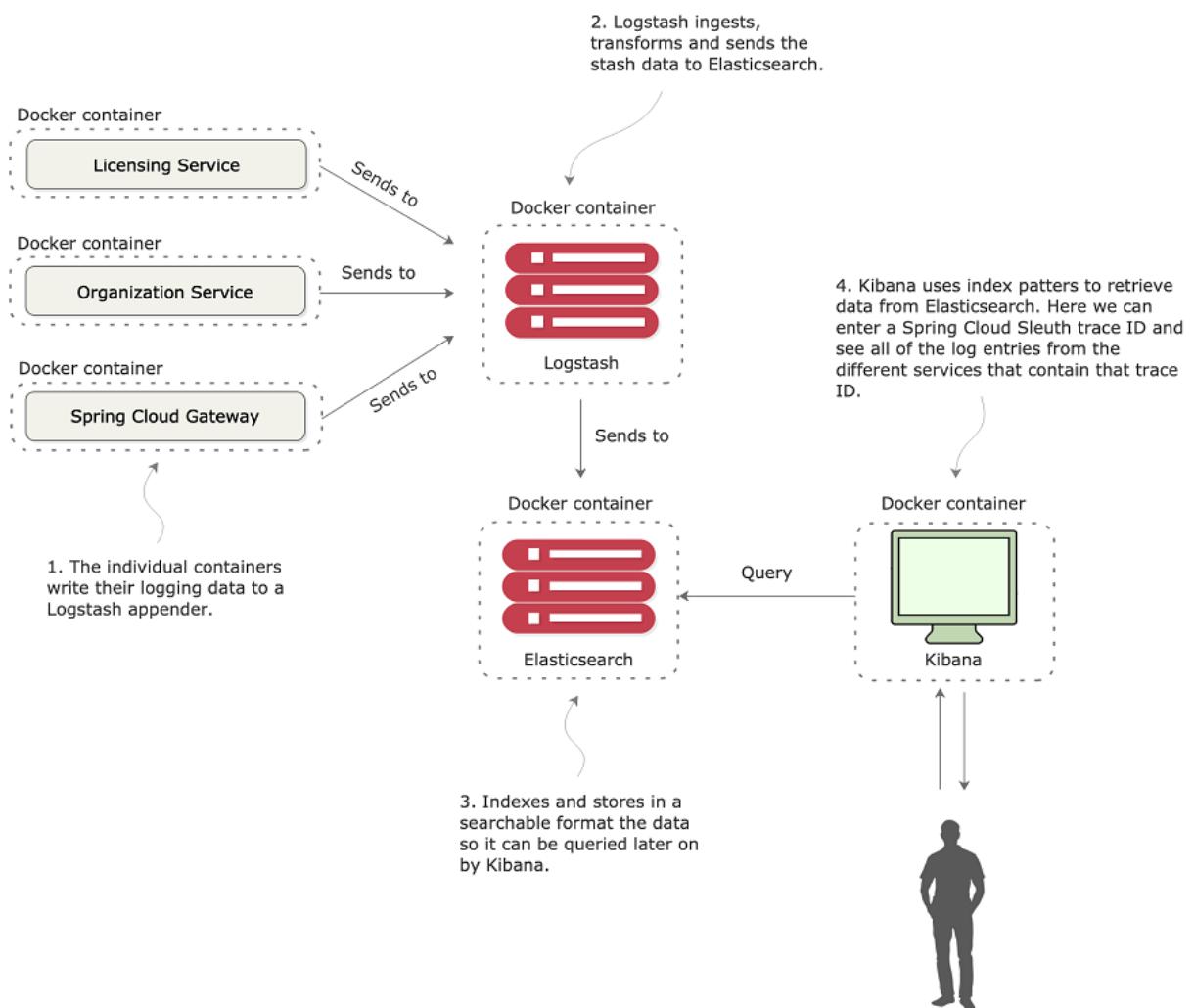
## 11.2.1 A Spring Cloud Sleuth/ELK stack implementation in action

In figure 11.3, we saw a general unified logging architecture. Let's now see how the same architecture can be implemented with Spring Cloud Sleuth and ELK Stack.

To set up ELK to work with our environment, we have to take the following actions:

1. Configure Logback in our services.
2. Define and run the ELK stack applications in docker containers.
3. Configure Kibana.
4. Test the implementation by issuing queries based on the correlation ID from Spring Cloud Sleuth.

Figure 11.4 shows the end state for our implementation and how Spring Cloud Sleuth and ELK fit together for our solution.



## **Figure 11.4 ELK allows us to quickly implement a unified logging architecture.**

Figure 11.4 we can see the logging architecture using the ELK Stack. The licensing, organization, and gateway service communicate via TCP with Logstash and send the log data. Logstash, filters, transforms, and passes the data to a central data store in this specific case Elasticsearch. Elasticsearch indexes and stores the data in a searchable format so it can be queried later on by Kibana.

Once the data is stored, Kibana can use index patterns to retrieve data from Elasticsearch. At this point, we can create a specific query index and enter a Spring Cloud Sleuth Trace ID to see all of the log entries from the different services that contain that trace ID (Correlation ID).

Once the data is stored, the development or operation team can look for the services' real-time logs by just accessing Kibana.

### **11.2.2 Configure logback in our services.**

Now, that we've seen the logging architecture with ELK, let's start configuring logback in our service. In order to do this, we need to do the following two steps.

1. Add the logstash-logback-encoder dependency in the pom.xml of our services.
2. Create the Logstash TCP Appender in a Logback configuration file.

## ***ADD THE LOGSTASH ENCODER***

To begin, we need to add the logstash-logback-encoder dependency in the pom.xml file of our licensing, organization, and gateway services. Remember, the pom.xml file can be found in the root directory of the source code.

```
<dependency>
<groupId>net.logstash.logback</groupId>
<artifactId>logstash-logback-encoder</artifactId>
<version>6.3</version>
</dependency>
```

## ***CREATE THE LOGSTASH TCP APPENDER***

Once added the dependency, we need to tell the licensing service that it needs to communicate with Logstash to send the application logs, and format those application logs into a JSON format. Logback, by default, produces the application logs in plain text, but to use the Elasticsearch indexes, we need to make sure we are sending the log data in a JSON format. There are three ways to accomplish this.

1. Use the net.logstash.logback.encoder.LogstashEncoder
2. Use the  
net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder
3. Parse the plain text log data in Logstash.

For purposes of this example, I will use the LogstashEncoder. I've chosen this option because it is the easiest and fastest

to use and also because, in this example, we don't need to add additional fields to the logger. With LoggingEventCompositeJsonEncoder, we can add new patterns, fields, disable default providers, and more. If we choose one of these two options, the one in charge of parsing the log files into a Logstash format is the Logback; on the third and last option, we delegate the parsing entirely to the Logstash using the JSON filter.

All three options are good, but I suggest using the LoggingEventCompositeJsonEncoder when you have to add or remove default configurations. The other two options will depend entirely on your business needs. You can choose whether to handle the log info in the application or Logstash.

To configure this encoder, we are going to create a Logback configuration file called logback-spring.xml. This configuration file should be located in the service resources folder. For the licensing service, the Logback configuration is shown in the following code listing 11.1 and can be found in the licensing service's /licensing-service/src/main/resources/logback-spring.xml file.

## **Listing 11.1 Configuring Logback with Logstash in the licensing service.**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<include resource="org/springframework/boot/logging/logback/base.xml"/>
<springProperty scope="context" name="application_name"
source="spring.application.name"/>
<appender name="logstash"
class="net.logstash.logback.appenders.LogstashTcpSocketAppender"> #A
<destination>logstash:5000</destination> #B
<encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
</appender>
<root level="INFO">
<appender-ref ref="logstash"/>
```

```

<appender-ref ref="CONSOLE"/>
</root>
<logger name="org.springframework" level="INFO"/>
<logger name="com.optimagrowth" level="DEBUG"/>
</configuration>

```

#A This indicates that we are going to use the TcpSocketAppender to communicate with Logstash.

#B Logstash hostname and port data to establish the TCP communication.

The previous configuration will produce the following log output shown in figure 11.5.

```
{
 "_index": "logstash-2020.05.30-000001",
 "_type": "_doc",
 "_id": "NwhPY3IBRu5zD4iyn8z0",
 "_version": 1,
 "_score": null,
 "_source": {
 "X-Span-Export": "false",
 "level_value": 10000,
 "@version": "1",
 "thread_name": "http-nio-8080-exec-9",
 "host": "licensing-service.docker_backend",
 "spanId": "6e761268be8708ed",
 "X-B3-TraceId": "6e761268be8708ed",
 "logger_name": "com.optimagrowth.license.service.client.OrganizationRestTemplateClient",
 "X-B3-SpanId": "6e761268be8708ed",
 "port": 51522,
 "spanExportable": "false",
 "message": "I have successfully retrieved an organization f31ced82-53e6-48d3-8969-0095ec7cdaf5 from the redis cache",
 "@timestamp": "2020-05-30T02:01:02.043Z",
 "traceId": "6e761268be8708ed",
 "application_name": "licensing-service",
 "level": "DEBUG"
 },
 "fields": {
 "@timestamp": [
 "2020-05-30T02:01:02.043Z"
]
 },
 "sort": [
 1590804062043
]
}
```

**Figure 11.5 Application log formatted with the LogstashEncoder.**

In Figure 11.5 we can see two important aspects. The first one is that the LogstashEncoder includes all the values stored in the logger Mapped Diagnostic Context (MDC) by default. The second one is that because we already have the Spring Cloud Sleuth dependency in our service, we can see the TraceId, X-B3-TraceId, SpanId, X-B3-SpanId, spanExportable fields in our log data. It is essential to highlight that all the X-b3 format name is the default Header propagation that Spring Cloud Sleuth uses from service to service. This name consists of the prefix x, that is used for custom headers that are not part of the HTTP specification and the b3 stands for "BigBrotherBird" the previous name of Zipkin.

**NOTE** If you want to know more about the MDC fields, I highly recommend you read the following SL4J logger documentation <http://www.slf4j.org/manual.html#mdc> and the Spring Cloud Sleuth documentation <https://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/2.1.0.RELEASE/single/spring-cloud-sleuth.html>.

The log data shown in the figure 11.5 is also configurable by using the LoggingEventCompositeJsonEncoder. By using the composite encoder, we can disable all the providers that were added by default into our configuration, add new patterns to display custom or existent mdc fields, and more. The following code listing 11.2 shows a brief example of the logback-spring.xml configuration that deletes some of the output fields and creates a new pattern with a custom field and other existent fields.

## **Listing 11.2 Configuring Logback with Logstash in the licensing service.**

```
<encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
```

```
<providers>
<mdc>
<excludeMdcKeyName>X-B3-TraceId</excludeMdcKeyName>
<excludeMdcKeyName>X-B3-SpanId</excludeMdcKeyName>
<excludeMdcKeyName>X-B3-ParentSpanId</excludeMdcKeyName>
</mdc>
<context/>
<version/>
<logLevel/>
<loggerName/>
<pattern>
<pattern>
<omitEmptyFields>true</omitEmptyFields>
{
"application": {
version: "1.0"
},
"trace": {
"trace_id": "%mdc{traceId}",
"span_id": "%mdc{spanId}",
"parent_span_id": "%mdc{X-B3-ParentSpanId}",
"exportable": "%mdc{spanExportable}"
}
}
</pattern>
</pattern>
<threadName/>
<message/>
<logstashMarkers/>
<arguments/>
<stackTrace/>
</providers>
</encoder>
```

For this example, I've chosen the LogstashEncoder option like I previously explained, but feel free to determine the option that best fits your needs.

Now that we have our Logback configuration in the licensing service, let's add the same configuration to our configuration and gateway services and let's continue with the other step to define and run the ELK stack applications in docker containers.

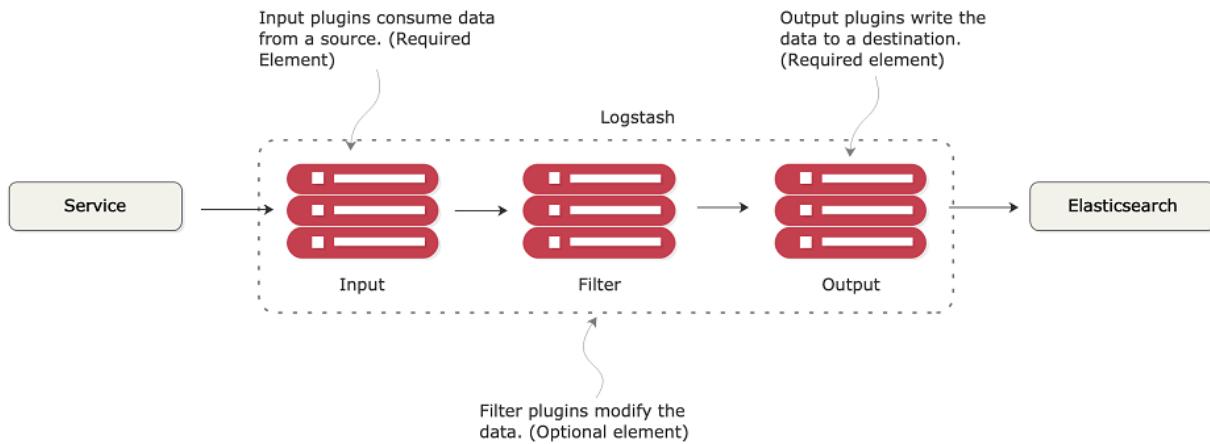
## 11.2.3 Define and run ELK stack applications in Docker

To set up our ELK stack containers, we also need to follow two simple steps. The first one is to create the Logstash configuration file, and the other one is to define the ELK stack applications in our Docker configuration.

Before we start creating our configuration, it's important to highlight that the Logstash pipeline has two required and one optional element. The required elements are the inputs and outputs. The input enables a specific source of events to be read by Logstash; Logstash supports a variety of input plugins such as GitHub, Http, TCP, Kafka, and more. On the other hand, the output is in charge of sending the event data to a particular destination. Elastic also supports a variety of plugins for this, such as CSV, Elasticsearch, email, file, MongoDB, Redis, stdout, and more.

The optional element in the Logstash configuration is the filter plugins. These filters are in charge of performing intermediary processing on an event such as translations, adding new information, parsing dates, truncate fields, and more. Remember, Logstash ingests and transforms the log data received.

The following figure 11.6, describes the Logstash process.



**Figure 11.6 Logstash configuration process contains two required and one optional element.**

For purposes of this example, we are going to use as input plugin the logback TCP Appender previously configured and as output the Elasticsearch engine. The following code listing 11.3 shows the /docker/config/logstash.conf file.

### Listing 11.3 Logstash configuration file

```

input { #A
 tcp {
 port => 5000 #B
 codec => json_lines
 }
}
filter {
 mutate {
 add_tag => ["manningPublications"] #C
 }
}
output { #D
 elasticsearch {
 hosts => "elasticsearch:9200" #E
 }
}

```

#A TCP Input plugin reads events from a TCP socket.

#B Logstash port.

```
#C Mutate filter, adds a specific tag to the events.
#D Elasticsearch output plugin, sends the log data to the Elasticsearch engine.
#E Elasticsearch port.
```

In the above code listing 11.3, we can see five essential elements. The first one is the input section. In this section, we are going to specify the tcp plugin for consuming the log data. The second one is the port 5000; this is the port that we are going to specify for Logstash later on in the docker-compose.yml file. If we go back to figure 11.4, you'll remember that we are going to be sending the applications logs directly to the Logstash.

The third element is optional and corresponds to the filters; for this particular scenario, I added a mutate filter. This filter adds the manningPublications tag to the events. A real scenario example of a possible tag perhaps could be the environment where the application is running.

The four and fifth elements specify the output plugin for our Logstash service. The previous code sends the processed data to Elasticsearch, service that will be running on the port 9200.

**NOTE** If you are interested in knowing more about all the input, output, and filter plugins elastic offers, I highly recommend you visit the following pages: <https://www.elastic.co/guide/en/logstash/current/input-plugins.html>, <https://www.elastic.co/guide/en/logstash/current/output-plugins.html>, and <https://www.elastic.co/guide/en/logstash/current/filter-plugins.html>.

Now, that we have the Logstash configuration, let's add the three ELK docker entries to the docker-compose.yml file. Remember, we are using this file to fire up all of the Docker containers used for the code examples in this chapter and

the previous chapters. The following code listing 11.4 shows the docker/docker-compose.yml file with the new entries:

## **Listing 11.4 ELK Stack Docker-compose configuration.**

```
#rest of the docker-compose.yml remove for conciseness
#Some additonal configuration has been removed for conciseness.
elasticsearch:
 image: docker.elastic.co/elasticsearch/elasticsearch:7.7.0 #A
 container_name: elasticsearch
 volumes:
 - esdata1:/usr/share/elasticsearch/data
 ports:
 - 9300:9300 #B
 - 9200:9200 #C
kibana:
 image: docker.elastic.co/kibana/kibana:7.7.0 #D
 container_name: kibana
 environment:
 ELASTICSEARCH_URL: "http://elasticsearch:9300" #E
 ports:
 - 5601:5601 #F
logstash:
 image: docker.elastic.co/logstash/logstash:7.7.0 #G
 container_name: logstash
 command: logstash -f /etc/logstash/conf.d/logstash.conf #H
 volumes:
 - ./config:/etc/logstash/conf.d. #I
 ports:
 - "5000:5000" #J
#rest of the docker-compose.yml remove for conciseness
```

#A Indicates the Elasticsearch image where are going to be using for our container. In this case we are going to use the 7.7.0 version.

#B Maps the 9300 port as the cluster communication port.

#C Maps the 9200 port as the REST communication port.

#D Indicates the Kibana image where are going to be using.

#E Sets the Elasticsearch URL and indicates the node/transport API 9300 port.

#F Maps the Kibana Web Application port.

#G Indicates the Logstash image where are going to be using.

#H Loads the Logstash config from a specific file or directory.

#I Mounts the configuration file in a the Logstash running container.

#J Maps the port for Logstash.

**NOTE** The previous code listing 11.4 contains a small part of the docker-compose.yml file for this chapter. In case you want to see the complete file, you can visit the following link. <https://github.com/ihuaylupo/manning-smia/tree/master/chapter11/docker>

Remember, to run the docker environment, we need to execute the following commands in the root directory where the parent pom.xml is located. The mvn command will create a new image with the changes we made for the organization, licensing, and gateway services.

```
mvn clean package dockerfile:build
docker-compose -f docker/docker-compose.yml up
```

**NOTE** If you see the following error 137 exit code with <container\_name> container on your console while executing the docker-compose command. Please visit the following link, to increase the memory on your Docker.

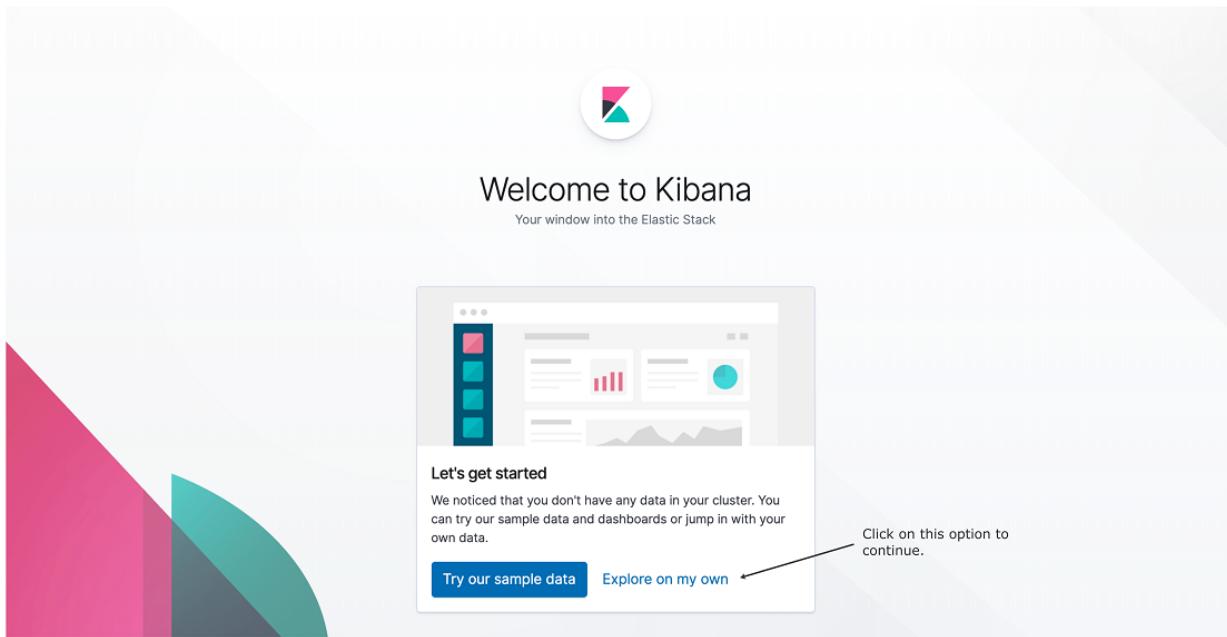
<https://www.petefreitag.com/item/848.cfm>

Now, that we have all of our docker environment running, let's configure move on with the next step.

## 11.2.4 Configuring Kibana

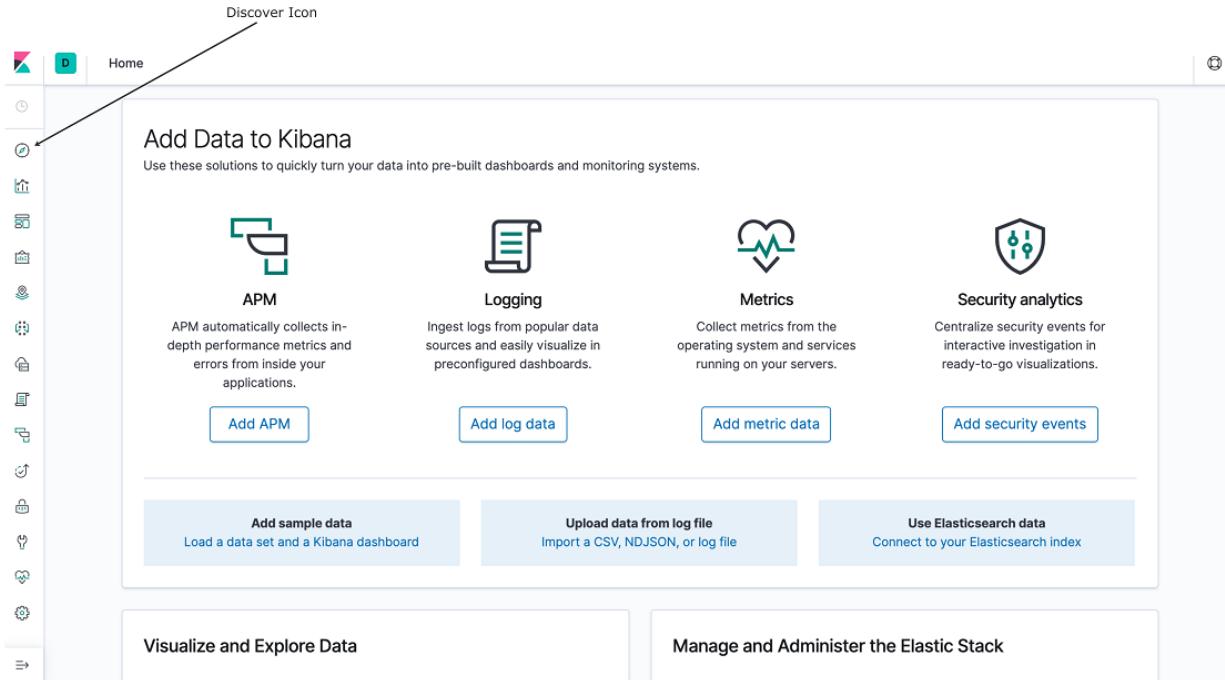
Configuring Kibana is a straightforward process, and it's important to highlight that we will only need to configure it the first time. To access Kibana just open a web browser with the following link. <http://localhost:5601/>

The first time we access Kibana, a welcome page will be displayed. This page will display two options. The first one is to allow us to play with some sample data, and the other one is to explore the data that has been generated by our services. Figure 11.7 shows Kibana's welcome page.



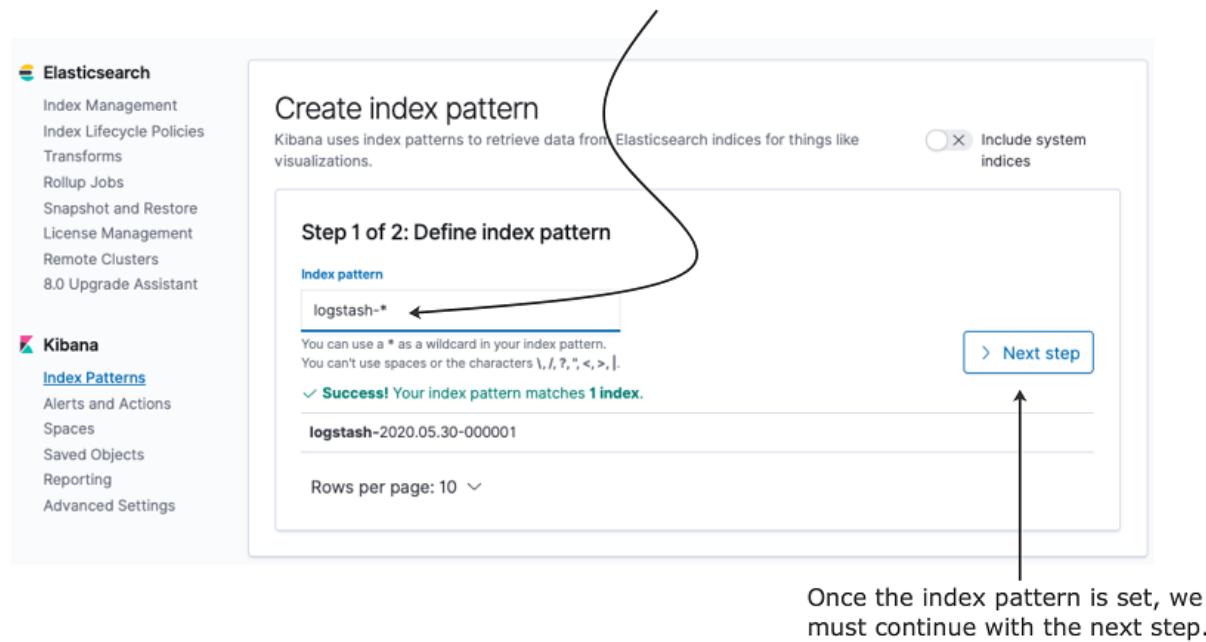
**Figure 11.7 Kibana's welcome page offers two options: Try out the application with a sample data or to explore on our own.**

To continue exploring our data, let's click the Explore on my own link. Once clicked, we will see an Add Data page like the one shown in Figure 11.8. On this page, we need to click the discover icon that appears on the icon menu on the left side of the page



**Figure 11.8 Kibana's setup page. At this page we will see a set of options we can configure in our Kibana application.**

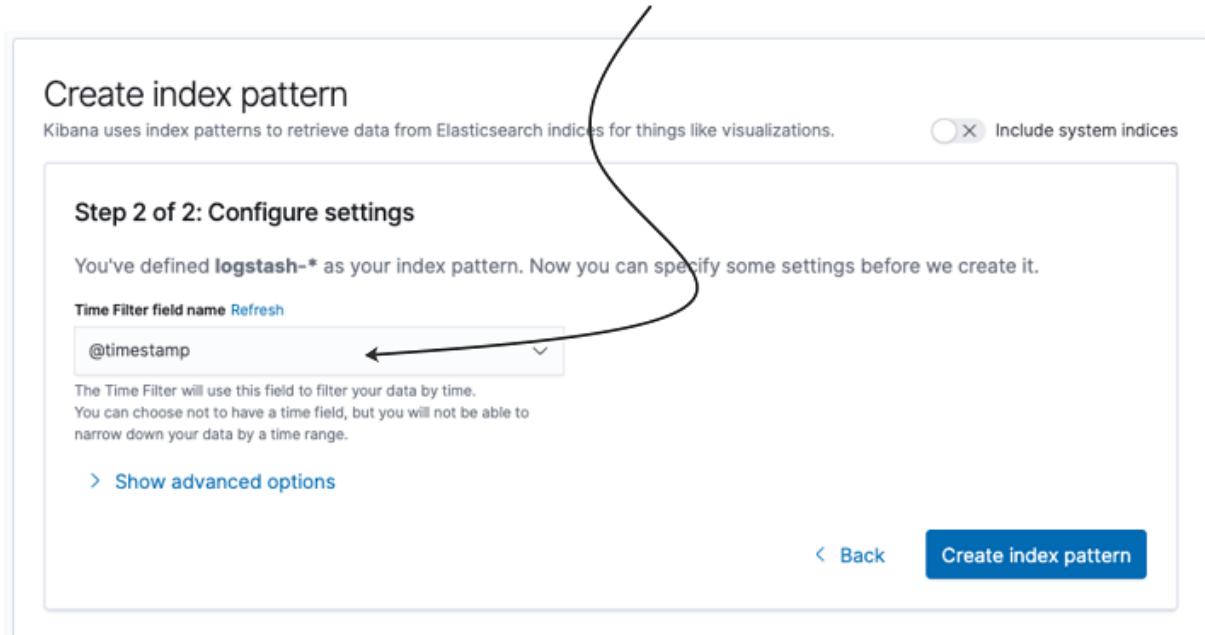
Kibana uses a set of index patterns to retrieve the data from the Elasticsearch engine. So, in order to continue, we must create an index pattern. Remember, the index pattern is in charge of telling Kibana which Elasticsearch indexes we want to explore. For example, in our case, we will create an index pattern indicating that we will retrieve all of the Logstash information from the Elasticsearch. So, lets click the Index Patterns link under the Kibana section at the left of the page shown in figure 11.9.



## Figure 11.9 Configuring the index pattern

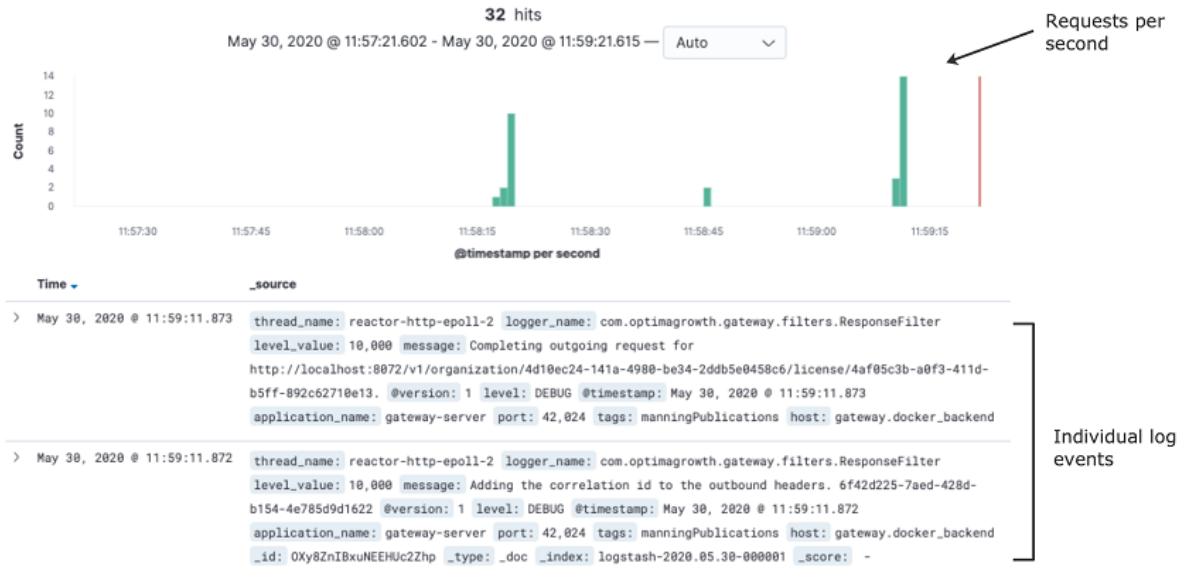
At the index patterns page, we can see that Logstash has already created an index. However, this index is not ready to use yet, to finish setting up the index, we must specify an index pattern for that index. To create it we need to write the following index pattern `logstash-*` and click the next step button shown in figure 11.9.

In the next step, we are going to specify a time filter. To achieve this, we only have to select the @timestamp option under the time filter field dropdown list and then click the create index pattern. Figure 11.10 shows this process.



**Figure 11.10 Configuring the timestamp filter for our index pattern. This index will allow us to filter events using a time range.**

We can now start making requests to our services to see the real-time logs in Kibana. Figure 11.11 shows an example of what the data sent to ELK should look like. If you don't see figure 11.11 page, just click again the discover icon.



**Figure 11.11 Individual service log events are stored, analyzed and displayed by the ELK stack.**

At this point, we're all set up with Kibana. Let's continue with our final step.

## 11.2.5 Searching for Spring Cloud Sleuth Trace IDs in Kibana

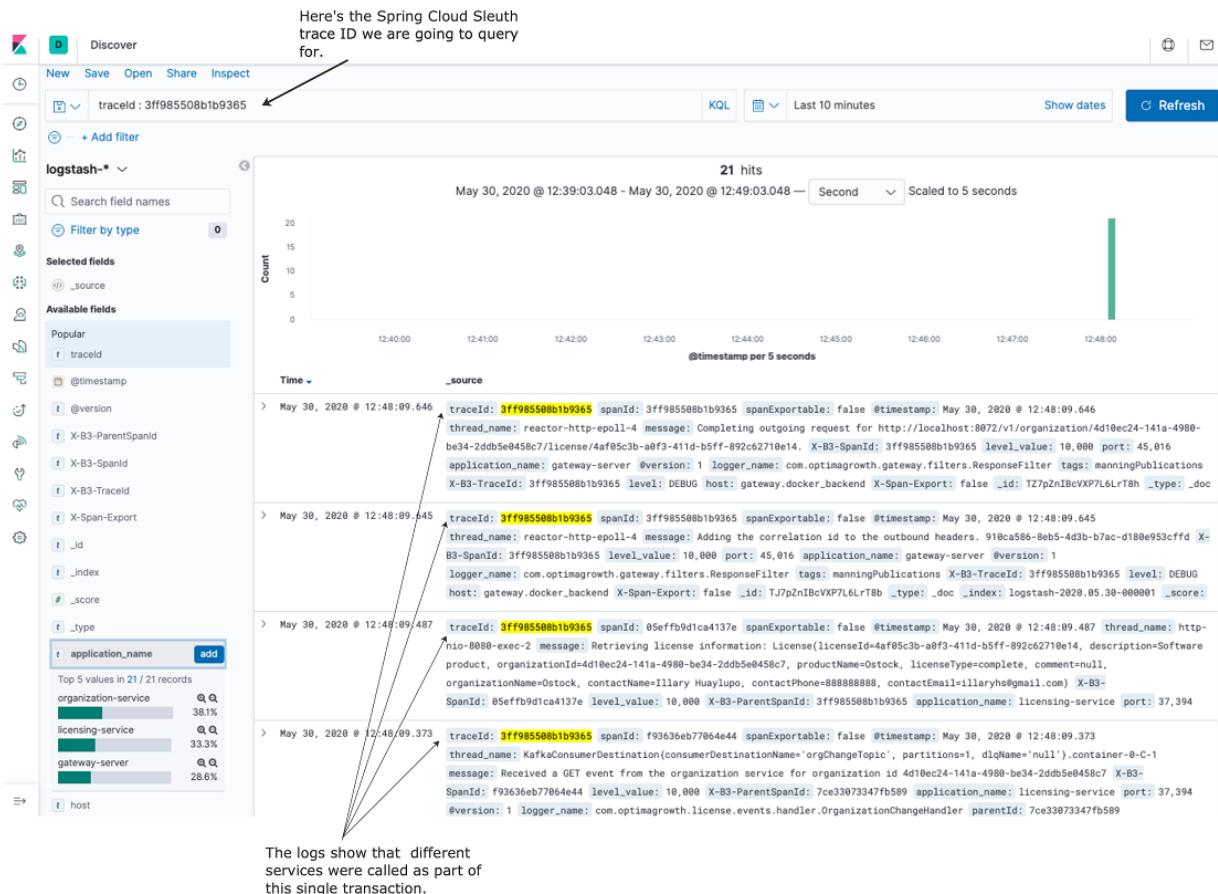
Now that our logs are flowing to ELK, we can really start appreciating Spring Cloud Sleuth, adding trace IDs to all our log entries. To query for all the log entries related to a single transaction, we need to take a trace ID, and query it on the Kibana's discover screen.

Kibana uses by default the Kibana Query Language (KQL). This language is a simplified query syntax. While writing our specific query, you'll see that Kibana also provides a guide

and autocomplete option to simplify the process of creating our custom queries.

**NOTE** In order to apply the next filter you need to select a valid trace id. The trace ID I will use on the next example is not going to work on your Kibana.

Figure 11.12 shows how to execute a query with the Spring Cloud sleuth trace ID. I will use the following Trace Id 3ff985508b1b9365.



**Figure 11.13 The trace ID allows you to filter all log entries related to that single transaction.**

We can expand each log event if we want to see more details about a specific event. By doing this, all the fields associated with that event will be displayed in a table or JSON format. We will also be able to see all the additional info we added or transform during the Logstash processing. For example, the tag we added with the mutate filter in Logstash. The following figure 11.14, shows all the fields for an event.

```

 @timestamp May 30, 2020 @ 12:48:09.487
 @version 1
 X-B3-ParentSpanId 3ff985508b1b9365
 X-B3-SpanId 05effb9d1ca4137e
 X-B3-TraceId 3ff985508b1b9365
 X-Span-Export false
 _id SS7pZnIBcVXP7L6LrD-B
 _index logstash-2020.05.30-000001
 _score -
 _type _doc
 application_name licensing-service
 host licensing-service.docker_backend
 level DEBUG
 level_value 10,000
 logger_name com.optimagrowth.license.service.LicenseService
 message Retrieving license information: License(licenseId=4af05c3b-a0f3-411d-b5ff-892c62710e14, description=Software product, organizationId=d10ec24-141a-4988-be34-2ddb5e0458c7, productName=Ostock, licenseType=complete, comment=null, organizationName=Ostock, contactName=Illary Huaylupo, contactPhone=8888888888, contactEmail=illaryhs@gmail.com)
 parentId 3ff985508b1b9365
 port 37,394
 spanExportable false
 spanId 05effb9d1ca4137e
 tags manningPublications
 thread_name http-nio-8080-exec-2
 traceId 3ff985508b1b9365

```

- `_index`: Index pattern with timestamp.
- `application_name`: Spring application name.
- `logger_name`: Logger class
- `spanId`: Spring Cloud Sleuth Span ID
- `tags`: Tag added with Logstash.
- `traceId`: Spring Cloud Sleuth Trace ID

**Figure 11.14 Detailed view of all the fields of a log event.**

## 11.2.6 Adding the correlation ID to the HTTP response with Spring Cloud Gateway

If we inspect the HTTP response back from any service call made with Spring Cloud Sleuth, we'll see that the trace ID used in the call is never returned in the HTTP response headers. If we inspect the documentation for Spring Cloud Sleuth, we'll see that the Spring Cloud Sleuth team believes that returning any of the tracing data can be a potential security issue (though they don't explicitly list their reasons why they believe this.)

However, I've found that the returning of a correlation or tracing ID in the HTTP response is invaluable when debugging a problem. Spring Cloud Sleuth does allow us to "decorate" the HTTP response information with its tracing and span IDs. However, the process to do this involves writing three classes and injecting two custom Spring beans. If you'd like to take this approach, you can see it in the Spring Cloud Sleuth documentation (<https://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.0.12.RELEASE/>). A much simpler solution is to write a Spring Cloud Gateway filter that will inject the trace ID in the HTTP response.

In chapter 8, when we introduced the Spring Cloud gateway, we saw how to build a gateway response filter to add the correlation ID we generated for use in our services to the HTTP response returned by the caller. We're now going to modify that filter to add the Spring Cloud Sleuth header.

To set up our gateway response filter, we need to make sure that we have the Spring cloud Sleuth dependencies in our pom.xml file.

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

The spring-cloud-starter-sleuth dependency will be used to tell Spring Cloud Sleuth that we want the gateway to participate in a Spring Cloud trace. Later in the chapter, when we introduce Zipkin, we'll see that the gateway service will be the first call in any service invocation.

Once the dependency is in place, the actual response filter is trivial to implement. The following code listing 11.5 shows the source code used to build the filter. The file is located in /gatewayserver/src/main/java/com/optimagrowth/gateway/filters/ResponseFilter.java.

### **Listing 11.5 Adding the Spring Cloud Sleuth trace ID via a response filter.**

```
package com.optimagrowth.gateway.filters;
//Imports removed for conciseness.
import brave.Tracer;
import reactor.core.publisher.Mono;
@Configuration
public class ResponseFilter {
final Logger logger = LoggerFactory.getLogger(ResponseFilter.class);
@Autowired
Tracer tracer; #A
@Autowired
FilterUtils filterUtils;
@Bean
public GlobalFilter postGlobalFilter() {
return (exchange, chain) -> {
return chain.filter(exchange).then(Mono.fromRunnable(() -> {
String traceId = tracer.currentSpan().context().traceIdString(); #B
logger.debug("Adding the correlation id to the outbound headers. {}",

```

```
traceId);
exchange.getResponse().getHeaders().add(FilterUtils.CORRELATION_ID,
traceId);
logger.debug("Completing outgoing request for {}.",
exchange.getRequest().getURI());
});
};
}
}
```

#A The Tracer class is the entry point to access trace and span ID information.

#B We're going to add a span to the HTTP Response header called tmx-correlation-ID the Spring Cloud Sleuth trace ID.

Because the gateway is now Spring Cloud Sleuth-enabled, we can access tracing information from within our ResponseFilter by autowiring in the Tracer class into the ResponseFilter. The Tracer class allows us to access information about the current Spring Cloud Sleuth trace being executed. The tracer.currentSpan().context().traceIdString() method allows us to retrieve as a String the current trace ID for the transaction underway.

It's trivial to add the trace ID to the outgoing HTTP response passing back through the gateway. This is done by calling

```
exchange.getResponse().getHeaders().add(FilterUtils.CORRELATION_ID, traceId);
```

With this code now in place, if we invoke an O-stock microservice through our gateway, we should get an HTTP response back called tmx-correlation-id with the Spring Cloud Sleuth Trace ID.

Figure 11.13 shows the results of a call to GET the licenses of an organization.

<http://localhost:8072/license/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c7/license/4af05c3b-a0f3-411d-b5ff-892c62710e14>

The screenshot shows a Postman request to the URL `http://localhost:8072/license/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c7/license/4af05c3b-a0f3-411d-b5ff-892c62710e14`. The 'Authorization' tab is selected, showing a 'Bearer Token' type. The token value is a long string of characters. The 'Headers' tab is selected, showing 10 headers. One of the headers is 'tmx-correlation-id' with the value 'c32b7a2b3f4a33b0'. An annotation points to this value with the text 'The Spring Cloud Sleuth Trace ID.'.

KEY	VALUE
transfer-encoding	chunked
X-Content-Type-Options	nosniff
X-XSS-Protection	1; mode=block
Cache-Control	no-cache, no-store, max-age=0, must-revalidate
Pragma	no-cache
Expires	0
X-Frame-Options	DENY
Content-Type	application/hal+json
Date	Sat, 30 May 2020 19:34:58 GMT
tmx-correlation-id	c32b7a2b3f4a33b0

**Figure 11.13 With the Spring Cloud Sleuth trace ID returned, we can easily query Kibana for the logs.**

## 11.3 Distributed tracing with Open Zipkin

Having a unified logging platform with correlation IDs is a powerful debugging tool. However, for the rest of the chapter, we're going to move away from tracing log entries and instead look at how to visualize the flow of transactions as they move across different microservices. A bright, concise picture can worth more than a million log entries.

Distributed tracing involves providing a visual picture of how a transaction flows across our different microservices.

Distributed tracing tools will also give a rough approximation of individual microservice response times. However, distributed tracing tools shouldn't be confused with full-blown Application Performance Management (APM) packages. These packages can provide out-of-the-box, low-level performance data on the actual code within our service and can also provide performance data beyond response time, such as memory, CPU utilization, and I/O utilization.

This is where Spring Cloud Sleuth and the OpenZipkin (also referred to as Zipkin) project shine. Zipkin (<http://zipkin.io/>) is a distributed tracing platform that allows us to trace transactions across multiple service invocations. Zipkin allows us to graphically see the amount of time a transaction takes and breaks down the time spent in each microservice involved in the call. Zipkin is an invaluable tool for identifying performance issues in a microservices architecture.

Setting up Spring Cloud Sleuth and Zipkin involve four activities:

- Adding Spring Cloud Sleuth and Zipkin JAR files to the services that capture trace data
- Configuring a Spring property in each service to point to the Zipkin server that will collect the trace data
- Installing and configuring a Zipkin server to collect the data
- Defining the sampling strategy each client will use to send tracing information to Zipkin

## 11.3.1 Setting up the Spring Cloud Sleuth and Zipkin dependencies

We've included the Spring Cloud Sleuth dependencies to our licensing, and organization services. This JAR includes the necessary Spring Cloud Sleuth libraries needed to enable Spring Cloud Sleuth within a service. Now, we have to include a new dependency, the spring-cloud-sleuth-zipkin dependency, to integrate with Zipkin. The following code listing 11.6 shows the Maven entry that should be present in the Spring Cloud Gateway, licensing, and organization services.

### **Listing 11.6 Client-side Spring Cloud Sleuth and Zipkin dependencies**

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-sleuth-zipkin</artifactId>
<dependency>
```

## 11.3.2 Configuring the services to point to Zipkin

With the JAR files in place, we need to configure each service that wants to communicate with Zipkin. We do this by setting a Spring property that defines the URL used to communicate with Zipkin. The property that needs to be set is the `spring.zipkin.baseUrl` property. This property is set in each service's configuration file located in the repository of the Spring Cloud config server, for example the

/configserver/src/main/resources/config/licensing-service.properties file for the licensing service.

You need to make sure that the value of the spring.zipkin.baseUrl is set to localhost:9411 to run it locally. However, if you want to run it with Docker you need to override this value with zipkin:9411.

```
zipkin.baseUrl: zipkin:9411
```

### Zipkin, RabbitMQ, and Kafka

Zipkin does have the ability to send its tracing data to a Zipkin server via RabbitMQ or Kafka. From a functionality perspective, there's no difference in Zipkin behavior if we use HTTP, RabbitMQ, or Kafka. With the HTTP tracing, Zipkin uses an asynchronous thread to send performance data. The main advantage of using RabbitMQ or Kafka to collect our tracing data is that if our Zipkin server is down, and tracing messages sent to Zipkin will be “enqueued” until Zipkin can pick up the data.

The configuration of Spring Cloud Sleuth to send data to Zipkin via RabbitMQ and Kafka is covered in the Spring Cloud Sleuth documentation, so we won't cover it here in any further detail.

### 11.3.3 Configuring a Zipkin Server

There are several ways to set up the Zipkin, but we are going to use a docker container with the Zipkin server. This option will allow us to avoid the creation of a new project in our architecture. To do this, we need to add the following registry to our docker-compose.yml file located in the docker folder of the project.

```
zipkin:
 image: openzipkin/zipkin
 container_name: zipkin
 ports:
 - 9411: 9411
 networks:
 backend:
 aliases:
 - "zipkin"
```

Little configuration is needed to run a Zipkin server. One of the only things we're going to have to configure when we run Zipkin is the back end data store that Zipkin will use to store the tracing data from our services. Zipkin supports four different back end data stores. These data stores are

1. In-memory data
2. MySQL: <http://mysql.com/>
3. Cassandra: <https://cassandra.apache.org/>
4. Elasticsearch: <http://elastic.co/>

By default, Zipkin uses an in-memory data store for storing tracing data. The Zipkin team recommends against using the in-memory database in a production system. The in-memory database can hold a limited amount of data, and the data is lost when the Zipkin server is shut down or lost. But for purposes of this example, I will teach you how to use Elasticsearch as a data store to use the Elasticsearch configuration we previously did. The only additional setting we need to add to use Elasticsearch is the `STORAGE_TYPE` and `ES_HOSTS` variables in the environment section. The following code shows the complete docker-compose registry.

```
zipkin:
 image: openzipkin/zipkin
 container_name: zipkin
 depends_on:
```

```
- elasticsearch
environment:
- STORAGE_TYPE=elasticsearch
- "ES_HOSTS=elasticsearch:9300"
ports:
- "9411:9411"
networks:
backend:
aliases:
- "zipkin"
```

## 11.3.4 Setting tracing levels

At this point, we have the clients configured to talk to a Zipkin server, and we have the server configured and ready to be run. We need to do one more step before we start using Zipkin and is to define how often each service should write data to Zipkin.

By default, Zipkin will only write 10% of all transactions to the Zipkin server. By leaving the default value, we are going to be sure that Zipkin will not overwhelm our logging and analysis infrastructure. The transaction sampling can be controlled by setting a Spring property on each of the services sending data to Zipkin. This property is called `spring.sleuth.sampler.percentage`. The property takes a value between 0 and 1:

- A value of 0 means Spring Cloud Sleuth won't send Zipkin any transactions.
- A value of .5 means Spring Cloud Sleuth will send 50% of all transactions.

For our purposes, we're going to send trace information for all services and all transactions. To do this, we can set the value of `spring.sleuth.sampler.percentage`, or we can replace the default Sampler class used in Spring Cloud Sleuth with

the AlwaysSampler. The AlwaysSampler can be injected as a Spring Bean into an application. But in this example, I will use the `spring.sleuth.sampler.percentage` in the configuration file of the licensing, organization, and gateway services.

```
zipkin.baseUrl: zipkin:9411
spring.sleuth.sampler.percentage: 1
```

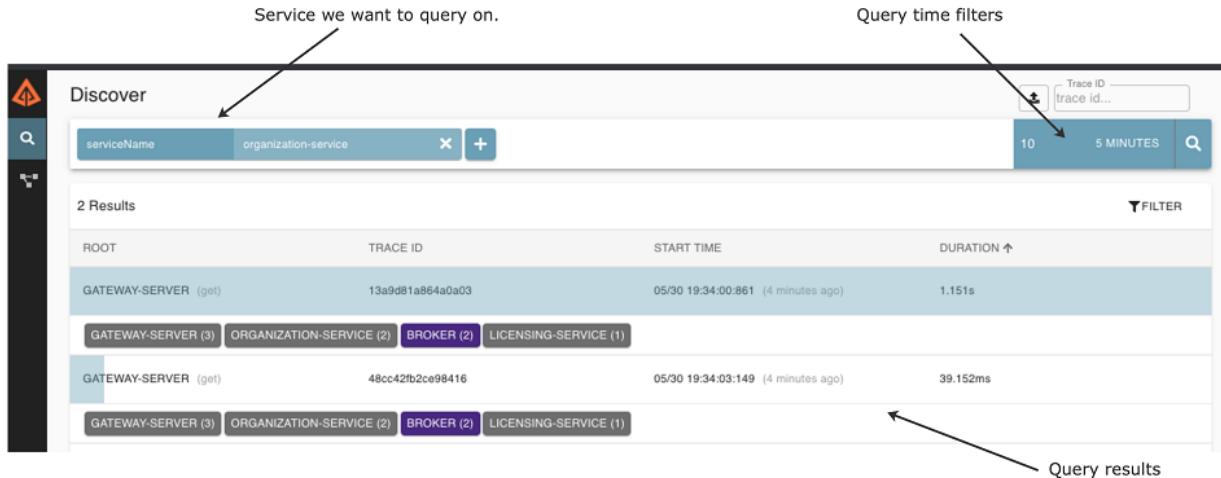
### 11.3.5 Using Zipkin to trace transactions

Let's start this section with a scenario. Imagine you're one of the developers on the O-stock application, and you're on-call this week. You get a support ticket from a customer who's complaining that one of the screens in the application is running slow. You have a suspicion that the licensing service being used by the screen is running slow. But why and where? The licensing service relies on the organization service, and both services make calls to different databases. Which service is the poor performer? Also, you know that these services are continually being modified, so someone might have added a new service call into the mix. Understanding all the services that participate in the user's transaction and their individual performance times is critical to supporting a distributed architecture such as a microservice architecture.

To solve this, we'll begin by using Zipkin to watch two transactions from our organization service as the Zipkin service traces them. The organization service is a simple

service that only makes a call to a single database. What we're going to do is use POSTMAN to send two calls to the organization service (GET <http://localhost:8072/organization/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c6>). The organization service calls will flow through the gateway before the calls get directed downstream to an organization service instance.

Now, if we look at the screenshot in figure 11.15, we'll see that Zipkin captured two transactions. Each of the transactions is broken down into one or more spans. In Zipkin, a span represents a specific service or call in which timing information is being captured. Each of the transactions in figure 11.15 has five spans captured in it: two spans in the gateway, two for the organization, and one for the licensing service. Remember, the gateway doesn't blindly forward an HTTP call. It receives the incoming HTTP call, terminates the incoming call, and then builds a new call out to the targeted service (in this case, the organization service). This termination of the original call is how the gateway can add pre-, response, and post-filters to each call entering the gateway. It's also why we see two spans in the gateway service.

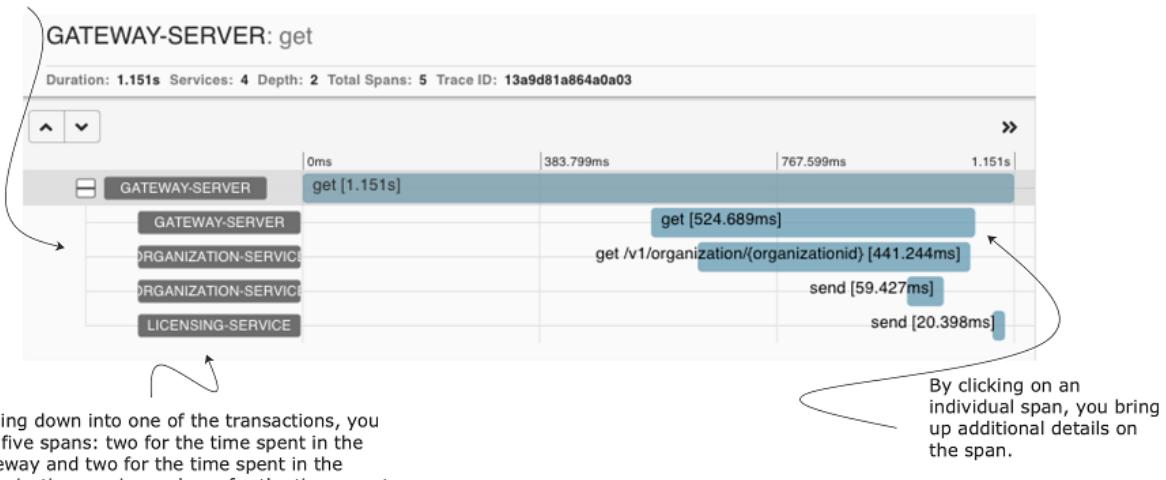


**Figure 11.15 The Zipkin query screen allow us to select the service we want to trace on, along with some basic query filters.**

The two calls to the organization service through the gateway took 1.151 seconds and 39.152 milliseconds, respectively.

Let's dig into the details of the longest-running call (1.151 seconds). We can see more detail by clicking on the transaction and drilling into the details. Figure 11.16 shows the details after we've clicked to drill down into further details.

A transaction is broken down into individual spans. A span represents part of the transaction being measured. Here the total time of each span in the transaction is displayed.



## Figure 11.16 Zipkin allows us to drill down and see the amount of time each span in a transaction takes.

In figure 11.16, we can see that the entire transaction from a gateway perspective took approximately 1.151 seconds. However, the organization service call made by the gateway took 524.689 milliseconds of the 1.151 seconds involved in the overall call. Each span presented can be drilled down into for even more detail. Click on the organization-service span and see what additional details can be seen from the call. Figure 11.17 shows the detail of this call.

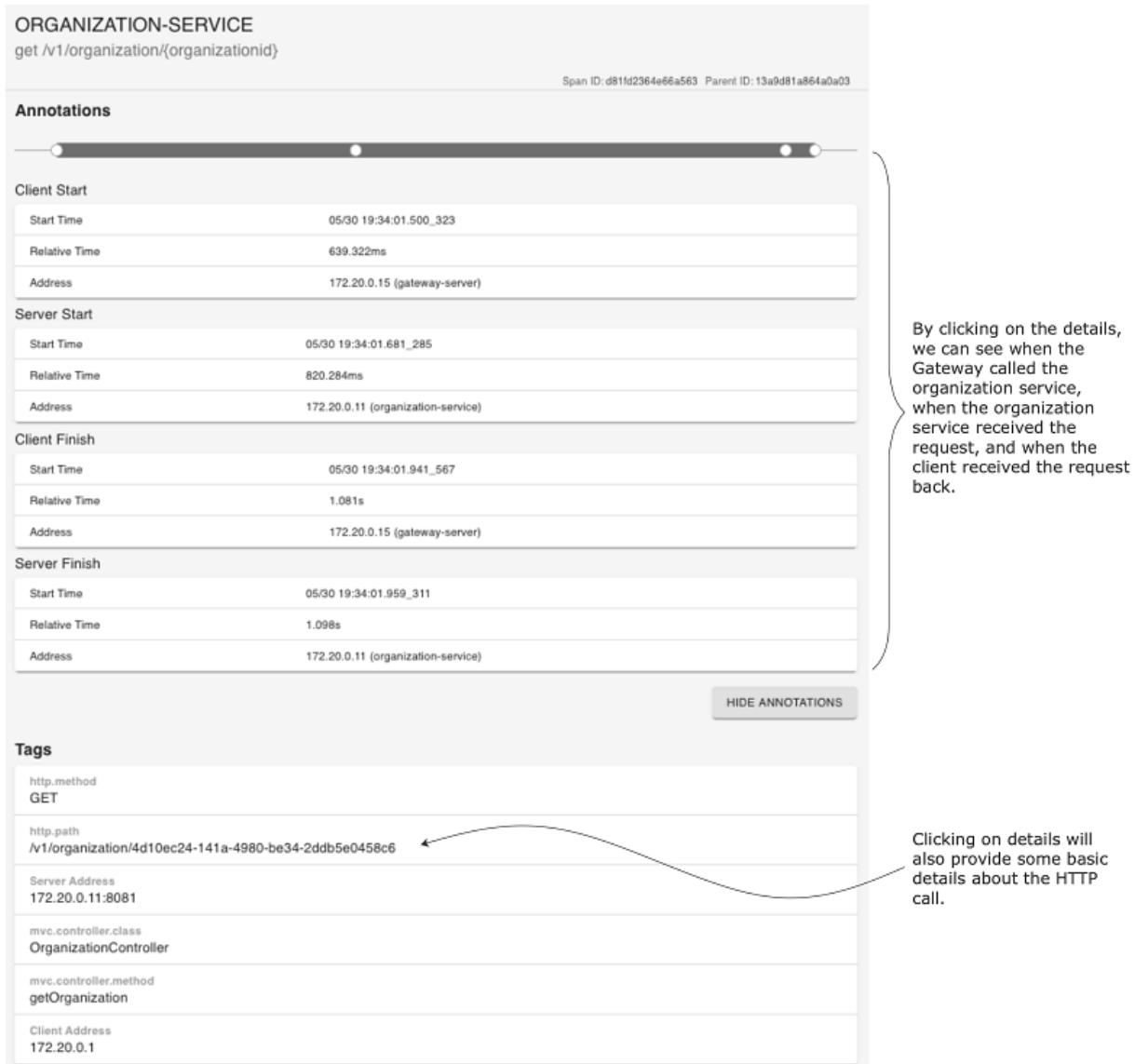
To add a custom span to the licensing service's call to Redis, we're going to instrument the `/licensing-service/src/main/java/com/optimagrowth/license/service/client/OrganizationRestTemplateClient.java` class. In this class we're going to instrument the `checkRedisCache()` method. The following code listing 11.7 shows this code.

## **Listing 11.7 Adding the CheckRedisCache method in the OrganizationRestTemplate**

```
package com.optimagrowth.license.service.client;
//Rest of imports removed for conciseness
@Component
public class OrganizationRestTemplateClient {
@Autowired
RestTemplate restTemplate;
@Autowired
Tracer tracer; #A
@Autowired
OrganizationRedisRepository redisRepository;
private static final Logger logger =
LoggerFactory.getLogger(OrganizationRestTemplateClient.class);
private Organization checkRedisCache(String organizationId) { #B
try {
return redisRepository.findById(organizationId).orElse(null);
} catch (Exception ex){
logger.error("Error encountered while trying to retrieve organization {} check
Redis Cache. Exception {}", organizationId, ex);
return null;
}
}
//Rest of class removed for conciseness
}
```

#A The Tracer class is used to programmatically access the Spring Cloud Sleuth trace information.

#B CheckRedisCache method.



**Figure 11.17 Clicking on an individual span gives further details on call timing and the details of the HTTP call.**

One of the most valuable pieces of information in figure 11.17 is the breakdown of when the client (Gateway) called the organization service when the organization service received the call, and when the organization service

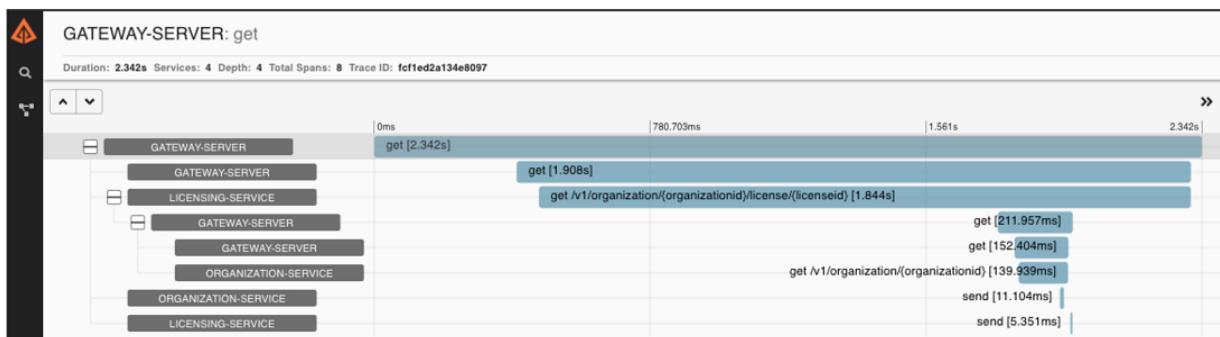
responded back. This type of timing information is invaluable in detecting and identifying network latency issues.

### 11.3.6 Visualizing a more complex transaction

What if we want to understand precisely what service dependencies exist between service calls? We can call the licensing service through the gateway and then query Zipkin for licensing service traces. We can do this with a GET call to the licensing services

<http://localhost:8072/license/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c8/license/4af05c3b-a0f3-411d-b5ff-892c62710e15> endpoint.

Figure 11.18 shows the detailed trace of the call to the licensing service.



**Figure 11.18 Viewing the details of a trace of how the licensing service call flows from the gateway to the licensing service and then to the organization service.**

In figure 11.18, we can see that the call to the licensing service involves eight HTTP calls. We see the call to the gateway and then from the gateway to the licensing service. The licensing service to the gateway and the gateway to the organization, and finally, the organization to the licensing service using the apache Kafka to update the Redis cache.

### 11.3.7 Capturing messaging traces

Messaging can introduce its own performance and latency issues inside of an application. A service might not be processing a message from a queue quickly enough. Or there could be a network latency problem. I've encountered all these scenarios while building microservice-based applications.

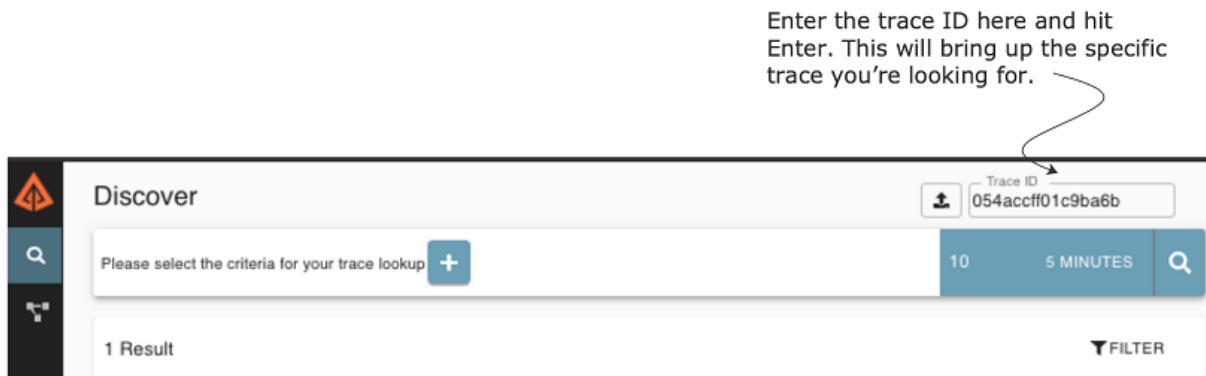
By using Spring Cloud Sleuth and Zipkin, we can identify when a message is published from a queue and when it's received. Spring Cloud Sleuth also sends Zipkin trace data on any inbound or outbound message channel registered in the service. We can also see what behavior takes place when the message is received on a queue and processed.

As we'll remember from chapter 10, whenever an organization record is added, updated, or deleted, a Kafka message is produced and published via Spring Cloud Stream. The licensing service receives the message and updates a Redis key-value store it's using to cache data.

Now we'll go ahead and delete an organization record and watch Spring Cloud Sleuth and Zipkin trace the transaction. We can issue a DELETE

<http://localhost:8072/organization/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c7> via POSTMAN to the organization service.

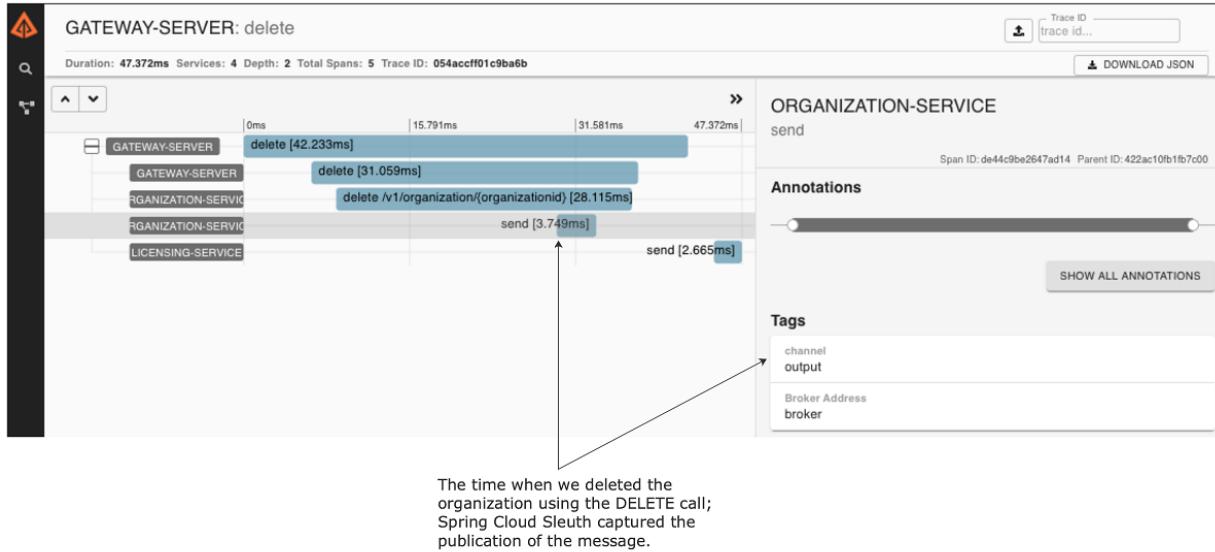
Remember, earlier in the chapter. We saw how to add the trace ID as an HTTP response header. We added a new HTTP response header called tmx-correlation-id. In my call, I had the tmx-correlation-id returned on my call with a value of 054accff01c9ba6b. We can search Zipkin for this specific trace ID by entering the trace ID returned by our call via the search box in the upper-right hand corner of the Zipkin query screen. Figure 11.19 shows where we can enter the trace ID.



**Figure 11.19 With the trace ID returned in the HTTP Response tmx-correlation-id filed we can easily find the transaction we are looking for.**

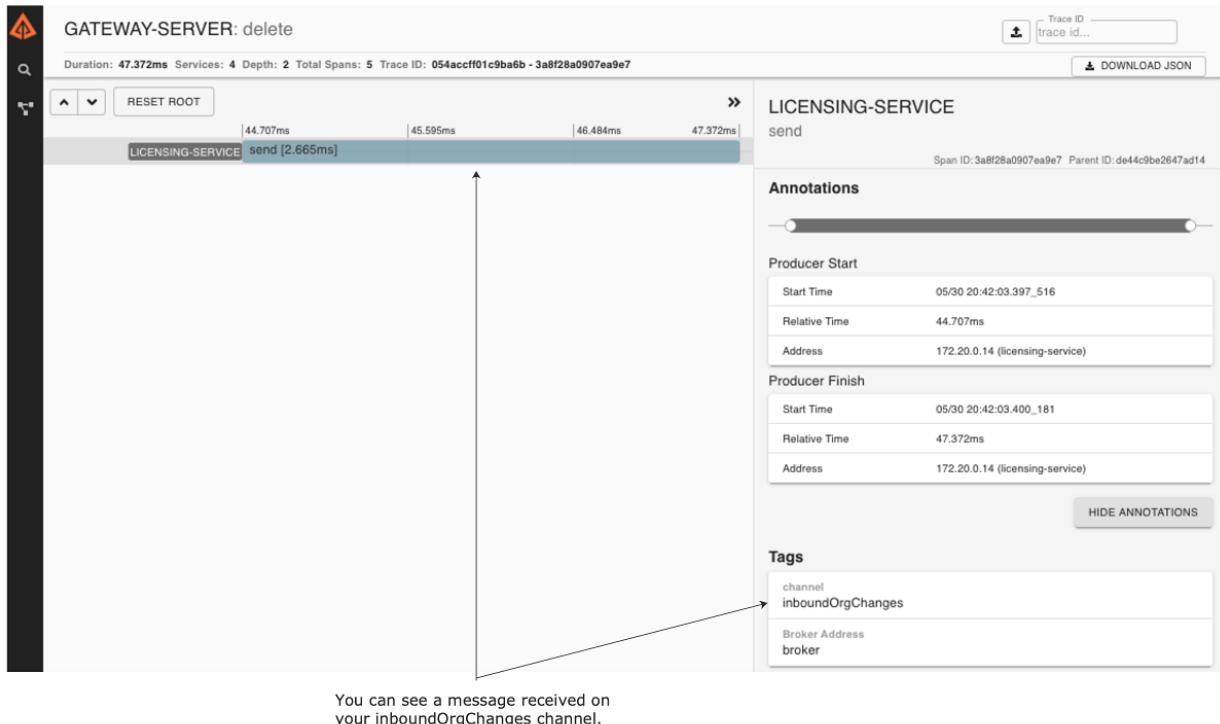
With the trace ID in hand, we can query Zipkin for the specific transaction and can see the publication of a delete message to our output message change. The second span is the message channel, output, and is used to publish to a Kafka topic call orgChangeTopic. Figure 11.20 shows the

output message channel and how it appears in the Zipkin trace.



**Figure 11.20 Spring Cloud Sleuth will automatically trace the publication and receipt of messages on Spring message channels.**

You can see the licensing service receive the message by click the licensing service span. Figure 11.21 shows the licensing span data.



**Figure 11.21 Using zipkin we can see the Kafka message being published by the organization service.**

Until now, we've used Zipkin to trace our HTTP and messaging calls from within our services. However, what if we want to perform traces out to third-party services that aren't instrumented by Zipkin? For example, what if we want to get tracing and timing information for a specific Redis or Postgres SQL call? Fortunately, Spring Cloud Sleuth and Zipkin allow us to add custom spans to our transaction so that we can trace the execution time associated with these third-party calls.

### 11.3.8 Adding custom spans

Adding a custom span is incredibly easy to do in Zipkin. We can start by adding a custom span to our licensing service so that we can trace how long it takes to pull data out of Redis. Then we're going to add a custom span to the organization service to see how long it takes to retrieve data from our organization database.

The code listing 11.8, creates a custom span called `readLicensingDataFromRedis`.

### **Listing 11.8 Adding the Spring Cloud Sleuth trace ID via a response filter.**

```
package com.optimagrowth.license.service.client;
//Rest of code removed for conciseness
private Organization checkRedisCache(String organizationId) {
 ScopedSpan newSpan = tracer.startScopedSpan("readLicensingDataFromRedis"); #A
 try {
 return redisRepository.findById(organizationId).orElse(null);
 }catch (Exception ex){
 logger.error("Error encountered while trying to retrieve organization {} check
Redis Cache. Exception {}", organizationId, ex);
 return null;
 }finally {
 newSpan.tag("peer.service", "redis"); #B
 newSpan.annotate("Client received");
 newSpan.finish(); #C
 }
}
```

#A Creates a new span called “`readLicensingDataFromRedis`”.  
#B We can add tag information to the span. In this class we provide the name of the service that's going to be captured by Zipkin.  
#C Close and finish the span. If not, we will get error messages in the logs indicating that a span has been left open.

Now we'll also add a custom span, called `getOrgDbCall`, to the organization service to monitor how long it takes to retrieve organization data from the Postgres database. The trace for organization service database calls can be seen in

the /organization-service/src/main/java/com/optimagrowth/organization/service/OrganizationService.java class. The method containing the custom trace is the findById() method call.

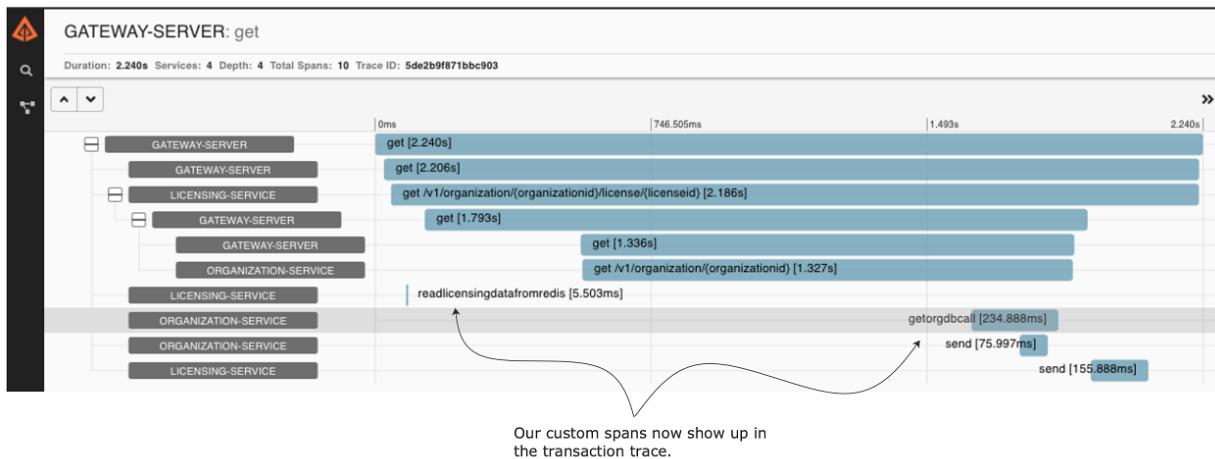
The following code listing 11.9 shows the source code from the organization service's findById() method.

### **Listing 11.9 The instrumented findById() method**

```
package com.optimagrowth.organization.service;
//Rest of imports removed for conciseness
import brave.ScopedSpan;
import brave.Tracer;
@Service
public class OrganizationService {
//Rest of the code removed for conciseness
@Autowired
Tracer tracer;
public Organization findById(String organizationId) {
Optional<Organization> opt = null;
ScopedSpan newSpan = tracer.startScopedSpan("getOrgDBCall");
try {
opt = repository.findById(organizationId);
simpleSourceBean.publishOrganizationChange("GET", organizationId);
if (!opt.isPresent()) {
String message = String.format("Unable to find an organization with the
Organization id %s", organizationId);
logger.error(message);
throw new IllegalArgumentException(message);
}
logger.debug("Retrieving Organization Info: " + opt.get().toString());
}finally {
newSpan.tag("peer.service", "postgres");
newSpan.annotate("Client received");
newSpan.finish();
}
return opt.get();
}
//Rest of class removed for conciseness
}
```

With these two custom spans in place, restart the services and then hit the GET

<http://localhost:8072/license/v1/organization/4d10ec24-141a-4980-be34-2ddb5e0458c9/license/4af05c3b-a0f3-411d-b5ff-892c62710e16> endpoint. If we look at the transaction in Zipkin, we should see the addition of the two new spans. Figure 11.22 shows the additional custom spans added when we call the licensing service endpoint to retrieve licensing information.



**Figure 11.22 The custom defined spans now show up in the transaction trace.**

Figure 11.22 shows additional tracing and timing information related to our Redis and database lookups. We can break out that the read call to Redis took 5.503 milliseconds. Since the call didn't find an item in the Redis cache, the SQL call to the Postgres database took 234.888 milliseconds.

Now that we know how to set up a distributed tracing, API gateway, discovery service, and more, let's continue with the next chapter. In the next chapter, I will explain how to deploy everything that we've been building throughout the book.

## 11.4 Summary

- Spring Cloud Sleuth allows us to seamlessly add tracing information (correlation ID) to our microservice calls.
- Correlation IDs can be used to link log entries across multiple services. They allow us to see the behavior of a transaction across all the services involved in a single transaction.
- While correlation IDs are powerful, we need to partner this concept with a log aggregation platform that will allow us to ingest logs from multiple sources and then search and query their contents.
- We can integrate Docker containers with a log aggregation platform to capture all the application logging data.
- In this chapter, we integrated our Docker containers with the ELK (Elasticsearch, Logstash, Kibana) stack. That allows us to transform, store, visualize, and query the logging data of our services.
- While a unified logging platform is essential, the ability to visually trace a transaction through its microservices is also a valuable tool.
- Zipkin allows us to see the dependencies that exist between services when a call to a service is made.
- Spring Cloud Sleuth integrates with Zipkin. Zipkin allows us to graphically see the flow of our transactions and understand the performance characteristics of each microservice involved in a user's transaction.
- Spring Cloud Sleuth will automatically capture trace data for an HTTP call, and inbound/outbound message channel used within a Spring Cloud Sleuth enabled service.
- Spring Cloud Sleuth maps each of the service call to the concept of a span. Zipkin allows us to see the performance of a span.
- Spring Cloud Sleuth and Zipkin also allow us to define our own custom spans so that we can understand the

performance of non-Spring-based resources (a database server such as Postgres or Redis).

# 12 Deploying your microservices

## This chapter covers

- Understanding why the DevOps movement is critical to microservices
- Configuring the core Amazon infrastructure used by Ostock services
- Manually deploying Ostock services to Amazon
- Designing a build and deployment pipeline for your services
- Treating your infrastructure as code
- Deploying your application to the cloud

We're almost at the end of the book, but not the end of our microservices journey. While most of this book has focused on designing, building, and operationalizing Spring-based microservices using the Spring Cloud technology, we haven't yet touched on how to build and deploy microservices. Creating a build and deployment pipeline might seem like a mundane task, but in reality, it's one of the most critical pieces of our microservices architecture.

Why? Remember, one of the key advantages of a microservices architecture is that microservices are small units of code that can be quickly built, modified, and deployed to production independently of one another. The

small size of the service means that new features (and critical bug fixes) can be delivered with a high degree of velocity. Velocity is the keyword here because velocity implies that little to no friction exists between making a new feature or fixing a bug and getting our service deployed. Lead times for deployment should be minutes, not days.

To accomplish this, the mechanism that you use to build and deploy your code needs to be

- **Automated.** When we build our code, there should be no human intervention in the build and deployment process. The process of building the software, provisioning a machine image, and deploying the service should be automated and initiated by the act of committing code to the source repository.
- **Repeatable.** The process we use to build and deploy our software should be repeatable so that the same thing happens every time a build and deploy kicks off. Variability in our process is often the source of subtle bugs that are difficult to track down and resolve.
- **Complete.** The outcome of our deployed artifact should be an entire virtual machine or container image (Docker) that contains the “complete” runtime environment for the service. This is an important shift in the way we think about our infrastructure. The provisioning of our machine images needs to be completely automated via scripts and kept under source control with the service source code. In a microservice environment, this responsibility usually shifts from an operations team to the development team owning the service. Remember, one of the core tenants of microservice development is pushing the developers to complete operational responsibility for the service.
- **Immutable.** Once the machine image containing our service is built, the image's runtime configuration should

not be touched or changed after the image has been deployed. If changes need to be made, the configuration needs to happen in the scripts kept under source control, and the service and infrastructure need to go through the build process again. Runtime configuration changes (garbage collection settings, Spring profile being used) should be passed as environment variables to the image while application configuration should be kept separate from the container (Spring Cloud Config).

Building a robust and generalized build deployment pipeline is a significant amount of work and is often explicitly designed toward the runtime environment our services are going to run. It usually involves a specialized team of DevOps (developer operations) engineers whose sole job is to generalize the build process so that each team can build their microservices without having to reinvent the entire build process for themselves. Unfortunately, Spring is a development framework and doesn't offer a significant number of capabilities to implement a build and deployment pipeline.

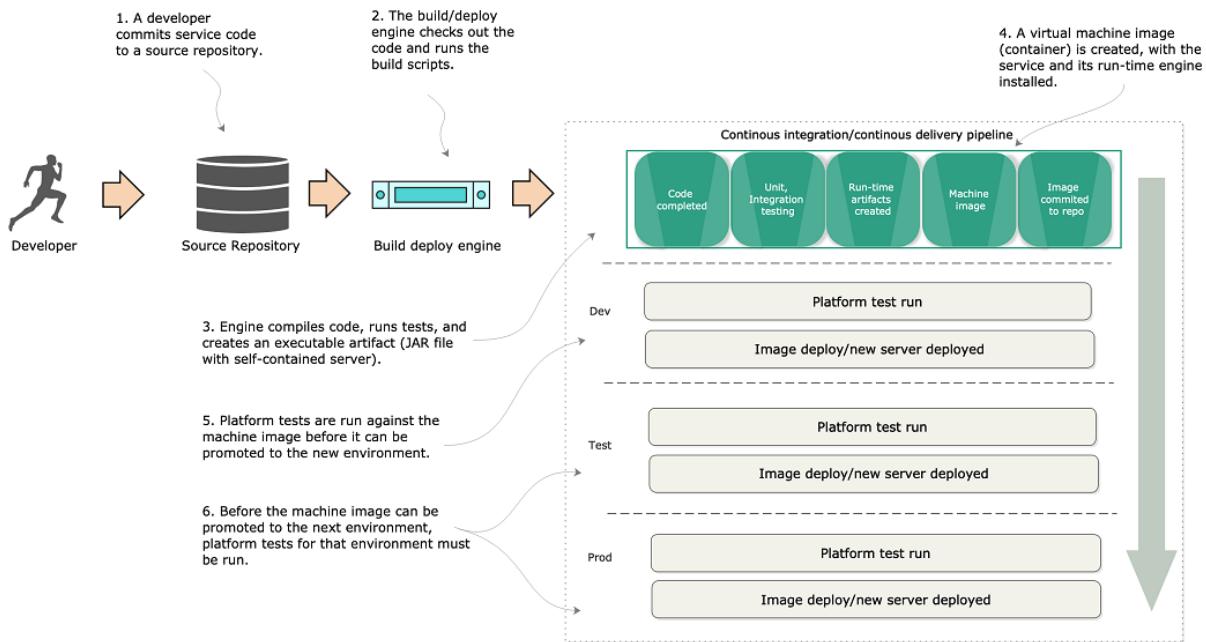
## **12.1 The architecture of a build/deployment pipeline**

The goal of this chapter is to provide you with the working pieces of a build/deployment pipeline so that you can take these pieces and tailor them to your specific environment.

Let's start our discussion by looking at the general architecture of our build deployment pipeline and several of the general patterns and themes that it represents. To keep

the examples flowing, I've done a few things that I wouldn't normally do in my own environment, and I'll call those pieces out accordingly.

Our discussion on deploying microservices will begin with a picture you saw way back in chapter 1. Figure 12.1 is a duplicate of the diagram we saw in chapter 1 and shows the pieces and steps involved in building a microservices build and deployment pipeline.



**Figure 12.1 Each component in the build and deployment pipeline automates a task that would have been manually done.**

Figure 12.1 should look somewhat familiar because it's based on the general build-deploy pattern used for implementing Continuous Integration (CI):

1. A developer commits their code to the source code repository.
2. A build tool monitors the source control repository for changes and kicks off a build when a change is detected.
3. During the build, the application's unit and integration tests are run, and if everything passes, a deployable software artifact is created (a JAR, WAR, or EAR).
4. This JAR, WAR, or EAR might then be deployed to an application server running on a server (usually a development server).

A similar process is followed up with the build and deployment pipeline until the code is ready to be deployed. In the build and deployment shown in figure 12.1, you're going to tack Continuous Delivery (CD) onto the process:

1. A developer commits their service code to a source repository.
2. A build/deploy engine monitors the source code repository for changes. If code is committed, the build/deploy engine will check out the code and run the code's build scripts.
3. The first step in the build/deploy process is to compile the code, run its unit and integration tests, and then compile the service to an executable artifact. Because your microservices are built using Spring Boot, your build process will create an executable JAR file that contains both the service code and self-contained Tomcat server.

4. This is where your build/deploy pipeline begins to deviate from a traditional Java CI build process. After your executable JAR is built, you're going to "bake" a machine image with your microservice deployed to it. This baking process will create a virtual machine image or container (Docker) and install your service onto it. When the virtual machine image is started, your service will be started and will be ready to begin taking requests. Unlike a traditional CI build process where you might deploy the compiled JAR or WAR to an application server that's independently (and often with a separate team) managed from the application, with the CI/CD process you're deploying the microservice, the runtime engine for the service, and the machine image all as one co-dependent unit that's managed by the development team that wrote the software.
5. Before you officially deploy to a new environment, the machine image is started, and a series of platform tests are run against the running image to determine if everything is running correctly. If the platform tests pass, the machine image is promoted to the new environment and made available for use.
6. Before a service is promoted to the next environment, the platform tests must be run for the environment. The promotion of the service to the new environment involves starting up the exact machine image that was used in the lower environment to the next environment. This is the secret sauce of the whole process. The entire machine image is deployed. No changes are made to any installed software (including the operating system) after the server is created. By

promoting and always using the same machine image, you guarantee the immutability of the server as it's promoted from one environment to the next).

---

## **Unit tests vs. Integration tests vs. platform test**

Figure 12.1 exposes several types of testing (unit, integration, and platform) during the build and deployment of a service. Three types of testing are typical in a build and deployment pipeline:

**Unit tests**— Unit tests are run immediately before the compilation of the service code, but before it's deployed to an environment. They're designed to run in complete isolation, with each unit test being small and narrow in focus. A unit test should have no dependencies on third-party infrastructure databases, services, and so on. Usually, a unit test scope will encompass the testing of a single method or function.

**Integration tests**—Integration tests are run immediately after packaging the service code. These tests are designed to test an entire workflow and stub or mock out primary services or components that would need to be called off box. During an integration test, you might be running an in-memory database to hold data, mocking out third-party service calls, and so on. Integration tests test an entire workflow or code path. For integration tests, third-party dependencies are mocked or stubbed so that any requests that would invoke a remote service are mocked or stubbed, so that calls never leave the build server.

**Platform tests**—Platform tests are run right before a service is deployed to an environment. These tests typically test an entire business flow and also call all the third-party dependencies that would usually be called in a production system. Platform tests are running live in a particular environment and don't involve any mocked-out services. Platform tests are run to determine integration problems with third-party services that would typically not be detected when a third-party service is stubbed out during an integration test.

If you are interested in diving into more detail on how to create unit, integration, and platform tests, I highly recommend Alex Soto Bueno, Andy Gumbrecht, and Jason Porter's book, *Testing Java Microservices* (Manning, 2018).

---

This build/deploy process is built on four core patterns. These patterns aren't my creation but have emerged from the collective experience of development teams building

microservice and cloud-based applications. These patterns include

- **Continuous Integration/Continuous Delivery (CI/CD).** With CI/CD, your application code isn't only being built and tested when it is committed; it's also consistently being deployed. The deployment of your code should go something like this: if the code passes its unit, integration, and platform tests, it should be immediately promoted to the next environment. The only stopping point in most organizations is the push to production.
- **Infrastructure as code.** The final software artifact that will be pushed to development and beyond is a machine image. The machine image and your microservice installed on it will be provisioned time immediately after your microservice's source code is compiled and tested. The provisioning of the machine image occurs through a series of scripts that are run with each build. No human hands should ever touch the server after it's been built. The provisioning scripts are kept under source control and managed like any other piece of code.
- **Immutable servers.** Once a server image is built, the server's configuration and microservice are never touched after the provisioning process. This guarantees that your environment won't suffer from "configuration drift," where a developer or system administrator made "one small change" that later caused an outage. If a change needs to be made, the provisioning scripts that provision the server are changed, and a new build is kicked off.

---

### On immutability and the rise of the Phoenix server

With the concept of immutable servers, we should always be guaranteed that a server's configuration matches precisely with what the machine image for the server says it does. A server should have the option to be killed and restarted from the machine image without any changes in the service or microservices

behavior. This killing and resurrection of a new server was termed “Phoenix Server” by Martin Fowler (<http://martinfowler.com/bliki/PhoenixServer.html>) because when the old server is killed, the new server should rise from the ashes. The Phoenix server pattern has two fundamental benefits.

First, it exposes and drives configuration drift out of your environment. If you’re constantly tearing down and setting up new servers, you’re more likely to expose configuration drift early. This is a tremendous help in ensuring consistency.

Second, the Phoenix server pattern helps to improve resiliency by helping find situations where a server or service isn’t cleanly recoverable after it has been killed and restarted. Remember, in a microservice architecture, your services should be stateless, and the death of a server should be a minor blip. Randomly killing and restarting servers quickly exposes situations where you have a state in your services or infrastructure. It’s better to find these situations and dependencies early in your deployment pipeline than when you’re on the phone with an angry company.

---

For this chapter, we’re going to see how to implement a build and deployment pipeline using several non-Spring tools.

We’re going to take the suite of microservices we’ve been building for this book and do the following:

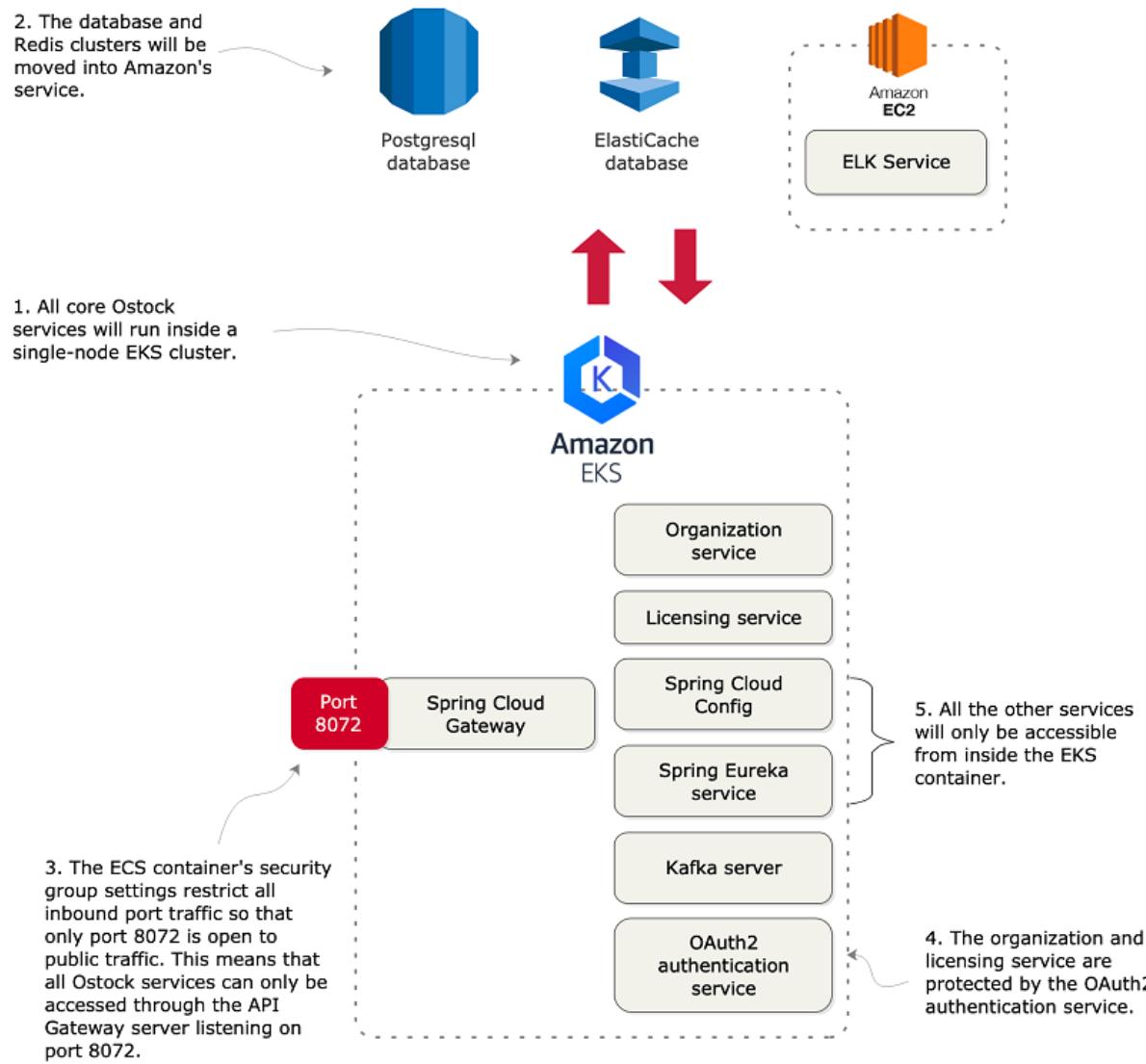
1. Integrate the Maven build scripts we’ve been using into a continuous integration/deployment cloud-tool called Jenkins
2. Build immutable Docker images for each service and push those images to a centralized repository
3. Deploy the entire suite of microservices to Amazon’s Cloud using Amazon’s Container Service for Kubernetes (EKS)

**NOTE** In this book, I will not explain in detail how Kubernetes works. In case you are new or want to learn more about how it works, I highly recommend you review Marko Lukša’s excellent book called *Kubernetes in Action*, which covers this subject in detail.

Before we start with our pipeline, let's make a pause and start with setting up the core infrastructure in the cloud.

## 12.2 Ostock: setting up the core infrastructure in the cloud

Throughout all the code examples in this book, we've run all of our applications inside a single virtual machine image with each individual service running as a Docker container. We're going to change that now by separating our database server (PostgreSQL) and caching server (Redis) away from Docker into Amazon's cloud. All the other services will remain running as Docker containers running inside a single-node Amazon EKS cluster. Figure 12.2 shows the deployment of the Ostock services to the Amazon cloud.



**Figure 12.2 By using Docker, all our services can be deployed to a cloud provider such as Amazon EKS.**

Let's walk through figure 12.2 and dive into more detail:

1. All our Ostock services (minus the database and the Redis cluster) are going to be deployed as Docker containers running inside of a single-node EKS cluster. EKS configures and sets up all the servers needed to

run a Docker cluster. EKS also can monitor the health of containers running in Docker and restart services if the service crashes.

2. With the deployment to the Amazon cloud, we're going to move away from using your own PostgreSQL database and Redis server and instead use the Amazon RDS and Amazon ElastiCache services. We could continue to run the Postgres and Redis data stores in Docker, but I wanted to highlight how easy it is to move from infrastructure that's owned and managed by us to infrastructure managed completely by the cloud provider (in this case, Amazon).
3. Unlike our desktop deployment, we want all traffic for the server to go through our API gateway. We're going to use an Amazon security group to only allow port 8072 on the deployed EKS cluster to be accessible to the world.
4. We'll still use Spring's OAuth2 server to protect our services. Before the organization and licensing services can be accessed, the user will need to authenticate with our authentication services (see chapter 9 for details on this) and present a valid OAuth2 token on every service call.
5. All our servers, including our Kafka server, won't be publicly accessible to the outside world via their exposed Docker ports.

---

## Some prerequisites for working

While To set up your Amazon infrastructure, you're going to need the following:

1. **Your own Amazon Web Services (AWS) account.** You should have a basic understanding of the AWS console and the concepts behind working in the

environment.

2. **A web browser.** For the manual setup, you're going to set up everything from the console.
3. **AWS CLI.** Unified tool to manage our AWS Services.  
<https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html>
4. **Kubectl.** Tool that allow us to communicate and interact with our Kubernetes cluster. <https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl>
5. **IAM Authenticator.** Tool to provide authentication to our Kubernetes cluster.  
<https://docs.aws.amazon.com/eks/latest/userguide/install-aws-iam-authenticator.html>
6. **Eksctl.** Simply command line utility for managing and creating AWS EKS clusters in our AWS Account.  
<https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html>

If you're entirely new to AWS, I highly recommend you pick up a copy of Michael and Andreas Wittig's book Amazon Web Services in Action (Manning, 2018). The first chapter of the book (<https://www.manning.com/books/amazon-web-services-in-action-second-edition>) is available for download and includes a well-written tutorial at the end of the chapter on how to sign up and configure your AWS account. Amazon Web Services in Action is a well-written and comprehensive book on AWS.

Finally, in this chapter, I've tried as much as possible to use the free-tier services offered by Amazon. The only place where I couldn't do this is when setting up the EKS cluster. I used a m4.large server that costs approximately .10 cents per hour to run. Ensure that you shut down your services after you're done if you don't want to incur high costs.

**NOTE:** There's no guarantee that the Amazon resources (Postgres, Redis, and EKS) that I'm using in this chapter will be available if you want to run this code yourself. If you're going to run the code from this chapter, you need to set up your own GitHub repository (for your application configuration), your own Jenkins environment, Docker Hub (for your Docker images), and Amazon account, and then modify your application configuration to point to your account and credentials.

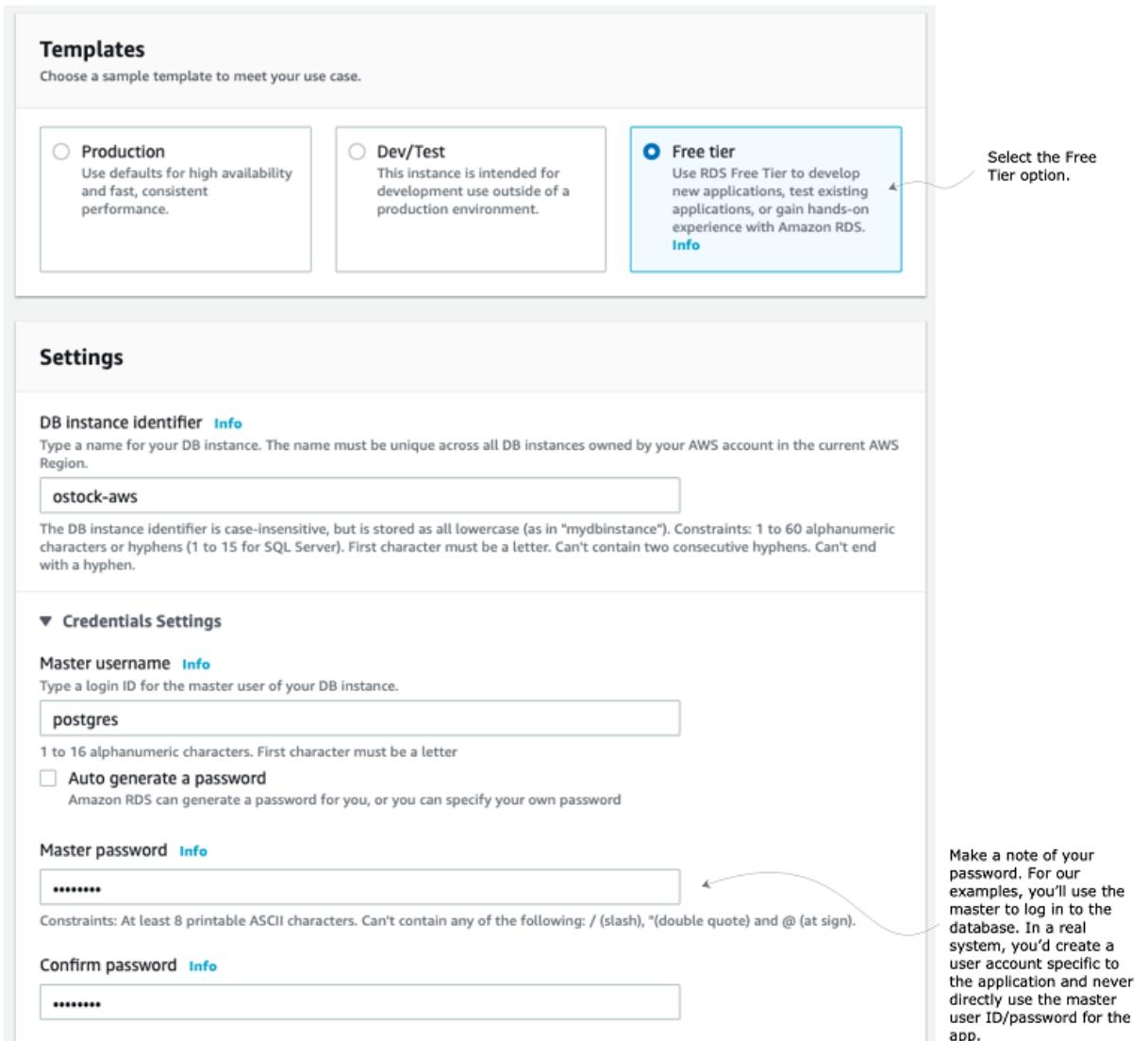
---

## 12.2.1 Creating the PostgreSQL database using Amazon RDS

Before we begin this section, we need to set up and configure our Amazon AWS account. Once this is done, our first task is to create the PostgreSQL database that we're going to use for our Ostock services. To do this, we're going to log in into the Amazon AWS console (<https://aws.amazon.com/console/>) and do the following:

1. You'll be presented with a list of Amazon web services when you first log into the console. Locate the link called RDS. Click on the link, and this will take you to the RDS dashboard.
2. On the dashboard, you'll find a big button that says, "Create Database." Click on it.
3. Amazon RDS supports different database engines. You should see a list of databases. Select PostgreSQL and click the "Option." This will display the database fields for the specific creation.

The first thing the Amazon database creation process will ask is whether this is a production database, a dev/test database, or a free tier. We're going to create select the free tier option. Next, we're going to set up necessary information about our PostgreSQL database and set the master user ID and password we're going to use to log into the database. Figure 12.3 shows this screen.



**Figure 12.3 Selecting whether the database is going to be a production, test or free tier database and setting up the basic database configuration.**

Next, we are going to leave the default configuration for the following sections:

- DB instance size
- Storage
- Availability & durability

- Database Authentication

The last and final step of the wizard is to set up the following information:

- **The Virtual private cloud (VPC):** Default VPC
- **Publicly accessible:** true
- **The database security groups:** Create a new security group and assigned a specific name for it. For purposes of this example I named the security group ostock-sg.
- **Port:** 5432 TCP/IP port that the database will use.

Figure 12.4 shows the contents of this screen.

**Virtual private cloud (VPC) [Info](#)**  
VPC that defines the virtual networking environment for this DB instance.

▼

Only VPCs with a corresponding DB subnet group are listed.

i After a database is created, you can't change the VPC selection.

**▼ Additional connectivity configuration**

**Subnet group [Info](#)**  
DB subnet group that defines which subnets and IP ranges the DB instance can use in the VPC you selected.

▼

**Publicly accessible [Info](#)**

Yes  
Amazon EC2 instances and devices outside the VPC can connect to your database. Choose one or more VPC security groups that specify which EC2 instances and devices inside the VPC can connect to the database.

No  
RDS will not assign a public IP address to the database. Only Amazon EC2 instances and devices inside the VPC can connect to your database.

**VPC security group**  
Choose one or more RDS security groups to allow access to your database. Ensure that the security group rules allow incoming traffic from EC2 instances and devices outside your VPC. (Security groups are required for publicly accessible databases.)

Choose existing  
Choose existing VPC security groups

Create new  
Create new VPC security group

New VPC security group name

Availability Zone [Info](#)

▼

**Database port [Info](#)**  
TCP/IP port that the database will use for application connections.

←

Note port number.  
The port number will be used as part of your service's connect string.

For now, we'll create a new security group and allow the database to be publicly accessible.

**Figure 12.4 Setting up the security group, port and backup options for the RDS database.**

At this point, our database creation process will begin (it can take several minutes). Once it's done, we'll need to configure the Ostock services to use the database. After the database is created (this will take several minutes), we'll navigate

back to the RDS dashboard and see our database created. Figure 12.5 shows this screen.

The screenshot shows the Amazon RDS Databases dashboard. At the top, there is a search bar labeled "Filter databases" and a "Create database" button. Below the search bar is a table with columns: DB Identifier, Role, Engine, Region & AZ, Size, Status, and CPU. A single row is selected, showing "ostock-aws" as the DB Identifier, "Instance" as the Role, "PostgreSQL" as the Engine, "us-east-1f" as the Region & AZ, "db.t2.micro" as the Size, "Available" as the Status, and "3.83%" as the CPU usage. To the left of the table, a callout bubble points to the "ostock-aws" entry with the text "Click database name to see database properties." Below the table, the URL "RDS > Databases > ostock-aws" is shown, followed by the database name "ostock-aws". The main content area is titled "Summary" and contains sections for DB Identifier, Role, CPU usage, Info, Class, and Region & AZ. Below the summary is a navigation bar with tabs: Connectivity & security (which is active), Monitoring, Logs & events, Configuration, Maintenance & backups, and Tags. The "Connectivity & security" tab displays information under three headings: Endpoint & port, Networking, and Security. A callout bubble points to the "Endpoint" section with the text "This is the endpoint and port you'll use to connect to the database." The endpoint listed is "ostock-aws.cd8qbcnzh.rds.amazonaws.com" on port 5432.

**Figure 12.5 The created Amazon RDS/PostgreSQL database.**

For this chapter, I created a new application profile called `aws-dev` for each microservice that needs to access the Amazon-based PostgreSQL database. I added a new Spring Cloud Config server application profile in the Spring Cloud Config GitHub repository (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter12/configserver/src/main/resources/config>) containing the Amazon database-connection

information. Remember, I'm using the classpath repository, but you can use an external Spring Cloud Repository, such as Github. The property files follow the naming convention (service-name)-aws-dev.yml or .properties in each of the property files (licensing service, and organization service) I'm using the new database.

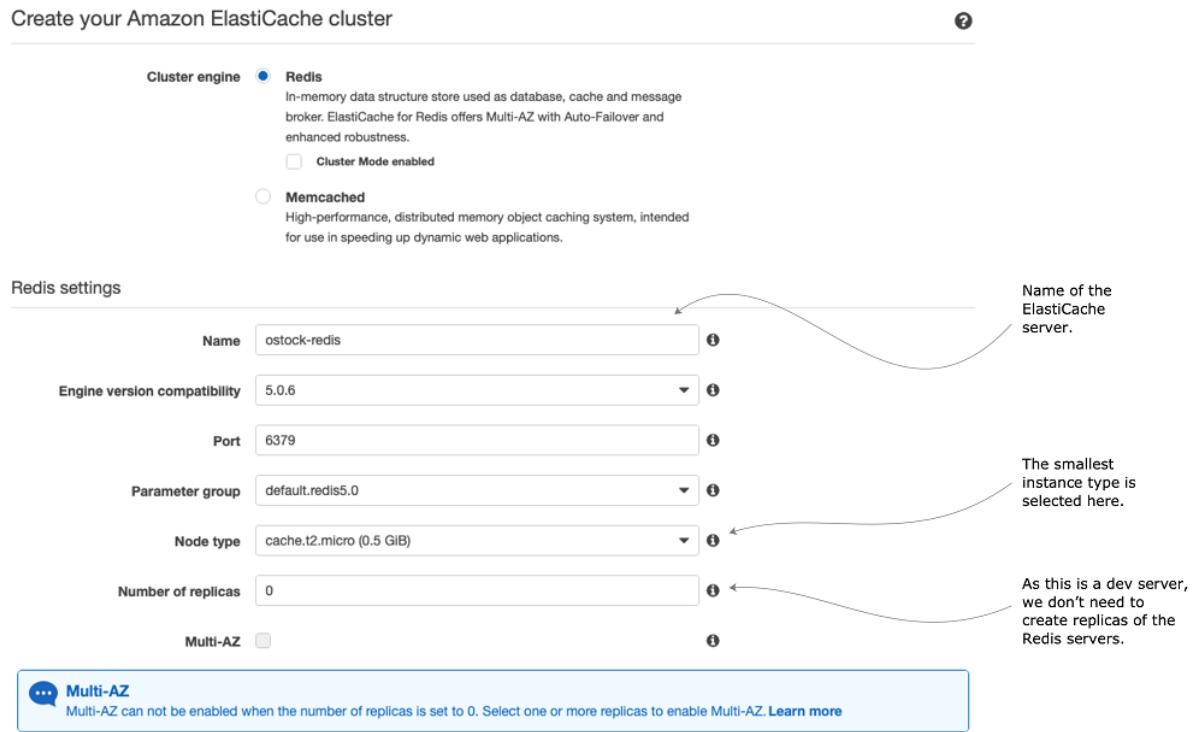
At this point, our database is ready to go (not bad for setting it up in approximately five clicks). Let's move to the next piece of application infrastructure and see how to create the Redis cluster that our Ostock licensing service is going to use.

## **12.2.2 Creating the Redis cluster in Amazon**

To set up the Redis cluster, we're going to use the Amazon ElastiCache service. Amazon ElastiCache allows us to build in-memory data caches using Redis or Memcached (<https://memcached.org/>). For the Ostock services, we're going to move the Redis server we were running in Docker to ElastiCache.

To begin, navigate back to the AWS Console's main page, search for the ElastiCache service and click the ElastiCache link.

From the ElastiCache console, select the Redis link (left-hand side of the screen), and then hit the blue Create button at the top of the screen. This will bring up the ElastiCache/Redis creation wizard. Figure 12.6 shows the Redis creation screen.



## Figure 12.6 With a few clicks we can set up a Redis cluster whose infrastructure is managed by Amazon.

As for the Advance Redis settings, let's select the same security group we created for the PostgreSQL database, and let's uncheck the “Enable Automatic Backups” option. Once filled with all the data, go ahead, and hit the create button. Amazon will begin the Redis cluster creation process (this will take several minutes). Amazon will build a single-node Redis server running on the smallest Amazon server instance available. Once we hit the button, we'll see our Redis cluster being created. Once the cluster is created, we can click on the name of the cluster, and it will take us to a detailed screen showing the endpoint used in the cluster. Figure 12.7 shows the details of the Redis clustered after it has been created.

Cluster Name	Mode	Shards	Nodes	Node Type	Status	Update Action Status	Encryption in-transit	Encryption at-rest	Global
ostock-redis	Redis	0	1 node	cache.t2.micro	available	up to date	No	No	-
Name:	ostock-redis						Global Datastore:	-	
Global Datastore Role:	-						Creation Time:	June 14, 2020 at 6:28:44 PM UTC-6	
Configuration Endpoint:	-						Status:	available	
Primary Endpoint:	ostock-redis.ymengq.0001.use1.cache.amazonaws.com:6379						Update Status:	up to date	
Engine:	Redis						Reader Endpoint:	-	
Engine Version Compatibility:	5.0.6						Node type:	cache.t2.micro	
Availability Zones:	us-east-1c						Shards:	0	
Number of Nodes:	1 node						Multi-AZ:	Disabled	

## Figure 12.7 The Redis endpoint is the key piece of information your services need to connect to Redis.

The licensing service is the only one of our services to use Redis, so make sure that if you deploy the code examples in this chapter to your own Amazon instance, you modify the licensing service's Spring Cloud Config files appropriately.

## 12.3 Beyond the infrastructure: deploying OStock and ELK

At this point, you have part of the infrastructure set up and can now move into the second half of the chapter. In this second part, you're going to deploy the Elasticsearch, Logstash and Kibana (ELK) stack services in an EC2 instance and the Ostock services to an Amazon EKS container. Remember, an EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) where applications can be run. The Ostock deploy process is going to be divided into two

parts as well. The first part of your work is for the terminally impatient (like me) and will show how to deploy Ostock manually to your Amazon EKS Cluster. This will help you understand the mechanics of deploying the service and see the deployed services running in your container. While getting your hands dirty and manually deploying your services is fun, but it isn't sustainable or recommended.

The second part is where you're going to automate the entire build and deployment process and take the human being out of the picture. This is your targeted end state and really caps the work you've been doing in the book by demonstrating how to design, build, and deploy microservices to the cloud.

### **12.3.1 Creating an EC2 with the ELK Services**

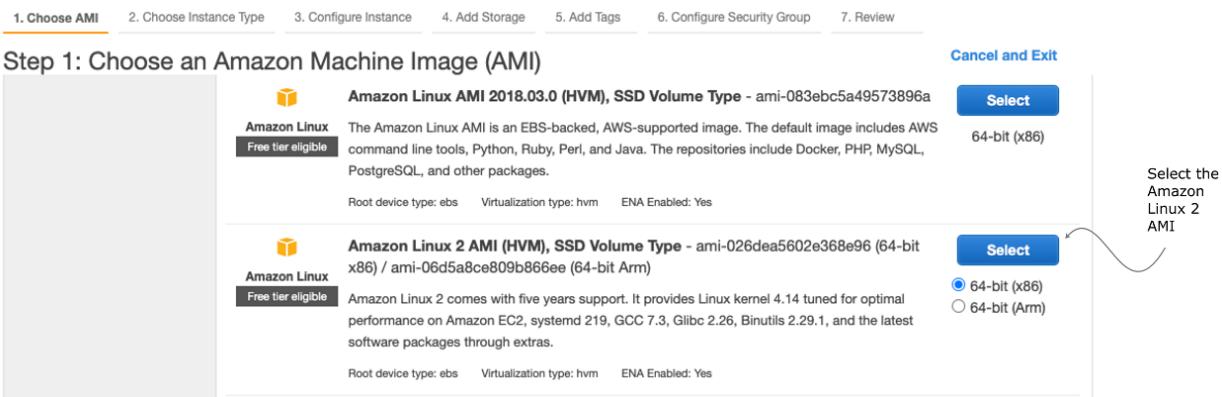
To setup the ELK, we are going to use an EC2 instance. The idea of having this service separated from the ones we are going to be deploying in the ELK is to show that we can have different instances and still use the services.

To create the EC2 instances we must do the following steps.

1. Choose an Amazon machine Image (AMI).
2. Select an instance type.
3. Set the default configuration for the Configure Instance Details, Storage and tags.
4. Create a new security group.
5. Launch the EC2 instance.

To begin, navigate back to the AWS Console's main page, search for the EC2 service and click the link.

From the EC2 console, hit the blue Launch instance button. This will bring up the EC2 creation wizard. This wizard contains seven different steps and I will guide you through each of them. Figure 12.8 shows the first step in the EC2 creation process.



**Figure 12.8 Select the Amazon Linux 2 AMI for the ELK Service instance.**

The second step is to choose the instance type, for purposes of this example, we can select an m4.large server because of its large amount of memory (8 GB) and low hourly cost (.010 cents per hour). Figure 12.9 shows this second step.

1. Choose AMI	2. Choose Instance Type	3. Configure Instance	4. Add Storage	5. Add Tags	6. Configure Security Group	7. Review
<b>Step 2: Choose an Instance Type</b>						
<input type="checkbox"/>	General purpose	m5.16xlarge	64	256	EBS only	Yes
<input type="checkbox"/>	General purpose	m5.24xlarge	96	384	EBS only	Yes
<input type="checkbox"/>	General purpose	m5.metal	96	384	EBS only	Yes
<input checked="" type="checkbox"/>	General purpose	m4.large	2	8	EBS only	Yes
<input type="checkbox"/>	General purpose	m4.xlarge	4	16	EBS only	Yes

You can choose a m4.large server because of its large amount of memory (8 GB) and low hourly cost (.010 cents per hour).

## Figure 12.9 Selecting the instance type for the ELK Service instance.

Once selected the instance, click the “Next: Configure Instance Details”. In the third step, fourth and fifth step we are not to change anything, so just click next and leave all the default configuration.

In the sixth step, we have to select to create a new security group or select an existing Amazon security group that you’ve created to apply to the new EC2 instance.

For purposes of this example, we will allow all inbound traffic from the world (0.0.0.0/0 is the network mask for the entire internet). In real scenarios, you need to be careful with this, so I recommend you take some time to analyze the inbound rules that bests fit your needs. The following figure 12.10, shows the specific data for this step.

**Step 6: Configure Security Group**

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group:  Create a new security group  Select an existing security group

**Security group name:** elk-service-sg  
**Description:** ELK Service Security Group

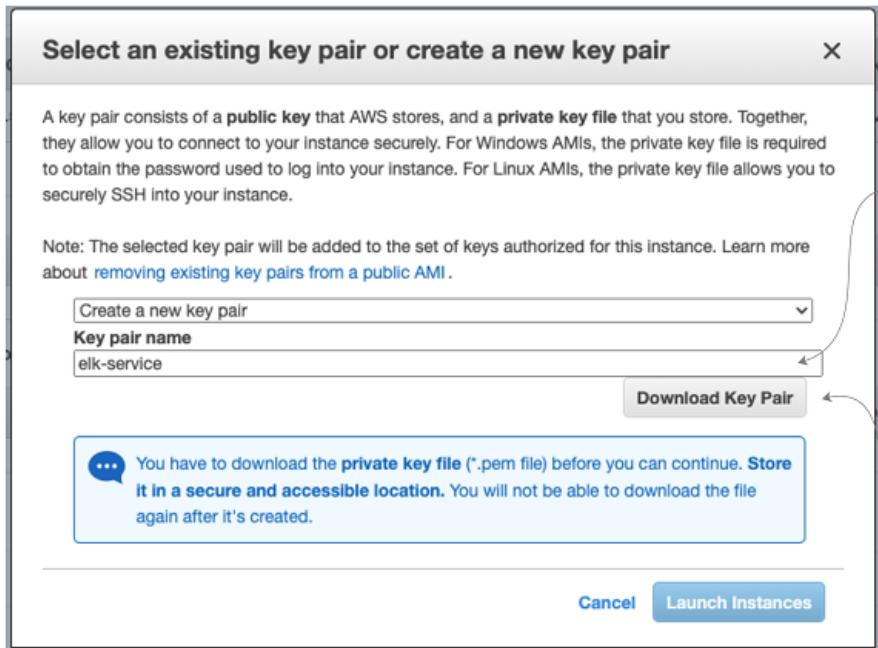
Type	Protocol	Port Range	Source	Description
SSH	TCP	22	Custom 0.0.0.0/0	e.g. SSH for Admin Desktop
All traffic	All	0 - 65535	Anywhere 0.0.0.0/0, ::/0	e.g. SSH for Admin Desktop

**Add Rule**

We're going to create a new security group with one inbound rule that will allow all traffic.

## Figure 12.10 Creating a security group for the EC2 instance.

Once the security group is set, click the Review and Launch button to see the review page with all the EC2 instance configuration. Once verified that all the configuration is correct, you should click Launch and continue to execute the last step. This final step involves the creation of a key pair to connect to the EC2 instance. The following figure 12.11 shows the key pair creation process.



Key pair file name.

Make sure you download and store the private key in a safe place before continuing.

## Figure 12.11 Creating a security group for the EC2 instance.

At this point, we have all the infrastructure we need to run the ELK stack. Figure 12.12 shows the EC2 instance create.

The screenshot shows the AWS EC2 dashboard. At the top, there are buttons for 'Launch Instance', 'Connect', and 'Actions'. Below that is a search bar and a filter section with dropdowns for 'Name', 'Instance ID', 'Instance Type', 'Availability Zone', 'Instance State', 'Status Checks', and 'All'. A single instance is listed: 'i-0fd822819c7253fe1' (m4.large, us-east-2c, running, 2/2 checks). A callout bubble says 'Make sure your instance is running.' On the instance details page, tabs for 'Description', 'Status Checks', 'Monitoring', and 'Tags' are shown. Under 'Description', fields include 'Instance ID: i-0fd822819c7253fe1', 'Public DNS (IPv4): ec2-3-136-161-26.us-east-2.compute.amazonaws.com', 'Instance state: running', and 'Instance type: m4.large'. A callout bubble says 'Take note of the IPv4 Public IP.' under the Public DNS field.

**Figure 12.12 EC2 dashboard page containing the created EC2 instance.**

Before moving to the next step, take note on the IPV4 because we are going to need it to connect to our EC2 instance.

**NOTE** To continue with this part I highly recommend you download the AWS folder from the chapter 12 repository. (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter12/AWS>). In this repository, I already created the service scripts .yaml and docker-compose files that we are going to be using for the deploys in the EC2 and EKS containers.

To continue, open a terminal console on the path where you downloaded the key pair file and execute the following commands.

```
chmod 400 <pemKey>.pem
scp -r -i <pemKey>.pem ~/project/folder/from/root ec2-user@<IPv4>:~/
```

It's important to highlight that we are going to use as project folder, a folder that contains all of the configuration files we need to deploy our services. If you downloaded the AWS folder that would be your project folder. Once executed the previous you should see the console output shown in figure 12.13.

```
Last login: Sat Jun 20 06:35:47 2020 from 186.176.131.131

 _\ __))
 _\ (/ Amazon Linux AMI
 ___\|_|_|

https://aws.amazon.com/amazon-linux-ami/2018.03-release-notes/
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ec2-user@ip-172-31-42-196 ~]$ ls -l
total 4
drwxr-xr-x 3 ec2-user ec2-user 4096 Jun 18 16:55 AWS
```

### **Figure 12.13 Log into the EC2 instance using ssh and the key pair we previously created.**

If we list the files and directories of the root of our EC2 instance, we will see that the entire project folder was pushed into the instance.

Now that we have everything set up, let's continue with the ELK deployment.

### **12.3.2 Deploying the ELK stack in the EC2 instance**

Now, that we have created the instance and we logged in, let's continue with the deployment of the ELK services. To achieve this, we must execute the following steps:

1. Update the EC2 instance
2. Install docker
3. Install docker-compose
4. Run Docker
5. Run the docker-compose file that contains the ELK services.

To execute all of the previous steps, just run the following commands shown in listing 12.1.

### **Listing 12.1 ELK deployment in an EC2 instance**

```
sudo yum update #A
sudo yum install docker #B
sudo curl -L https://github.com/docker/compose/releases/download/1.21.0/docker-
compose-`uname -s`-`uname -m` | sudo tee /usr/local/bin/docker-compose > /dev/null
#C
sudo chmod +x /usr/local/bin/docker-compose
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
sudo service docker start #D
sudo docker-compose -f AWS/EC2/docker-compose.yml up -d #E
```

#A Command that updates the applications installed on a system, in this particular scenario, the EC2 instance.  
#B Command to install Docker  
#C Command to install docker-compose  
#D Run docker in the EC2 instance  
#E Run the docker-compose file located in the AWS/EC2 folder. The -d option is to run the process in the background.

At this point, we have all the ELK services running. To verify that everything is up, you can hit the following URL <http://<IPv4>:5601/>, and you will see the Kibana dashboard.

Now, that we have the ELK services running, let's create an Elastic Kubernetes Service Cluster to deploy our microservices.

## 12.3.3 Creating an EKS Cluster

The last and final step before deploying the Ostock services is to set up an Amazon EKS cluster.

An Elastic Kubernetes Service (EKS), is a service offered by Amazon that allows us to run Kubernetes on AWS.

Kubernetes is a container orchestrator that handles the deployment, scheduling, creation, deletion of containers. As previously discussed, managing a large number of containers and microservices can become a challenging task, with tools such as Kubernetes, we can manage to handle all of our containers in a faster and efficient way. For example, let's imagine the following scenario: We need to update a microservice and, therefore, its image, which is propagated in dozens or maybe hundreds of containers. Do you imagine yourself deleting manually and recreating those containers? With Kubernetes, you can destroy and recreate them just by executing a short command. This is only one of the numerous advantages that Kubernetes brings to the table.

Now that we have a brief overview of what Kubernetes is and does let's continue. The process of creating the EKS cluster can be done by using the AWS Web console and the AWS CLI. For purposes of this example, I will show you how to do it using the CLI.

**NOTE** To continue with this part, make sure you have all the tools previously mentioned in the sidebar “Some prerequisites for working” and that you have configured the AWS CLI. In case you haven't, I highly recommend you look at the following documentation.

<https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html>

After we've configured and installed all the necessary tools, we can start creating and configuring our Kubernetes cluster. To do so, we need to

1. Provisioning a Kubernetes cluster
2. Push the microservices images into repositories.
3. Deploy our microservices manually in the Kubernetes Cluster.

## ***PROVISIONING A KUBERNETES CLUSTER***

To create our cluster using the command-line interface, we need to use the eksctl tool. This is accomplished by using the following straightforward command.

```
eksctl create cluster --name=ostock-dev-cluster --nodes=1 --node-type=m4.large
```

The previous eksctl command line allows us to create a Kubernetes cluster named ostock-dev-cluster, which uses a single m4.large EC2 instance as a worker node. In this scenario, I'm using the same instance type we previously selected in the EC2 ELK service creation. Remember that this instance costs 0.10 cents per hour. If you want to know more about the EC2 instances pricing, visit the official Amazon documentation at the following link.

<https://aws.amazon.com/ec2/pricing/on-demand/>

The execution of the command takes several minutes. While processing, you'll see different line outputs, but you will

know once it is over because you should see the following output.

```
[✓] EKS cluster "ostock-dev-cluster" in "region-code" region is ready
```

Once executed, you can execute the `kubectl get svc` command to verify that `kubectl` configuration is correct. This command will show the following output.

```
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.100.0.1 <none> 443/TCP 1m
```

## ***PUSHING THE MICROSERVICES IMAGES INTO REPOSITORIES***

At this point, we have always built our services into our local machine. To use these images in our EKS cluster, we need to push the Docker container images to a container repository. A container repository is like a Maven repository for your created Docker images. Docker images can be tagged and uploaded to it, and other projects can download and use the images.

There are several repositories such as the Docker Hub, but for this example, let's continue with the AWS infrastructure, and use the Amazon Elastic Container Registry (ECR) (<https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html>).

The first step, to push the images to the container registry is making sure that we have the docker images in our local

docker. In case you don't just go ahead and execute the following command on the parent pom.xml file.

```
mvn clean package dockerfile:build
```

Once executed the command we should now see the images in the docker images list. To verify this, execute the docker images command and you should see the following output.

```
REPOSITORY TAG IMAGE ID SIZE
ostock/organization-service chapter12 b1c7b262926e 485MB
ostock/gatewayserver chapter12 61c6fc020dcf 450MB
ostock/configserver chapter12 877c9d855d91 432MB
ostock/licensing-service chapter12 6a76bee3e40c 490MB
ostock/authentication-service chapter12 5e5e74f29c2 452MB
ostock/eurekaserver chapter12 e6bc59ae1d87 451MB
```

The next step is to authenticate our Docker client with our ECR registry. To achieve this first we need to obtain a password and our AWS account id by executing the following the commands.

```
aws ecr get-login-password
aws sts get-caller-identity --output text --query "Account"
```

The first command will retrieve the password and the second command our AWS account. Both values are going to be used to authenticate our Docker client.

Now, that we have our credentials, let's authenticate by executing the following command.

```
docker login -u AWS -p [password] https://[aws_account_id].dkr.ecr.[region].amazonaws.com
```

Once authenticated, the next step is creating the repositories where we are going to store our images. To create these repositories let's execute the following commands.

```
aws ecr create-repository --repository-name ostock/configserver
aws ecr create-repository --repository-name ostock/gatewayserver
aws ecr create-repository --repository-name ostock/eurekaserver
aws ecr create-repository --repository-name ostock/authentication-service
aws ecr create-repository --repository-name ostock/licensing-service
aws ecr create-repository --repository-name ostock/organization-service
```

When each command is executed you should see an output similar to the following:

```
{
 "repository": {
 "repositoryArn": "arn:aws:ecr:us-east-2:
8193XXXXXX43:repository/ostock/configserver",
 "registryId": "8193XXXXXX43",
 "repositoryName": "ostock/configserver",
 "repositoryUri": "8193XXXXXX43.dkr.ecr.us-east-
2.amazonaws.com/ostock/configserver",
 "createdAt": "2020-06-18T11:53:06-06:00",
 "imageTagMutability": "MUTABLE",
 "imageScanningConfiguration": {
 "scanOnPush": false
 }
 }
}
```

Make sure to take note of all the repository URI's because we are going to need it to create the tag and to push the images.

The next step is to create the tags for our images, to achieve this let's execute the following commands.

```
docker tag ostock/configserver:chapter12 [configserver-repository-uri]:chapter12
docker tag ostock/gatewayserver:chapter12 [gatewayserver-repository-uri]:chapter12
```

```

docker tag ostock/eurekaserver:chapter12 [eurekaserver-repository-uri]:chapter12
docker tag ostock/authentication-service:chapter12 [authentication-service-
repository-uri]:chapter12
docker tag ostock/licensing-service:chapter12 [licensing-service-repository-
uri]:chapter12
docker tag ostock/organization-service:chapter12 [organization-service-repository-
uri]:chapter12

```

This command is going to create a new tag for all the images, at this point if you execute the docker images command you should see a similar output to the one shown in figure 12.14.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
81932222443.dkr.ecr.us-east-2.amazonaws.com/ostock/organization-service	chapter12	b1c7b262926e	45 hours ago	485MB
ostock/organization-service	chapter12	b1c7b262926e	45 hours ago	485MB
81932222443.dkr.ecr.us-east-2.amazonaws.com/ostock/gatewayserver	chapter12	61c6fc020dcf	45 hours ago	450MB
ostock/gatewayserver	chapter12	61c6fc020dcf	45 hours ago	450MB
81932222443.dkr.ecr.us-east-2.amazonaws.com/ostock/configserver	chapter12	877c9d855d91	45 hours ago	432MB
ostock/configserver	chapter12	877c9d855d91	45 hours ago	432MB
81932222443.dkr.ecr.us-east-2.amazonaws.com/ostock/licensing-service	chapter12	6a76bee3e40c	46 hours ago	490MB
ostock/licensing-service	chapter12	6a76bee3e40c	46 hours ago	490MB
81932222443.dkr.ecr.us-east-2.amazonaws.com/ostock/authentication-service	chapter12	f5e5e74f29c2	5 days ago	452MB
ostock/authentication-service	chapter12	f5e5e74f29c2	5 days ago	452MB
81932222443.dkr.ecr.us-east-2.amazonaws.com/ostock/eurekaserver	chapter12	e6bc59ae1d87	5 days ago	451MB
ostock/eurekaserver	chapter12	e6bc59ae1d87	5 days ago	451MB

## Figure 12.14 Docker images with the Amazon Elastic Container Registry Repository URI tag.

Finally, we can push our microservices images to the ECR registry repositories. To achieve this just execute the following commands.

```

docker push [configserver-repository-uri]:chapter12
docker push [gatewayserver-repository-uri]:chapter12
docker push [eurekaserver-repository-uri]:chapter12
docker push [authentication-service-repository-uri]:chapter12
docker push [licensing-service-repository-uri]:chapter12
docker push [organization-service-repository-uri]:chapter12

```

Once executed all the commands, you can visit the ECR service in the AWS Web Console and you should see a similar

list to the one shown in figure 12.15.

The screenshot shows the AWS ECR (Elastic Container Registry) interface. At the top, there's a navigation bar with 'ECR' and 'Repositories'. Below it is a search bar with 'Find repositories'. A prominent orange button on the right says 'Create repository'. The main area is a table titled 'Repositories (6)' with columns: 'Repository name', 'URI', 'Created at', 'Tag immutability', and 'Scan on push'. Each row lists a repository name in blue, its corresponding URI starting with '819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/' followed by the service name, the creation date and time, and two status indicators: 'Disabled' for both tag immutability and scan on push.

Repository name	URI	Created at	Tag immutability	Scan on push
ostock/authentication-service	819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/authentication-service	06/18/20, 11:54:29 AM	Disabled	Disabled
ostock/configserver	819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/configserver	06/18/20, 11:53:06 AM	Disabled	Disabled
ostock/eurekaserver	819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/eurekaserver	06/18/20, 11:54:13 AM	Disabled	Disabled
ostock/gatewayserver	819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/gatewayserver	06/18/20, 11:53:54 AM	Disabled	Disabled
ostock/licensing-service	819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/licensing-service	06/18/20, 11:54:44 AM	Disabled	Disabled
ostock/organization-service	819322222443.dkr.ecr.us-east-2.amazonaws.com/ostock/organization-service	06/18/20, 11:55:06 AM	Disabled	Disabled

**Figure 12.15 Elastic Container Registry repositories.**

## ***DEPLOYING OUR MICROSERVICES IN THE EKS CLUSTER***

To deploy our microservices first you need to make sure that the licensing, organization and gateway services have the Spring Cloud Config files appropriately. Remember, the PostgreSQL, Redis and the ELK stack services changed.

**NOTE** On the following (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter12>) GitHub repository, you can find all the changes I did, the config server is now pointing to a GitHub repository, and it also contains the configuration for the Logstash server. If you'd remember, when we created the Logstash configuration, we had a hard-coded value.

Before deploying our microservices, let's create the services for our Kafka and zookeeper services. There are several ways to create these services, for purposes of this scenario I chose to create them using some existent helm charts.

In case you are not familiar with Helm, Helm is a package manager for Kubernetes clusters and a Helm chart is a Helm package. A Helm package contains all the required resource definitions necessary to run a specific service, application or tool inside a Kubernetes cluster.

**NOTE** In case you don't have helm installed, visit the following link (<https://helm.sh/docs/intro/install/>) to see all the possible ways to install it on your computer.

Once installed, let's execute the following commands to create our Kafka and Zookeeper services.

```
helm install zookeeper bitnami/zookeeper \
--set replicaCount=1 \
--set auth.enabled=false \
--set allowAnonymousLogin=true
helm install kafka bitnami/kafka \
--set zookeeper.enabled=false \
--set replicaCount=1 \
--set externalZookeeper.servers=zookeeper
```

Once executed you should see an output with some of the details of the services. To be sure that everything ran successfully just execute the kubectl get pods command to see your services running.

```
NAME READY STATUS RESTARTS AGE
kafka-0 1/1 Running 0 77s
zookeeper-0 1/1 Running 0 101s
```

The next step to deploy our services is to convert our docker-compose file into a compatible Kubernetes format. Why? Kubernetes doesn't support the compose files, so in order to execute these files, we need to use a tool called Kompose.

Kompose is a conversion tool for the Docker compose to container orchestrations such as Kubernetes. Using this tool allows us to convert all the docker-compose.yaml file configurations with a straightforward command kompose convert <file>, run the docker-compose with the kompose up command, and more. If you want to know more about Kompose I highly recommend look the following documentation (<https://kompose.io/>).

For this example, we are going to use the convert files from Kompose. You can find these files in the AWS/EKS folder in the following GitHub repository (<https://github.com/ihuaylupo/manning-smia/tree/master/chapter12/AWS/EKS>). Before moving on, let's make a pause and review the service types in Kubernetes, these services types allow us to expose the services onto external IP addresses.

Kubernetes have the following four different types of services:

- **ClusterIP:** This service exposes the service on a cluster-internal IP. If we choose this option our service is only going to be visible within our cluster.
- **NodePort:** This service exposes the service at a static port (the NodePort value). The Kubernetes allocates a default port range from 3000 to 32767. You can change this by using the --service-node-port-range flag in the

spec.containers.commands section of the service.yaml file.

- **LoadBalancer:** This service exposes the service externally using a cloud load balancer.
- **ExternalName:** This service maps the service to the contents of the an external name.

If you take a look at the files, you'll see that some of the <service>.yaml files have a type=NodePort and a nodePort attribute. In Kubernetes, if we don't define a service type Kubernetes, uses the default ClusterIP service type.

Now, that we know more about the Kubernetes service types, let's continue. To create the services using those converted files, just execute the following commands on the root of the AWS/ELK folder.

```
kubectl apply -f <service>.yaml,<deployment>.yaml
```

I prefer to execute one by one to see that everything is being created successfully but you can also create all the services at the same time, to do this you only need to concatenate all the yaml files in the -f argument. For example, to create the config server, see the pod status and the logs we need to execute the following commands.

```
kubectl apply -f configserver-service.yaml,configserver-deployment.yaml
kubectl get pods
kubectl logs <POD_NAME> --follow
```

To test that our service is up and running, we need to add a few rules to our security group to allow all the incoming

traffic from the NodePorts. To do so, let's execute the following command to retrieve the security group ID.

```
aws ec2 describe-security-groups --filters Name=group-name,Values="*eksctl-ostock-dev-cluster-nodegroup*" --query "SecurityGroups[*].{Name:GroupName,ID:GroupId}"
```

With the security group ID, let's execute the following command to create the inbound rule. You can also do this in the Web console, just by going to the security group and create a new inbound traffic rule.

```
aws ec2 authorize-security-group-ingress --protocol tcp --port 31000 --group-id [security-group-id] --cidr 0.0.0.0/0
```

To get the external IP you can execute the following command kubectl get nodes -o wide. The command will show you a similar output to the one shown in figure 12.16.

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
ip-192-168-74-143.us-east-2.compute.internal	Ready	<none>	125m	v1.16.8-eks-e16311	192.168.74.143	3.15.208.238	Amazon Linux 2	4.14.181-140.257.amzn2.x86_64	docker://19.3.6

## Figure 12.16 Node external IP Address.

Now, you can open the following URL in your browser.

```
http:<node-external-ip>:<NodePort>/actuator
```

In case you need to delete a pod, service or deployment you can execute the kubectl delete -f <service>.yaml,

<deployment>.yaml or the kubectl delete <POD\_NAME> command.

We are almost done, if you looked at the yaml files the postgres.yaml file is a bit different. In this file what we are going to specify is that the database service is going to be used with an external address. In order to make the connection work you should specify the endpoint for the RDS postgres service. The following code listing 12.2 shows you how.

## **Listing 12.2 Adding the external reference to the database service**

```
apiVersion: v1
kind: Service
metadata:
 labels:
 app: postgres-service
 name: postgres-service
 spec:
 externalName: ostock-aws.cjuqpwnyahhy.us-east-2.rds.amazonaws.com #A
 selector:
 app: postgres-service
 type: ExternalName
 status:
 loadBalancer: {}
```

#A Your RDS PostgreSQL endpoint.

Make sure you add in the externalName the endpoint of your RDS PostgreSQL instance. Once, done the change you can execute the following command.

```
kubectl apply -f postgres.yaml
```

Once the services are created and running, you can now continue running all of the other services. While developing this chapter, I saw that we have to create a VPC peering and update the route tables of both the RDS and the EKS cluster to allow the communication between them. I will not be describing how to do this, but I highly recommend reading the following article that contains all of the steps:

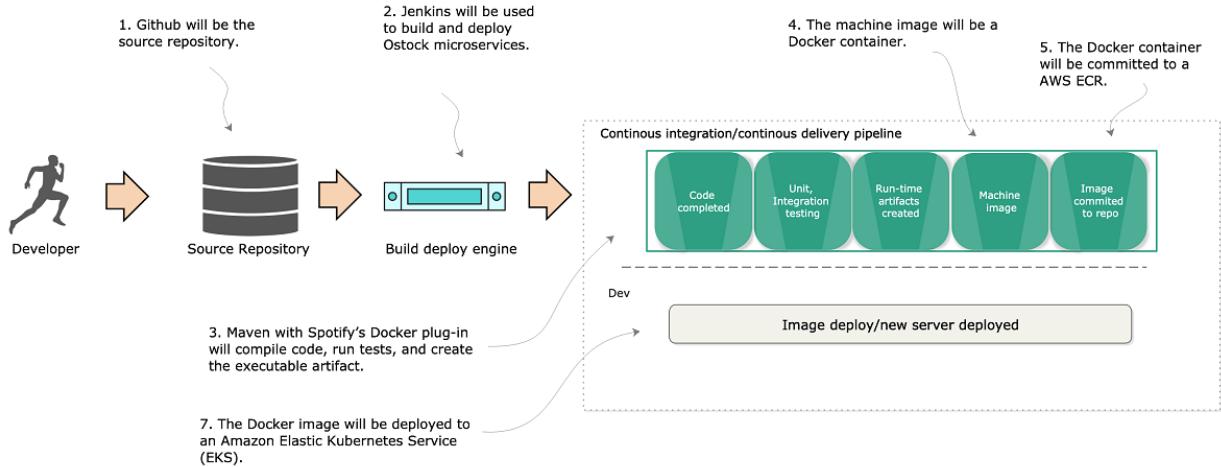
<https://dev.to/bensooraj/accessing-amazon-rds-from-aws-eks-2pc3> and/or the official AWS Documentation to understand a little bit more VPC peering

<https://docs.aws.amazon.com/vpc/latest/peering/create-vpc-peering-connection.html>.

At this point, you've successfully deployed your first set of services to an Amazon EKS cluster. Now, let's build on this by looking at how to design a build and deployment pipeline that can automate the process of compiling, packaging, and deploying your services to Amazon.

## **12.4 Your build and deployment pipeline in action**

From the general architecture laid out in section 12.1, you can see that there are many moving pieces behind a build/deployment pipeline. Because the purpose of this book is to show you things “in action,” we’re going to walk through the specifics of implementing a build/deployment pipeline for the Ostock services. Figure 12.17 lays out the different technologies we’re going to use to implement our pipeline.



**Figure 12.17 Technologies used in the Ostock build.**

1. **GitHub** (<http://github.com>). GitHub is our source control repository. All the application code for this book is in GitHub. I chose GitHub as the source control repository because GitHub offers a wide variety of webhooks and robust REST-based APIs for integrating GitHub into your build process.
2. **Jenkins** (<https://www.jenkins.io/>). Jenkins is the continuous integration engine I used for building and deploying the Ostock microservices and provisioning the Docker image that will be deployed. Jenkins can be easily configured via the web interface and includes several built-in plugins that are going to make our work easier.
3. **Maven/Spotify Docker Plugin** (<https://github.com/spotify/dockerfile-maven>) or **Spring Boot Build Image Target** (<https://spring.io/guides/gs/spring-boot-docker/>). While we use vanilla Maven to compile, test, and package

Java code, essential Maven plug-ins can allow us to kick off the creation of a Docker build right from within Maven.

4. **Docker** (<https://www.docker.com/>). I chose Docker as our container platform for two reasons. First, Docker is portable across multiple cloud providers. I can take the same Docker container and deploy it to AWS, Azure, or Cloud Foundry with a minimal amount of work. Second, Docker is lightweight. By the end of this book, you've built and deployed approximately 10 Docker containers (including a database server, messaging platform, and a search engine). Deploying the same number of virtual machines on a local desktop would be difficult due to the sheer size and speed of each image.
5. **Amazon Elastic Container Registry** (<https://aws.amazon.com/ecr/>). After a service has been built and a Docker image has been created, it's tagged with a unique identifier and pushed to a central repository. For the Docker image repository, I chose to use AWS ECR.
6. **Amazon's EKS Container Service (EKS)**. The final destination for our microservices will be Docker instances deployed to Amazon's Docker platform. I chose Amazon as the cloud platform because it's by far the most mature of the cloud providers and makes it trivial to deploy Docker services.

## 12.5 Beginning our build deploy/pipeline: Github

## and Jenkins

Dozens of source control engines and build deploy engines (both on-premise and cloud-based) can implement your build and deploy pipeline. For the examples in this book, I purposely chose GitHub as the source control repository and Jenkins as the build engine. The Git source control repository is extremely popular, and GitHub is one of the largest cloud-based source control repositories available today.

Jenkins is a build engine that integrates tightly with GitHub. It's straightforward to use. Its simplicity and opinionated nature make it easy to get a simple build pipeline off the ground.

Up to now, all of the code examples in this book could be run solely from your desktop (with the exception of connectivity out to GitHub). For this chapter, if you want to completely follow the code examples, you'll need to set up your own GitHub, Jenkins, and AWS accounts. We're not going to walk through on how to set up the Jenkins, however if you are not familiar with the Jenkins you can take a look at the step by step file that I made that walks you through on how to set up the Jenkins environment from scratch in an EC2 instance. The file is located in the chapter 12 repository ([https://github.com/ihuaylupo/manning-smia/blob/master/chapter12/AWS/jenkins\\_Setup.md](https://github.com/ihuaylupo/manning-smia/blob/master/chapter12/AWS/jenkins_Setup.md)).

### 12.5.1 Setting up Github

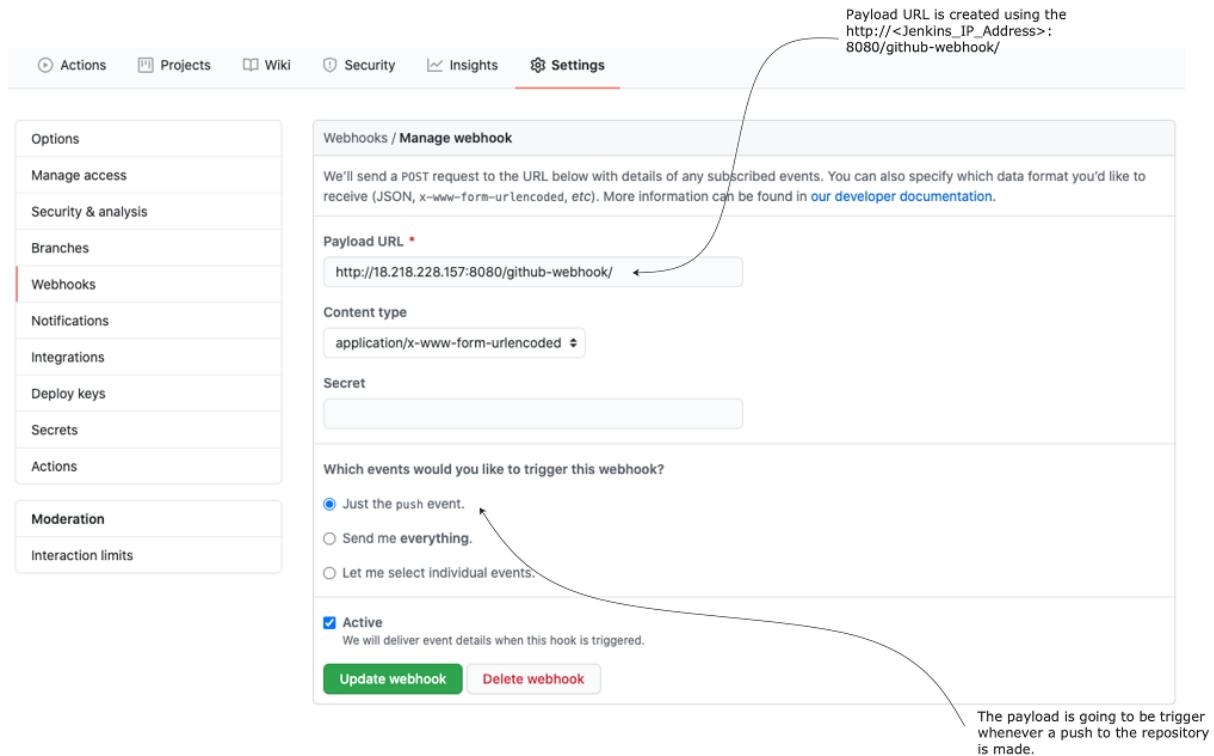
To start creating our pipeline, we must create a GitHub Webhook. What is a Webhook? A webhook is also known as a

web HTTP callback. These HTTP callbacks provide real-time information to other applications as it happens. Usually, they are triggered when a process or action is being executed on the application containing the webhook. If you haven't heard about this, you might be wondering why we need it. For our purposes, we are going to create a webhook in GitHub so it lets Jenkins know when a code push has been made so that way Jenkins can start the specific build process.

To create a Webhook in GitHub, we must do the following steps

1. Go to the repository containing your pipeline projects
2. Click the settings option
3. Select Webhooks
4. Click the "Add Webhook" button.
5. Provide a specific payload URL
6. Click the "Just the push event" option
7. Finally, click the Add Webhook button.

The following figure 12.18, shows this process.



## Figure 12.18 Generating a Webhook for Jenkins.

It's important to highlight that the Payload URL is created using the IP address of the Jenkins. This way GitHub can call the Jenkins whenever a push to the repository is made.

Now, that we have the webhook let's configure the pipeline using Jenkins.

### 12.5.2 Enabling our services to build in Jenkins

The heart of every service built in this book has been a Maven pom.xml file that's used to build the Spring Boot service, package it into an executable JAR, and build a Docker image that can be used to launch the service. Up

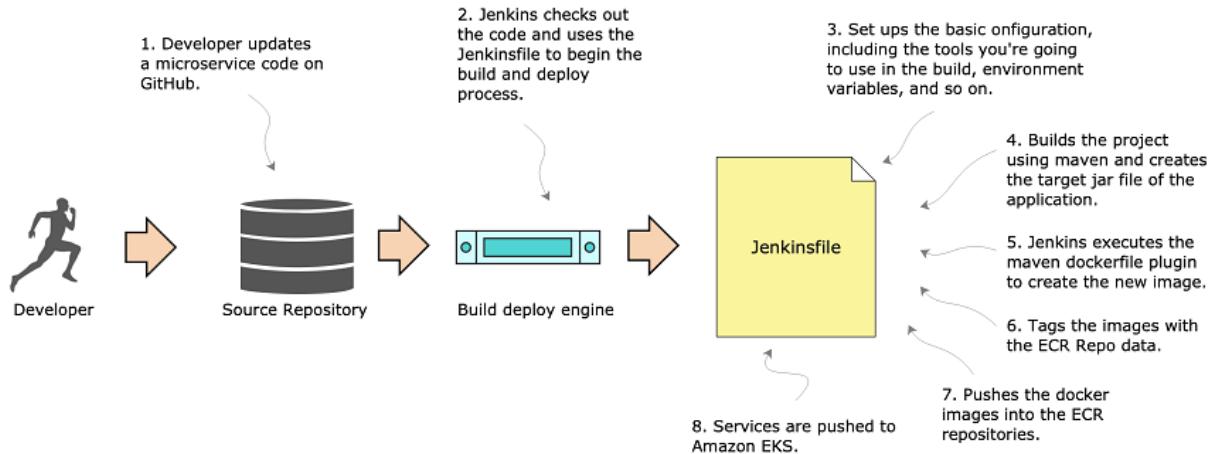
until this chapter, the compilation and startup of the services occurred by

1. Opening a command-line window on your local machine.
2. Running the Maven script for the chapter. This builds all the services for the chapter and then packages them into a Docker image that would be pushed to a locally running Docker repository.
3. Launching the newly created Docker images from your local Docker repo, by using docker-compose and docker-machine to launch all the services for the chapter.

The question is, how do we repeat this process in Jenkins? It all begins with a single file called JenkinsFile. The JenkinsFile is a script that describes the actions we want to execute when Jenkins executes our build. This file is stored in the root directory of our microservices GitHub repository.

When a push occurs on a GitHub repository, the webhook calls the Jenkins using the payload and the Jenkins job searches for the JenkinsFile to initiate the build process.

Figure 12.19 shows the steps the Jenkinsfile will undertake when a push is made to the GitHub repository.



**Figure 12.19 The concrete steps undertaken by the Jenkinsfile to build and deploy your software**

1. A developer makes a change to one of the microservices in the GitHub repository.
2. Jenkins is notified by GitHub that a push has occurred. This notification is made by the GitHub Webhook. Jenkins starts a process that will be used to execute the build. Jenkins will then check out the source code from GitHub and then use the Jenkinsfile to begin the overall build and deploy process.
3. Jenkins sets up the basic configuration in the build and installs any dependencies.
4. Jenkins builds the project executing the unit and integration tests and generates the target JAR file of the application.
5. Jenkins executes the maven docker-file plugin to create the new Docker images.

6. Then it tags the new images with the ECR repository data.
7. The build process pushes the images to the ECR with the same tag name you used to tag in step 6.
8. Your build process then connects to the EKS cluster and deploys the services using the service.yaml and deployment.yaml files.

## A quick note

For this book's purposes, I set up separate folders in the same GitHub repository for each chapter of the book. All the source code for the chapter can be built and deployed as a single unit. However, outside this book, I highly recommend that you set up each microservice in your environment with its own repository with its own independent build processes. This way, each service can be deployed independently of one another. With the build process example, I'm deploying the config server as a single unit only because I wanted to push the projects separately to the Amazon cloud and manage build scripts for each individual service. In this example, I'm only going to deploy the config server, but later on, you can use the same steps to create the other deployments.

Now that we've walked through the general steps involved in the Jenkinsfile, let's look at how to create the Jenkins pipeline to start.

**NOTE** To make the pipeline work you need to install several Jenkins Plugins such as GitHub Integration, GitHub, Maven Invoker, Maven Integration, Docker Pipeline, ECR, Kubernetes Continuous deploy and Kubernetes CLI plugins. To learn more about this plugins I highly recommend you read the official Jenkins documentation, for example for the Kubernetes CLI is the following <https://plugins.jenkins.io/kubernetes-cli/>.

Once installed all the plugins, we need to create the Kubernetes credentials and the ECR Credentials. First, let's start with the Kubernetes credentials to create them, we

need to go to Manage Jenkins, Manage Credentials, Jenkins, Global Credentials, and click the Add Credentials file. Then select the option Kubernetes Configuration (kubeconfig) and fill the information with your Kubernetes cluster info.

To retrieve the information from the Kubernetes cluster just connect to the Kubernetes cluster like I previously explained when we deployed manually our services and execute the following command:

```
kubectl config view
```

Copy the contents and paste it in the content section as shown in figure 12.20.

The diagram shows a screenshot of the Jenkins 'Manage Credentials' interface. A callout arrow originates from the text 'kubectl config view' and points to the 'Content' field of the 'Kubeconfig' credential entry. The 'Content' field contains a YAML snippet for AWS IAM authentication.

Kind: Kubernetes configuration (kubeconfig)

Scope: Global (Jenkins, nodes, items, all child items, etc)

ID: kubeconfig

Description: EKS Kubeconfig

Kubeconfig:  Enter directly

Content:

```
command: aws-iam-authenticator
env:
- name: AWS_STS_REGIONAL_ENDPOINTS
 value: regional
- name: AWS_DEFAULT_REGION
 value: us-east-2
```

From a file on the Jenkins master  
 From a file on the Kubernetes master node

OK

## **Figure 12.20 Configure KubeConfig credentials to connect to the EKS cluster**

Now, let's create the ECR credentials. In order to do create them, first we need to go to the IAM in the AWS console, next go to users and click the Add user option. In the creation page you need to specify a username, the access type, the policies, next download the .csv with the user credentials and finally save the user. I will guide you step by step, in the first page add the following data:

```
User name: ecr-user
Access type: Programmatic access
```

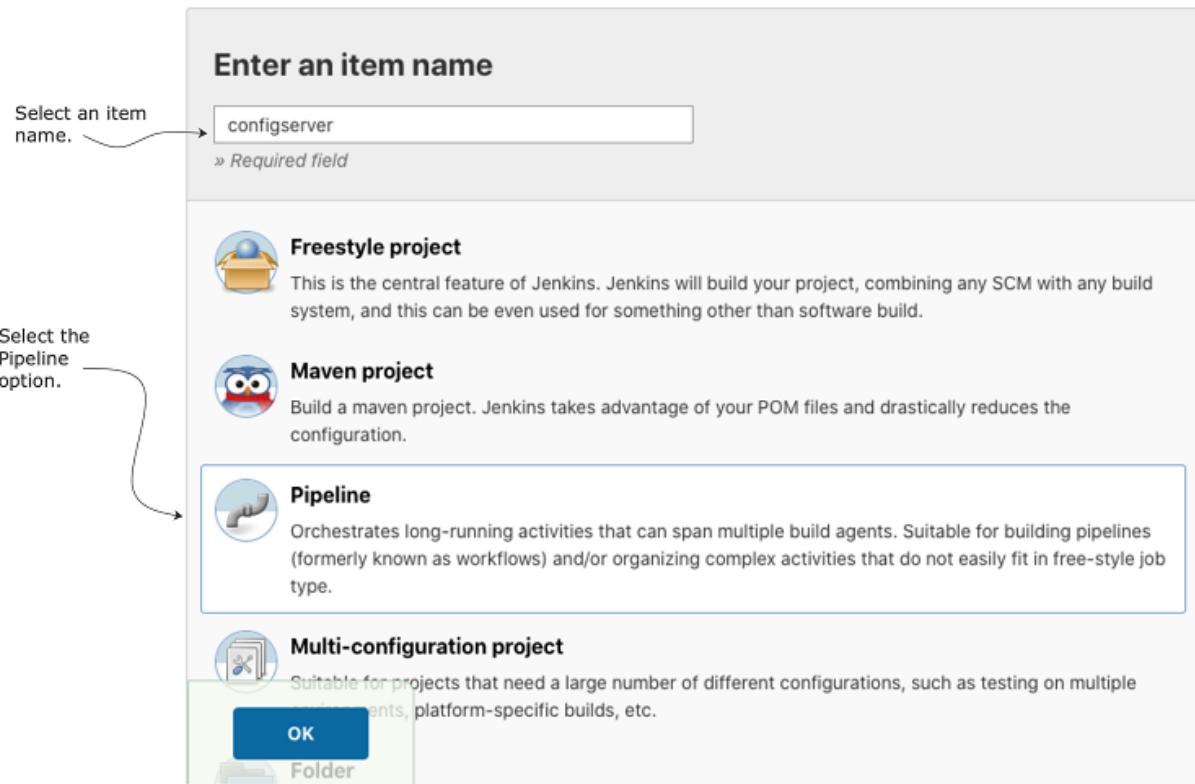
In the second page, the permissions page select the Attach existing policies directly option and search for the AmazonEC2ContainerRegistryFullAccess and select it. Finally, on the last page, click the download .csv. The download step is really important, because we are going to need those credentials later on. Finally, click the save button.

The next step is to add the credentials in Jenkins, but before that make sure you already installed the ECR plugin in the Jenkins console. Once installed, go to Manage Jenkins, Manage Credentials, Jenkins, Global Configuration and click the add credentials option.

In this page add the following data:

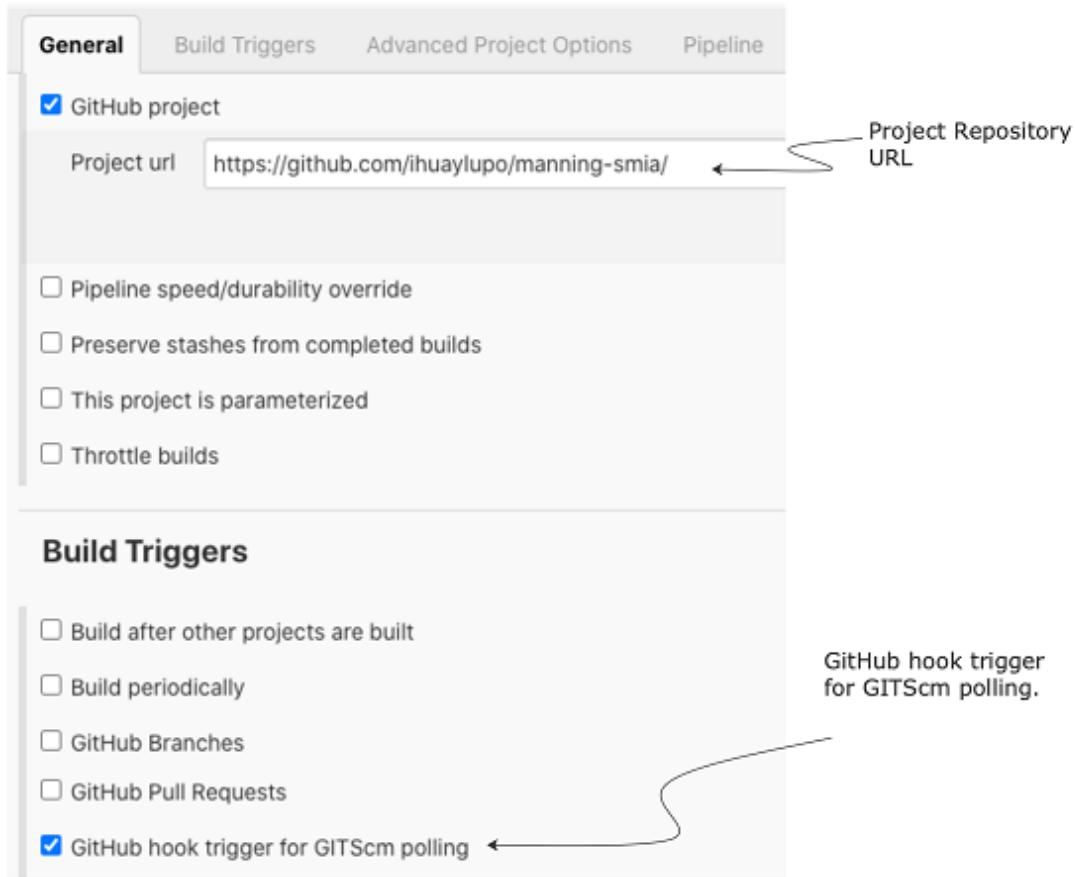
```
ID: ecr-user
Description: ECR User
Access Key ID: <Access_key_id_from_csv>
Secret Access Key: <Secret_access_key_from_csv>
```

Once created the credentials, you can continue with the pipeline creation. The following figure 10.21, shows the pipeline setup process. To access the page shown in figure 10.20 just go to the Jenkins dashboard and click the new item option at the top left of the screen.



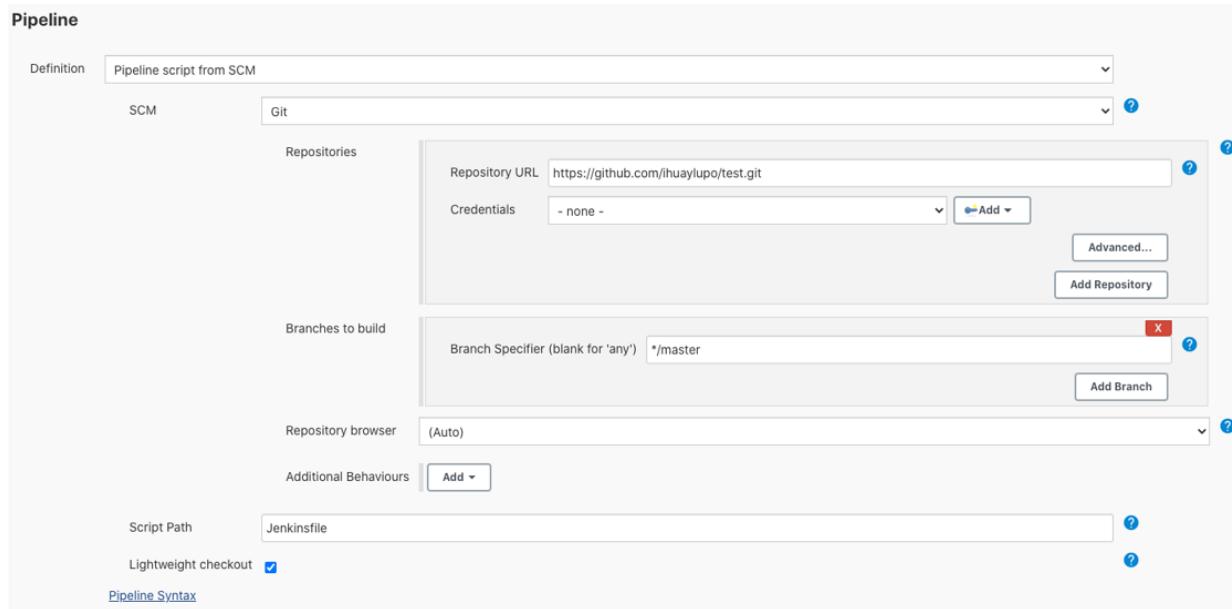
**Figure 12.21 Creating a Jenkins pipeline to build the config server.**

The next step is to add the GitHub repository URL and select the GitHub hook trigger for GITScm polling option. This option enables the webhook payload we configured in the previous section. The following figure 12.22 shows the option.



**Figure 12.22 Configuring the GitHub WebHook and repository URL in the Jenkins pipeline.**

Finally, the last step is to select the Pipeline Script from SCM, this is located under the Pipeline section in the Definition dropdown list. This last step allows us to search for the Jenkinsfile in the same source code repository as the application. The following figure 12.23 shows this process.



## Figure 12.23 Configuring the GitHub WebHook and repository URL in the Jenkins pipeline.

In the previous figure 12.23 we select Git from the SCM dropdown list and then we add all the specific information such as the Repository URL, the branch and the Script path that in our case is the root and the filename is Jenkinsfile.

**NOTE** If your repository needs credentials an error is going to be displayed.

Finally click apply and save to go to the pipeline homepage.

### 12.5.3 Understanding and generating the pipeline script

The Jenkinsfile deals with configuring the core runtime configuration of your build. Typically, this file is divided into several sections. It's important to highlight that you can

have as many sections/stages as you want. The next code listing 12.3 shows the Jenkinsfile for the config-server.

### **Listing 12.3 Jenkinsfile**

```
node {
def mvnHome
stage(' Preparation') { #A
git ' https://github.com/ihuaylupo/spmia.git' #B
mvnHome = tool 'M2_HOME' #C
}
stage(' Build') { #D
// Run the maven build
withEnv(["MVN_HOME=$mvnHome"]) {
if (isUnix()) {
sh "${mvnHome}/bin/mvn' -Dmaven.test.failure.ignore clean package" #E
} else {
bat(/"%MVN_HOME%\bin\mvn" -Dmaven.test.failure.ignore clean package/)
}
}
}
stage(' Results') { #F
junit '**/target/surefire-reports/TEST-*.*xml'
archiveArtifacts ' configserver/target/*.jar'
}
stage(' Build image') { #G
sh " ${mvnHome}/bin/mvn' -Ddocker.image.prefix=8193XXXXXX43.dkr.ecr.us-east-
2.amazonaws.com/ostock -Dproject.artifactId=configserver -
Ddocker.image.version=latest dockerfile:build" #H
}
stage(' Push image') { #I
docker.withRegistry(' https://8193XXXXXX43.dkr.ecr.us-east-2.amazonaws.com' ,
' ecr:us-east-2:ecr-user') {
sh "docker push 8193XXXXXX43.dkr.ecr.us-east-
2.amazonaws.com/ostock/configserver:latest" #J
}
}
stage(' Kubernetes deploy') { #K
kubernetesDeploy configs: ' configserver-deployment.yaml' , kubeConfig: [path: ''],
kubeconfigId: ' kubeconfig' , secretName: '' , ssh: [sshCredentialsId: '*' ,
sshServer: ''], textCredentials: [certificateAuthorityData: '' ,
clientCertificateData: '' , clientKeyData: '' , serverUrl: ' https://']
}
}
```

#A Defines the Preparation stage.

#B Sets again the git repository

#C Exposes the maven configuration from the Jenkins server.

#D Defines the build stage.

#E Executes the maven clean package goals.

```
#F Obtains the JUnits results and then archives the Application jar file that we
need for the Docker image.
#G Defines the build Image stage
#H Executes the maven dockerfile:build goal. This goal is in charge of creating
the new Docker image and also tags the image with the ECR repository
information.
#I Defines the push image stage.
#J In charge of pushing the images into the ECR repository.
#K Defines the Kubernetes deploy stage.
```

This file describes all of the steps involve in our pipeline, as you can see the file contains straightforward commands to execute each stage. In the first stage, we add the git configuration by doing this step we can remove the step where we define the GitHub URL repository in the build section. In the second stage, the second command line exposes the Maven installation path to use the maven command throughout our Jenkinsfile.

In the third stage, we define the command we will use to package our application in this case, I'm using the clean package goals. For this process we are going to use the maven environment variable where we define the maven home root directory.

The fourth stage allows us to execute the dockerfile:build maven goal that is in charge of creating the new Docker image. If you'd remember, we defined the maven spotify-dockerfile plugin in the pom.xml files of our microservices. If you take a closer look the dockerfile:build you'll notice that I'm sending some parameters to the maven. Let's take a look.

```
sh "${mvnHome}/bin/mvn' -Ddocker.image.prefix=8193XXXXXX43.dkr.ecr.us-east-
2.amazonaws.com/ostock -Dproject.artifactId=configserver -
Ddocker.image.version=latest dockerfile:build"
```

---

In this command, I'm passing the docker image prefix, the image version, and the artifact ID. But here you can send as many parameters as you want. Also, here is a good place to set the profile you are going to be using in the deployment. For example: Dev, stage, production.

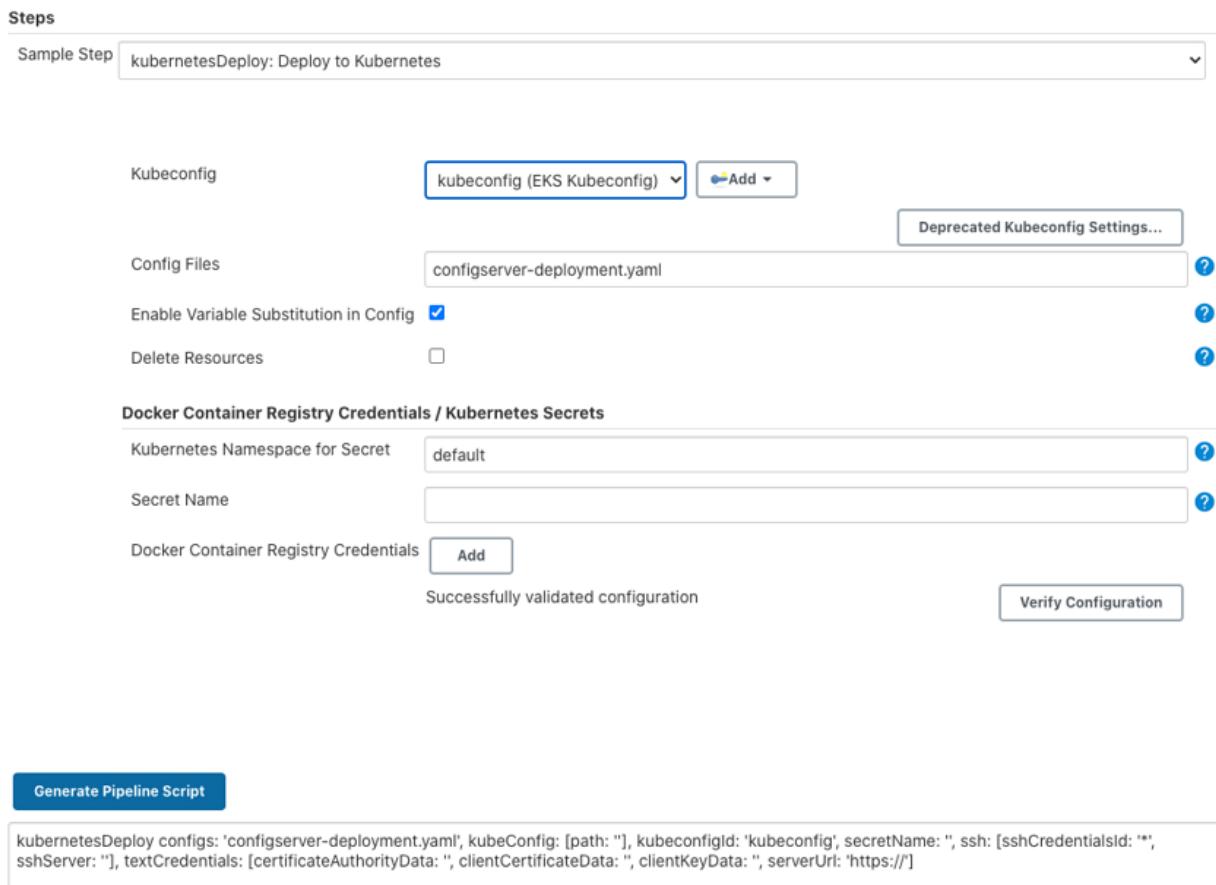
Also, in this stage, we create and assign the specific tags with the ECR repository data to the new images. In this particular scenario, I'm using the latest version, but here we can specify different variables to create different docker images.

The fifth stage is in charge of pushing the images to the ECR repositories. In this case we will add the config server just created Docker image into the 8193XXXXXX43.dkr.ecr.us-east-2.amazonaws.com/ostock repository.

The final stage is in charge of making the deployment to our EKS cluster. This command may vary depending on your configuration, so I will give you a tip on how to create the pipeline script for this stage automatically.

## **12.5.4 Creating the Kubernetes pipeline scripts**

Jenkins offers a Pipeline Syntax option allows us to generate automatically different pipeline snippets that we can use in our Jenkinsfile. Figure 12.24 shows this process.



## Figure 12.24 Using the pipeline syntax option to generate the **KubernetesDeploy** command.

To use this option, we need to execute the following steps:

1. Go to the Jenkins dashboard page and click your pipeline.
2. Then click the Pipeline Syntax option shown in the left column.
3. In the Snippet Generator page, click the sample Step dropdown and select the **kubernetesDeploy: Deploy to Kubernetes** option.

4. In the Kubeconfig dropdown list select the Kubernetes credentials that we created in the previous section.
5. Then in the Config files, select the deployment.yaml file. This deployment yaml file is located in the root of the GitHub project.
6. Leave the other values as default.
7. Click the generate pipeline script.

**NOTE** Remember, for this book's purposes, I set up separate folders in the same GitHub repository for each chapter of the book. All the source code for the chapter can be built and deployed as a single unit. However, outside this book, I highly recommend that you set up each microservice in your environment with its own repository with its own independent build processes.

Once, everything set up you can now start doing changes to your code and your Jenkins pipeline will automatically be triggered.

## 12.6 Closing thoughts on the build/deployment pipeline

As this chapter closes out (and the book), I hope you've gained an appreciation for the amount of work that goes into building a build/deployment pipeline. A well-functioning build and deployment pipeline is critical to the deployment of services. The success of your microservice architecture depends on more than just the code involved in the service:

- Understand that the code in this build/deploy pipeline is simplified for the purposes of this book. A good build/deployment pipeline will be much more

generalized. It will be supported by the DevOps team and broken into a series of independent steps (compile > package > deploy > test) that the development teams can use to “hook” their microservice build scripts into.

- The virtual machine imaging process used in this chapter is simplistic, with each microservice being built using a Docker file to define the software that’s going to be installed on the Docker container. Many shops will use provisioning tools like Ansible(<https://github.com/ansible/ansible>), Puppet (<https://github.com/puppetlabs/puppet>), or Chef (<https://github.com/chef/chef>) to install and configure the operating systems onto the virtual machine or container images being built.
- The cloud deployment topology for your application has been consolidated to a single server. In the real build/deployment pipeline, each microservice would have its own build scripts and would be deployed independently of each other to a cluster EKS container.

Writing microservices is a challenging task; in microservices, the complexity of creating applications doesn’t go away; it is just transformed. The concepts around building individual microservices are easy to understand, but running and supporting a robust microservice architecture involves more than writing the code. Remember, microservices gives us a lot of options and decisions to make on how to create our architecture. But when taking these decisions, we need to consider that microservices are loosely coupled, abstract, independent, and constrained.

Hopefully, by now, I’ve given you enough information and experiences to create your own microservices architecture. In this book, you’ve learned what a microservice is and the critical design decisions needed to operationalize the

microservices. Also, how to create a complete microservice architecture following the twelve factors best practices (configuration management, service discovery, messaging, logging, tracing, security), using a variety of tools and technologies that can be perfectly combined with Spring to deliver a robust microservices environment and, finally, how to implement a build/deployment pipeline. All these previously mentioned topics create the perfect recipe to develop successful microservices. In closing, as Martin Fowler says, "if we don't create good architecture, then, in the end, we're deceiving our customers because we're slowing down their ability to compete."

## 12.7 Summary

- The build and deployment pipeline is a critical part of delivering microservices. A well-functioning build and deployment pipeline should allow new features and bug fixes to be deployed in minutes.
- The build and deployment pipeline should be automated with no direct human interaction to deliver a service. Any manual part of the process represents an opportunity for variability and failure.
- The build and deployment pipeline automation do require a great deal of scripting and configuration to get right. The amount of work needed to build it shouldn't be underestimated.
- The build and deployment pipeline should deliver an immutable virtual machine or container image. Once a server image has been created, it should never be modified.
- Environment-specific server configuration should be passed in as parameters at the time the server is set up.



# **Appendix A. Microservice Architecture Best Practices**

This appendix covers

- Understanding the best practices to create a microservice architecture

## **A.1 Microservice Architecture best practices**

In chapter 2, I explained some of the best practices that we should consider when creating a microservice, but with this appendix I want to go deeper into the best practices we should consider while we are creating a microservices architecture.

In the next sections, I will give you some guidelines (or best practices) to create a successful microservice architecture. It is important to highlight that there isn't a well-defined set of rules to achieve this, however, Hüseyin Babal, in his talk Ultimate Guide to Microservice Architecture presented a set of best practices to achieve a flexible, efficient, and scalable design.

I consider this talk as essential among the developer's community because it not only gives guides on how to develop a microservice architecture but also highlights which are the fundamental components of a microservice.

In this section, I have selected some of the practices that I consider most important and those that we will be developing throughout the book to create a flexible and successful architecture.

**NOTE** If you're interested in knowing all of the best practices mentioned in the Ultimate Guide to Microservice Architecture talk, please visit the following link that contains a video (<https://www.youtube.com/watch?v=CTnMUE06oZI>)

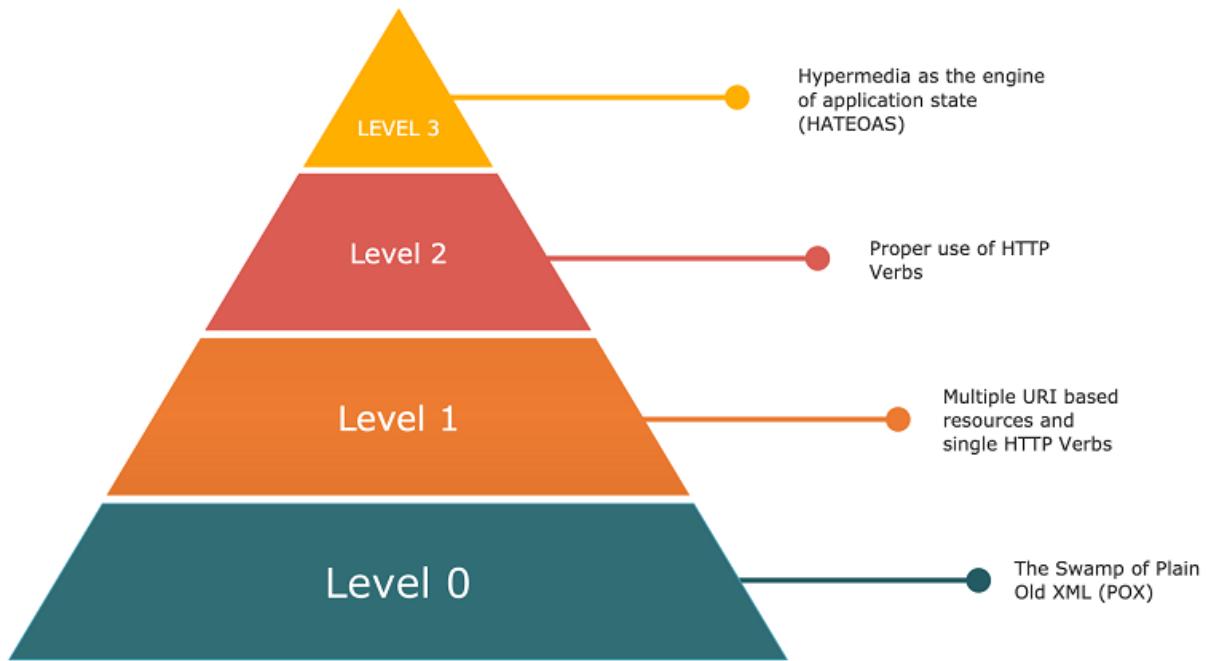
These are some of the best practices he mentioned

- Richardson level of maturity.
- Spring HATEOAS
- Distributed configuration
- Continuous delivery
- Monitoring
- Logging
- Application performance management
- API Gateways

### A.1.1 Richardson level of maturity

This best practice, suggests the use of the Richardson's Maturity Model described by Martin Fowler (<http://martinfowler.com/articles/richardsonMaturityModel.html>) as a guide in order to understand the main principles of REST architectures and to evaluate our REST architecture. It's important to highlight that we can see these levels more as a tool to help us understand the components of REST and

the ideas behind the restful thinking than a kind of assessment mechanism to your company. Figure 1.1 shows the different levels of maturity used to evaluate a service.



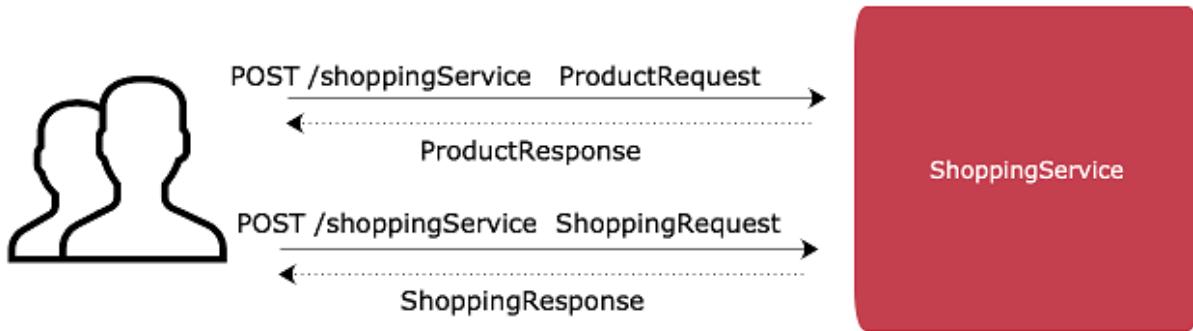
**Figure 1.1 Levels of the Richardson's Maturity Model**

## **LEVEL 0**

This first level of the Richardson Maturity Model represents the basic functionalities expected of an API. Basically, the API interaction in this point is a basic Remote procedure call (RPC) to a single URI, with mostly a back and forth XML in every request and response specifying the action, the target and the parameters to execute the service.

For example, let's assume that we want to buy a specific product from an online store. First, the store's software

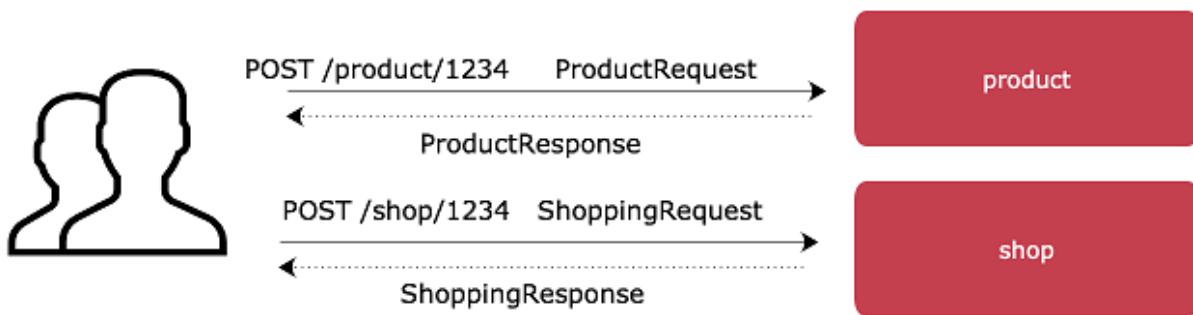
needs to verify if they have the product in stock in order to buy it, so at this level the online store will expose a service endpoint with a single URI. Figure 1.2 shows the level 0 services requests and responses.



**Figure 1.2 Level 0, simple RPC with a plain old XML (POX) back and forth.**

## **LEVEL 1**

The main idea of this level is to perform the actions to individual resources rather than making all of the requests to a singular service endpoint. Figure 1.3 shows, the individual resources with their requests and responses.

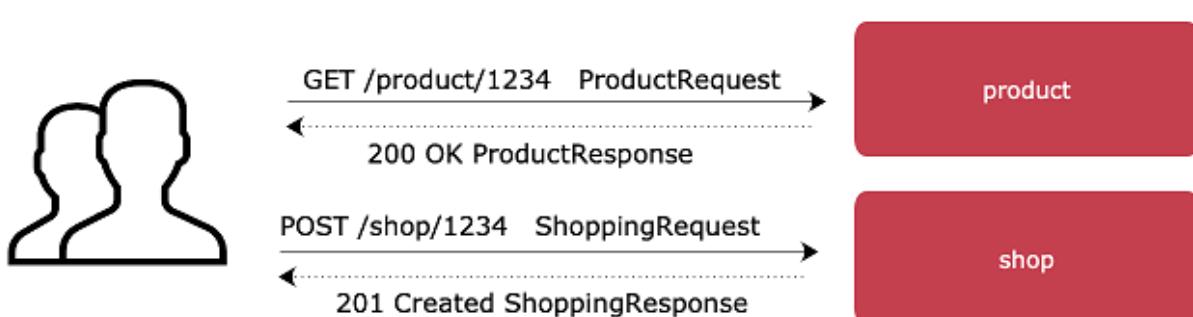


**Figure 1.3 Level 1, individual resources call to perform the actions.**

## LEVEL 2

At level 2, the services use HTTP verbs in order to perform actions. The GET http verb is used to retrieve, the POST to create, the PUT to update, the DELETE to remove.

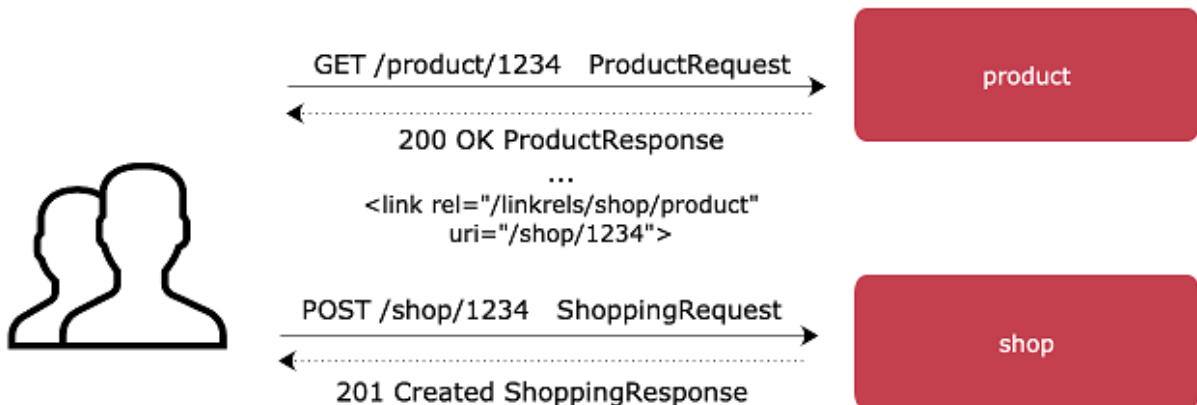
Remember, as I previously mentioned these levels more as a tool to help us understand the components of REST, in this particular scenario REST advocates mention to use all the HTTP Verbs. Figure 1.4 shows how to implement the use of http verbs in the store example.



**Figure 1.4 Level 2, the use of the GET http verb to retrieve the product information and the use of the POST http verb to create the shopping order.**

## **LEVEL 3**

The final level introduces the hypertext as the Engine of the Application State (HATEOAS). By implementing the HATEOAS we can make our API respond with additional information such as link resources to create richer interactions as shown in Figure 1.5.



**Figure 1.5 Level 3, the use of HATEOAS on the retrieve product response. In this specific case the link shows additional information on how to buy that specific product.**

Using this model, it is easy to explain that there is no definitive way to do REST, but we can choose the degree of adherence that best suits the needs of our projects.

## A.1.2 Spring HATEOAS

Spring HATEOAS (<https://spring.io/projects/spring-hateoas>) is a small project of Spring that allow us to create APIs that follow the HATEOAS principle explained in the Level 3 of the Richardson Maturity Model. With this project you can quickly create model classes for links, resource representation models, it also provides a link builder API to create specific links that point to Spring MVC controller methods, and more.

## A.1.3 Externalized configuration

This best practice goes hand by hand with the config best practice listed in the twelve-factor app mentioned in chapter two; the configuration of our microservices should never be in the same repository of our source code. Why is it that important? Microservices architectures are composed of a collection of services (microservices) that run on separate processes. Each service can be deployed and scaled independently, creating more instances of the same microservice. So, let's say that you, as a developer, want to change a configuration file of a specific microservice that has been scaled several times. If you don't follow this best practice and you have the configuration packaged within the deployed microservice, you'll be forced to redeploy each instance. Not only is a time-consuming task, but it also can lead to configuration issues between the microservices.

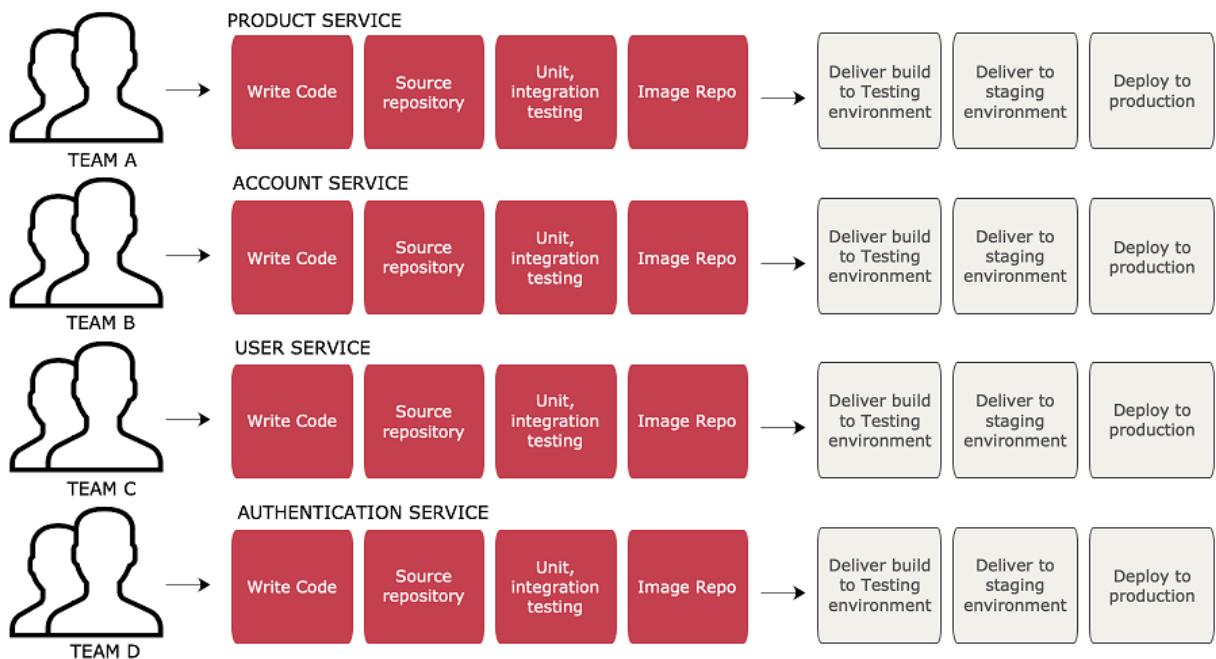
There are many ways to implement externalized configurations in a microservice architecture, but in this book, I'm going to use the Spring Cloud Config that I described on chapter 2.

## **A.1.4 Continuous Integration and Continuous delivery**

Continuous integration is a series of software development good practices where the team members integrate their changes to a repository in a short period, in order to detect possible errors and analyze the quality of the software they are creating. This is accomplished by using an automatic and continuous code check (build) that includes the execution of tests.

On the other hand, continuous delivery is a software development practice in which the process of delivering software is automated to allow short-term deliveries into a production environment.

When we are going to apply these processes into our microservice architecture, it's essential to keep in mind that there should never be a waiting list to integrate and release changes to production. For example, a team in charge of an "X" service must be able to publish changes to production at any time, without having to wait for the release of other services. Figure 1.6 shows a high-level diagram of how our CI/CD process should look like when we have several teams.



**Figure 1.6 Microservices CI/CD process with a test and staging environment. In this example each team has its own source repository, unit and integration testing, image repository and its own process of deployment. This allows each team to deploy and release newer versions without having to wait for other releases.**

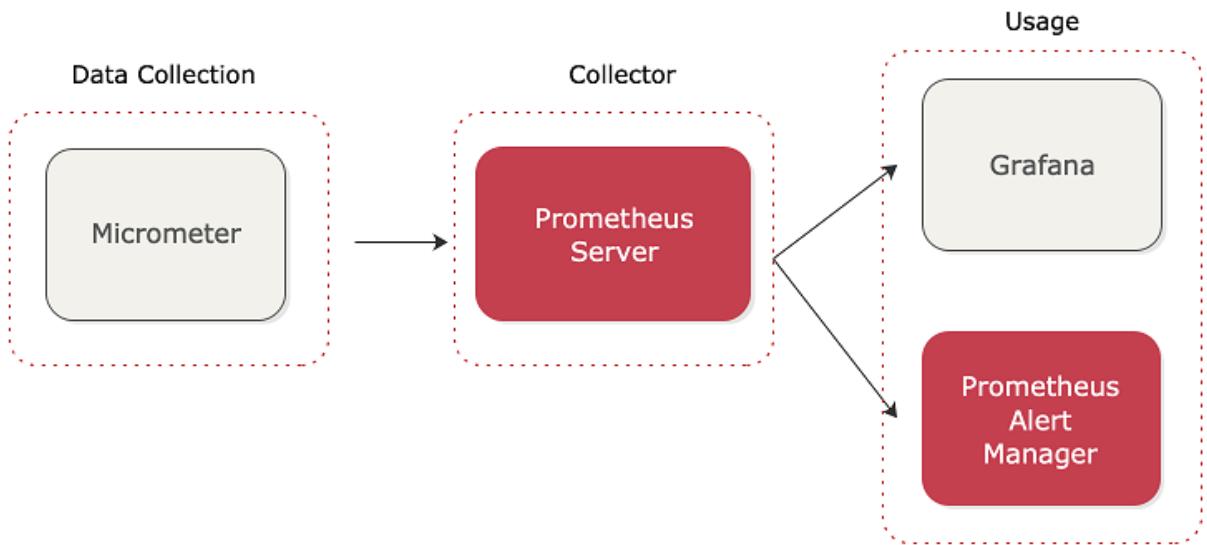
For the provisioning implementations, we're going to make a technology shift. The Spring framework(s) are geared toward application development and don't have tools for creating a "build and deployment" pipeline.

## A.1.5 Monitoring

The monitoring process is critical when we talked about microservices. Microservices are part of large and complex

distributed systems, and remember the more distributed a system is, the more complicated it is to find and solve the problems on it. Imagine we have an architecture with hundreds of microservices, and one of the Microservices starts affecting the performance of other services. How can we know which is the unstable service if we don't implement a suitable monitoring process?

In order to tackle this, I will explain in Appendix C how we can achieve a good monitoring system in our architecture with the following tools: Micrometer (<https://micrometer.io/>) default metrics library in Spring Boot 2, it allows us to obtain not only the application metrics but also the JVM metrics such as garbage collection and memory pools, and more. Prometheus (<https://prometheus.io/>) is an open-source monitoring and alert system whose main objective is to store all data as time series, Grafana (<https://grafana.com/>) is an analysis platform for metrics, which allows you to query, visualize, alert and understand data, no matter where they are stored. Figure 1.7 shows the interaction between micrometer, Prometheus and Grafana.



**Figure 1.7 Interaction between micrometer, Prometheus server and Grafana.**

## A.1.6 Logging

From this moment on, every time you think about logs think about distributed tracing, remember distributed tracing is a tool that helps us understand and debug our microservices by pinpointing where a failure occurs in a microservice architecture. In chapter 11, I explained how to achieve this with Spring Cloud. The traceability of the requests we make within our architecture is another new technological (Spring Cloud Sleuth) situation that comes into action when we speak about microservices. In monolith architectures, the majority of the requests that reached the application were resolved within the same application, the ones that didn't follow this rule were the ones that had an interaction with a database or some other services. However, in microservices architectures, we find an environment composed of several

applications (Spring Cloud), generating that a single client request can go through several applications until it is answered.

So, how can we follow the path of that request in a microservices architecture? By associating a unique request identifier that can be propagated across all the calls and by adding a central log collection in order to see all the entries of that client request.

## A.1.7 API Gateways

An API Gateway is an API REST interface system that provides a central access point to a group of microservices and /or defined third-party APIs. In other words, the Gateway API essentially decouples the interface that clients see from microservices implementation. This is particularly useful to avoid exposing internal services to external clients.

Keep in mind that this system not only interacts as a gateway but also allows you to add additional features such as:

- Authentication and authorization (OAuth2).
- Protection against threats (DoS, code injection, and more).
- Analysis and supervision (who uses their API's, when and how).
- Monitoring of incoming and outgoing traffic.

## A.2 Summary

- There isn't a well-defined set of rules to create a microservice architecture.
- The Richardson Level of Maturity provides a guide to understand the main principles of REST architectures.
- Continuous delivery it's a software development practice in which the process of delivering software is automated to allow short-term deliveries into a production environment.

# Appendix B. OAuth2 grant types

This chapter covers

- OAuth2 Password grant
- OAuth2 Client credentials grant
- OAuth2 Authorization code grant
- OAuth2 Implicit credentials grant
- OAuth2 Token refreshing

From reading chapter 9, you might be thinking that OAuth2 doesn't look too complicated. After all, you have an authentication service that checks a user's credentials and issues a token back to the user. The token can, in turn, be presented every time the user wants to call a service protected by the OAuth2 server.

With the interconnected nature of the web and cloud-based applications, users have come to expect that they can securely share their data and integrate functionality between different applications owned by various services. This presents a unique challenge from a security perspective because you want to integrate across different applications while not forcing users to share their credentials with each application they want to integrate with.

Fortunately, OAuth2 is a flexible authorization framework that provides multiple mechanisms for applications to authenticate and authorize users without forcing them to share credentials. Unfortunately, it's also one of the reasons why OAuth2 is considered complicated. These authentication mechanisms are called authentication grants. OAuth2 has four forms of authentication grants that client applications can use to authenticate users, receive an access token, and then validate that token. These grants are

- Password
- Client credential
- Authorization code
- Implicit

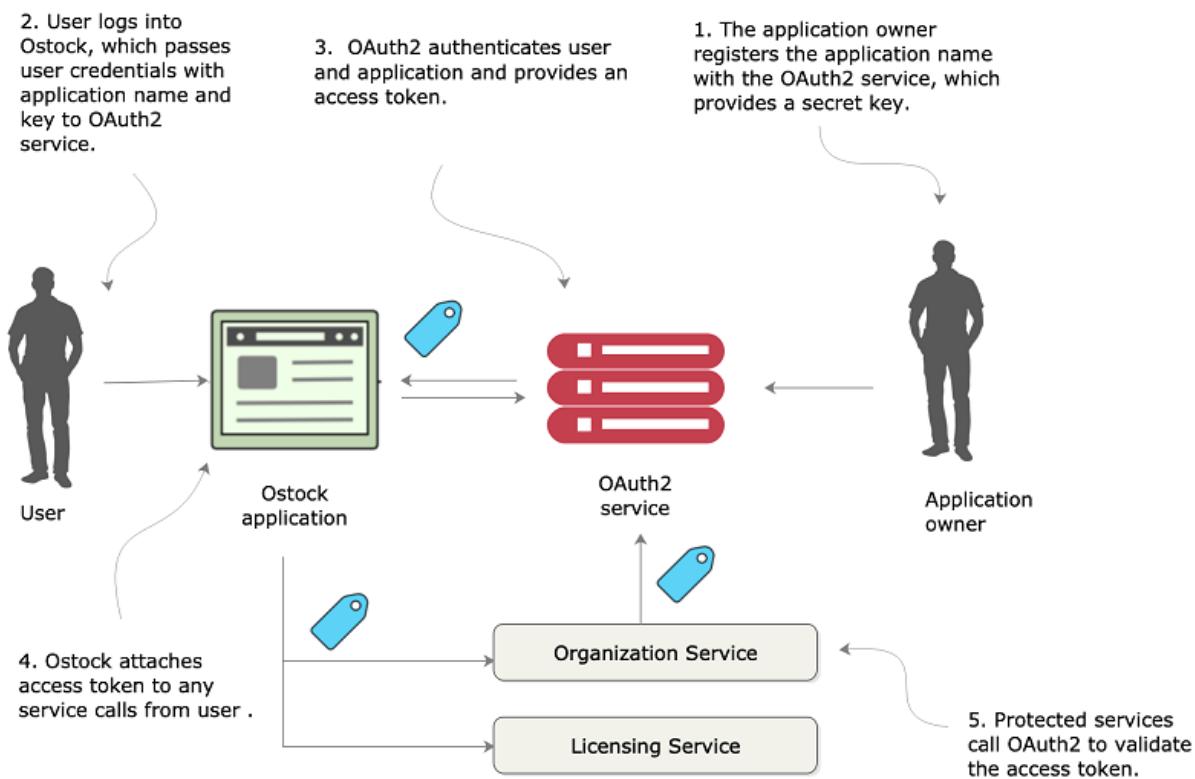
In the following sections, I walk through the activities that occur during the execution of each of these OAuth2 grant flows. I also talk about when to use one grant type over another.

## B.1 Password grants

An OAuth2 password grant is probably the most straightforward grant type to understand. This grant type is used when both the application and the services explicitly trust one another. For example, the Ostock web application and the licensing and organization services are both owned by the same company (OptimaGrowth), so there's a natural trust relationship between them.

**NOTE** To be explicit, when I refer to a “natural trust relationship,” I mean that the same organization completely owns the application and services. They’re managed under the same policies and procedures.

When a natural trust relationship exists, there's little concern about exposing an OAuth2 access token to the calling application. For example, the Ostock web application can use the OAuth2 password grant to capture the user's credentials and directly authenticate against the OAuth2 service. Figure B.1 shows the password grant in action between Ostock and the downstream services.



**Figure B.1 The OAuth2 service determines if the user accessing the service is an authenticated user.**

In figure B.1, the following actions are taking place:

1. Before the Ostock application can use a protected resource, it needs to be uniquely identified within the

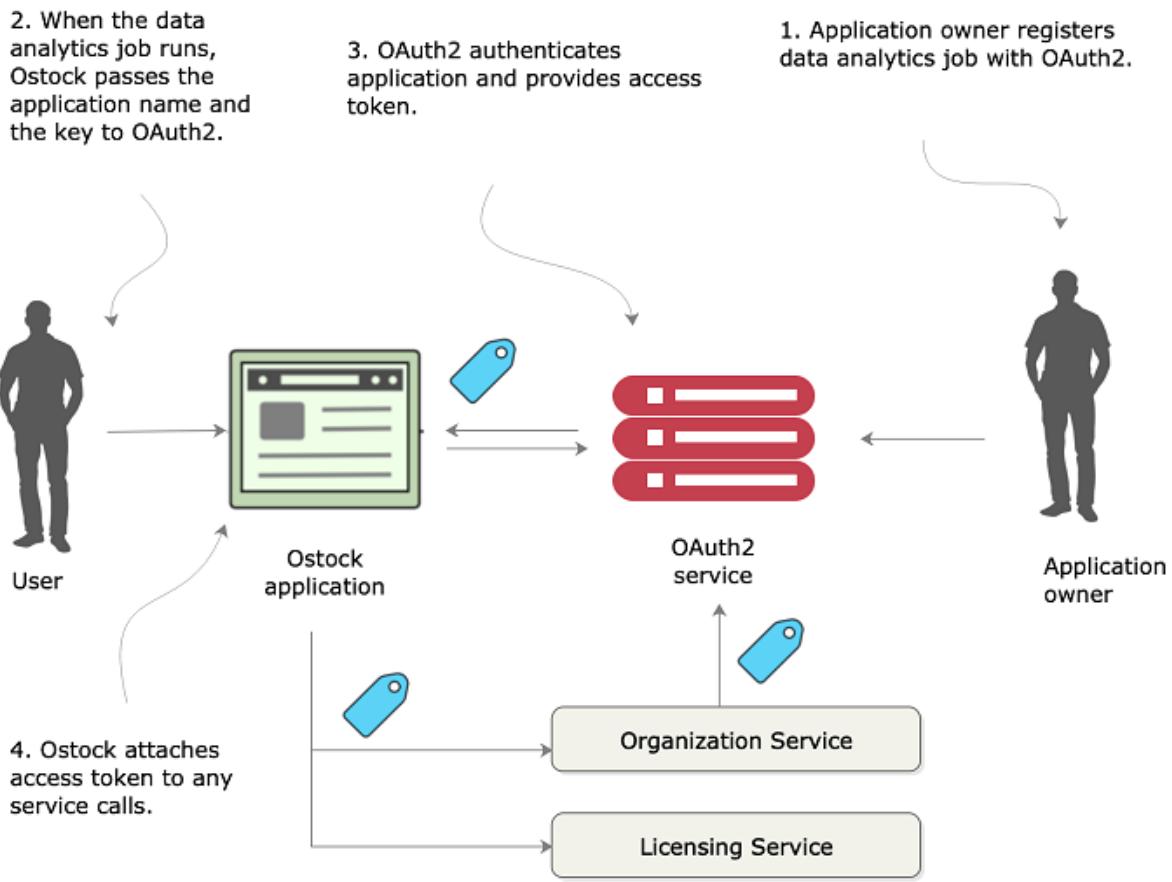
OAuth2 service. Usually, the owner of the application registers with the OAuth2 application service and provides a unique name for their application. The OAuth2 service then provides a secret key back to register the application. The name of the application and the secret key provided by the OAuth2 service uniquely identifies the application trying to access any protected resources.

2. The user logs into Ostock and provides their login credentials to the Ostock application. Ostock passes the user credentials and the application name/application secret key directly to the OAuth2 service.
3. The Ostock OAuth2 service authenticates the application and the user and then provides an OAuth2 access token back to the user.
4. Every time the Ostock application calls a service on behalf of the user, it passes along the access token provided by the OAuth2 server.
5. When a protected service is called (in this case, the licensing and organization service), the service calls back into the Ostock OAuth2 service to validate the token. If the token is good, the service being invoked allows the user to proceed. If the token is invalid, the OAuth2 service returns an HTTP status code of 403, indicating that the token is invalid.

## B.2 Client credential grants

The client credentials grant is typically used when an application needs to access an OAuth2 protected resource, but no human being is involved in the transaction. With the client credentials grant type, the OAuth2 server only authenticates based on the application name and the secret key provided by the resource owner. Again, the client credential task is usually used when the same company owns both applications. The difference between the password grant and the client credential grant is that a client credential grant authenticates by only using the registered application name and the secret key.

For example, let's say that the Ostock application has a data analytics job that runs once an hour. As part of its work, it makes calls out to Ostock services. However, the Ostock developers still want that application to authenticate and authorize itself before it can access the data in those services. This is where the client credential grant can be used. Figure B.2 shows this flow.



**Figure B.2 The client credential grant is for “no-user-involved” application authentication and authorization.**

1. The resource owner registers the Ostock data analytics application with the OAuth2 service. The resource owner will provide the application name and receive back a secret key.
2. When the Ostock data analytics job runs, it will present its application name and secret key provided by the resource owner.
3. The Ostock OAuth2 service will authenticate the application using the application name and the secret

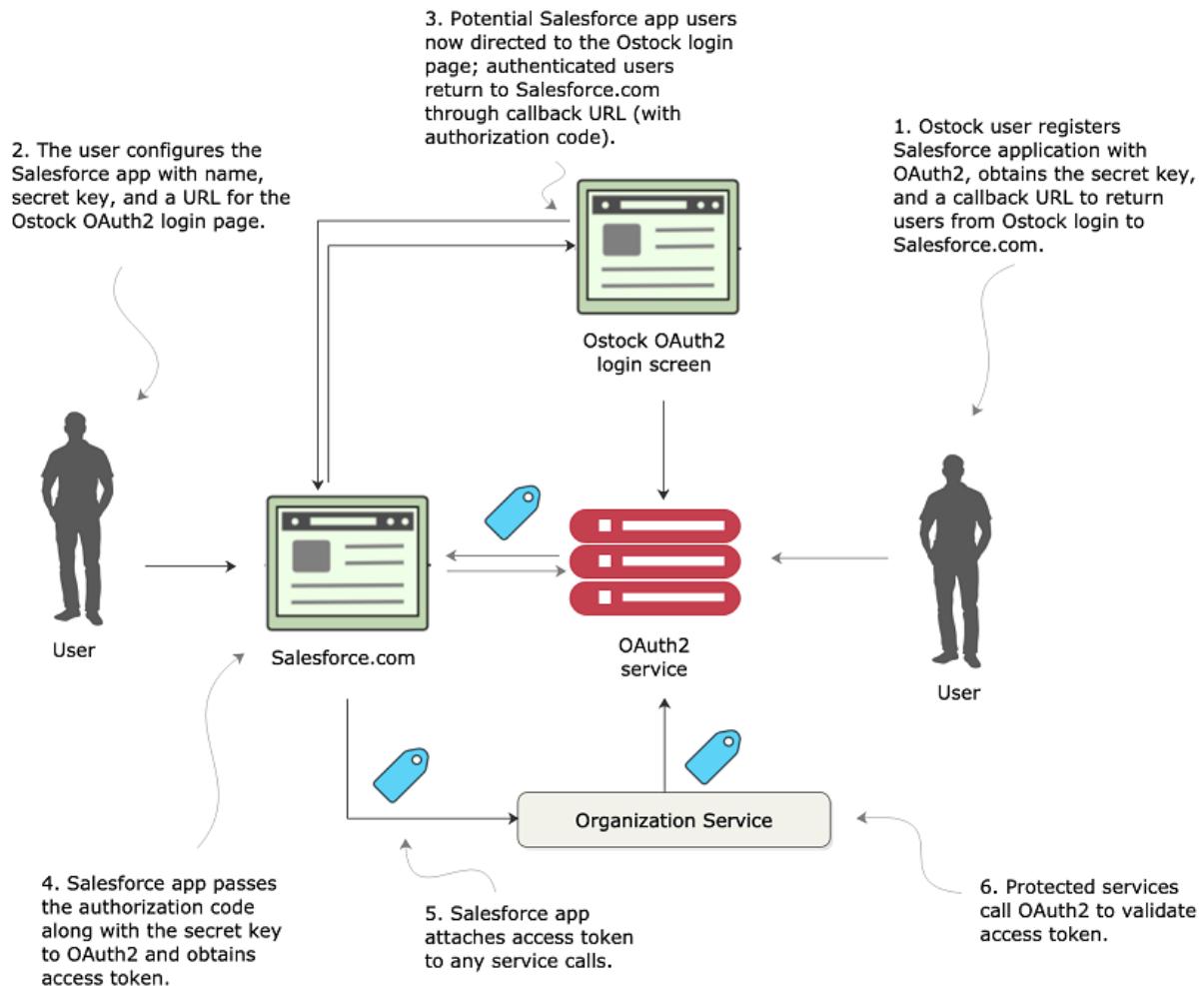
key provided and then return an OAuth2 access token.

4. Every time the application calls one of the Ostock services, it will present the OAuth2 access token it received with the service call.

## B.3 Authorization code grants

The authorization code grant is by far the most complicated of the OAuth2 grants, but it's also the most common flow used because it allows different applications from different vendors to share data and services without having to expose a user's credentials across multiple applications. It also enforces an extra layer of checking by not letting a calling application immediately get an OAuth2 access token, but rather a "pre-flight" authorization code.

The easy way to understand the authorization grant is through an example. Let's say you have an Ostock user who also uses Salesforce.com. The Ostock customer's IT department has built a Salesforce application that needs data from an Ostock service (the organization service). Let's walk through figure B.3 and see how the authorization code grant flow works to allow Salesforce to access data from the organization service, without the Ostock customer ever having to expose their Ostock credentials to Salesforce.



**Figure B.3 The authentication code grant allows applications to share data without exposing user credentials.**

1. The Ostock user logs in to Ostock and generates an application name and application secret key for their Salesforce application. As part of the registration process, they'll also provide a callback URL back to their Salesforce-based application. This callback URL is a Salesforce URL that will be called after the OAuth2 server has authenticated the user's Ostock credentials.

2. The user configures their Salesforce application with the following information:
  - a) Their application name they created for Salesforce
  - b) The secret key they generated for Salesforce
  - c) A URL that points to the Ostock OAuth2 login page
  - d) Now when the user tries to use their Salesforce application and access their Ostock data via the organization service, they'll be redirected over to the Ostock login page via the URL described in the previous bullet point. The user will provide their Ostock credentials. If they've provided valid credentials, the Ostock OAuth2 server will generate an authorization code and redirect the user back to Salesforce via the URL provided in number 1. The authorization code will be sent as a query parameter on the callback URL.
3. The custom Salesforce application will persist the authorization code. Note: this authorization code isn't an OAuth2 access token.
4. Once the authorization code has been stored, the custom Salesforce application can present the Salesforce application the secret key they generated during the registration process and the authorization code back to the Ostock OAuth2 server. The Ostock OAuth2 server will validate that the authorization code is valid and then return an OAuth2 token to the custom Salesforce application. This authorization code is used every time the custom Salesforce needs to authenticate the user and get an OAuth2 access token.
5. The Salesforce application will call the Ostock organization service, passing an OAuth2 token in the header.

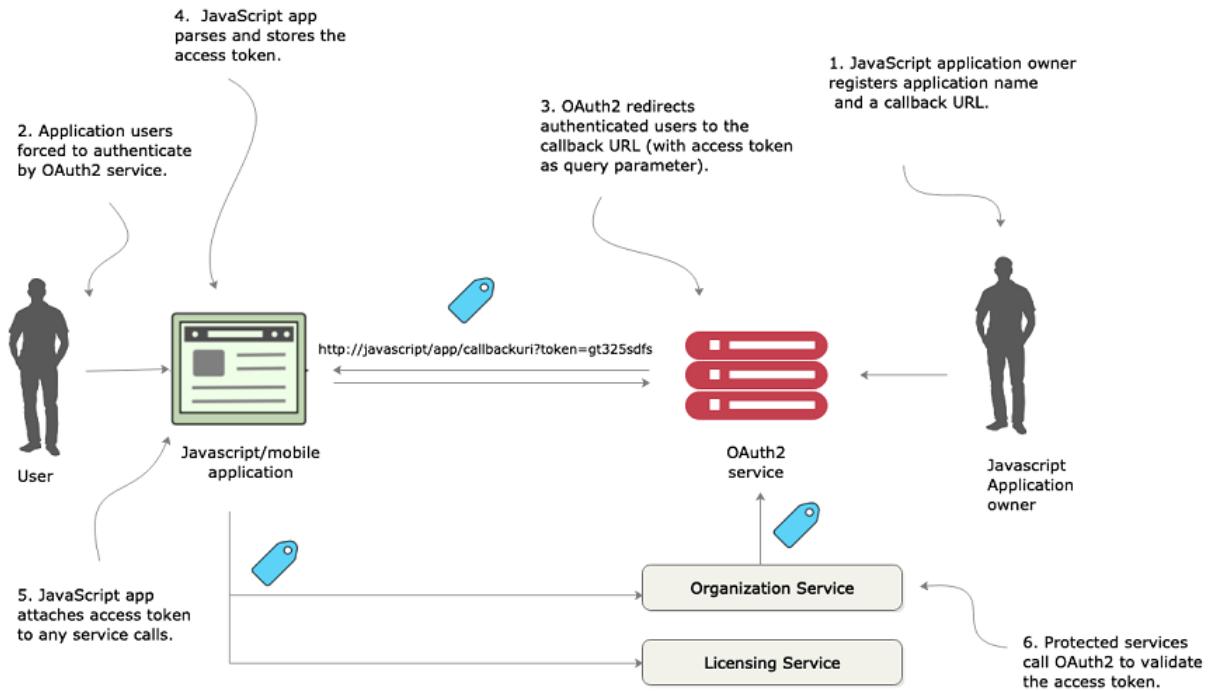
6. The organization service will validate the OAuth2 access token passed into the Ostock service call with the OAuth2 service. If the token is valid, the organization service will process the user's request.

Wow! I need to come up for air. Application-to-application integration is convoluted. The key to note from this entire process is that even though the user is logged into Salesforce and accessing Ostock data, at no time were the user's Ostock credentials directly exposed to Salesforce. After the initial authorization code was generated and provided by the OAuth2 service, the user never had to provide their credentials back to the Ostock service.

## B.4 Implicit grant

The authorization grant is used when you're running a web application through a traditional server-side web programming environment like Java or .NET. What happens if your client application is a pure JavaScript application or a mobile application that runs entirely in a web browser and doesn't rely on server-side calls to invoke third-party services?

This is where the last grant type, the implicit grant, comes into play. Figure B.4 shows the general flow of what occurs in the implicit grant.



**Figure B.4 The implicit grant is used in a browser-based Single-Page Application (SPA) JavaScript application.**

With an implicit grant, you're usually working with a pure JavaScript application running entirely inside the browser. In the other flows, the client communicates with an application server that's carrying out the user's requests, and the application server is interacting with any downstream services. With an implicit grant type, all service interactions happen directly from the user's client (usually a web browser). In figure B.4, the following activities are taking place:

1. The owner of the JavaScript application has registered the application with the Ostock OAuth2 server. They've provided an application name and also a call-back URL

that will be redirected with the OAuth2 access token for the user.

2. The JavaScript application will call the OAuth2 service. The JavaScript application must present a pre-registered application name. The OAuth2 server will force the user to authenticate.
3. If the user successfully authenticates, the Ostock OAuth2 service won't return a token, but instead, redirect the user back to a page the owner of the JavaScript application registered in step one. In the URL being redirected back to, the OAuth2 access token will be passed as a query parameter by the OAuth2 authentication service.
4. The application will take the incoming request and run a JavaScript script that will parse the OAuth2 access token and store it (usually as a cookie).
5. Every time a protected resource is called, the OAuth2 access token is presented to the called service.
6. The called service will validate the OAuth2 token and check that the user is authorized to do the activity they're attempting to do.

Keep several things in mind regarding the OAuth2 implicit grant:

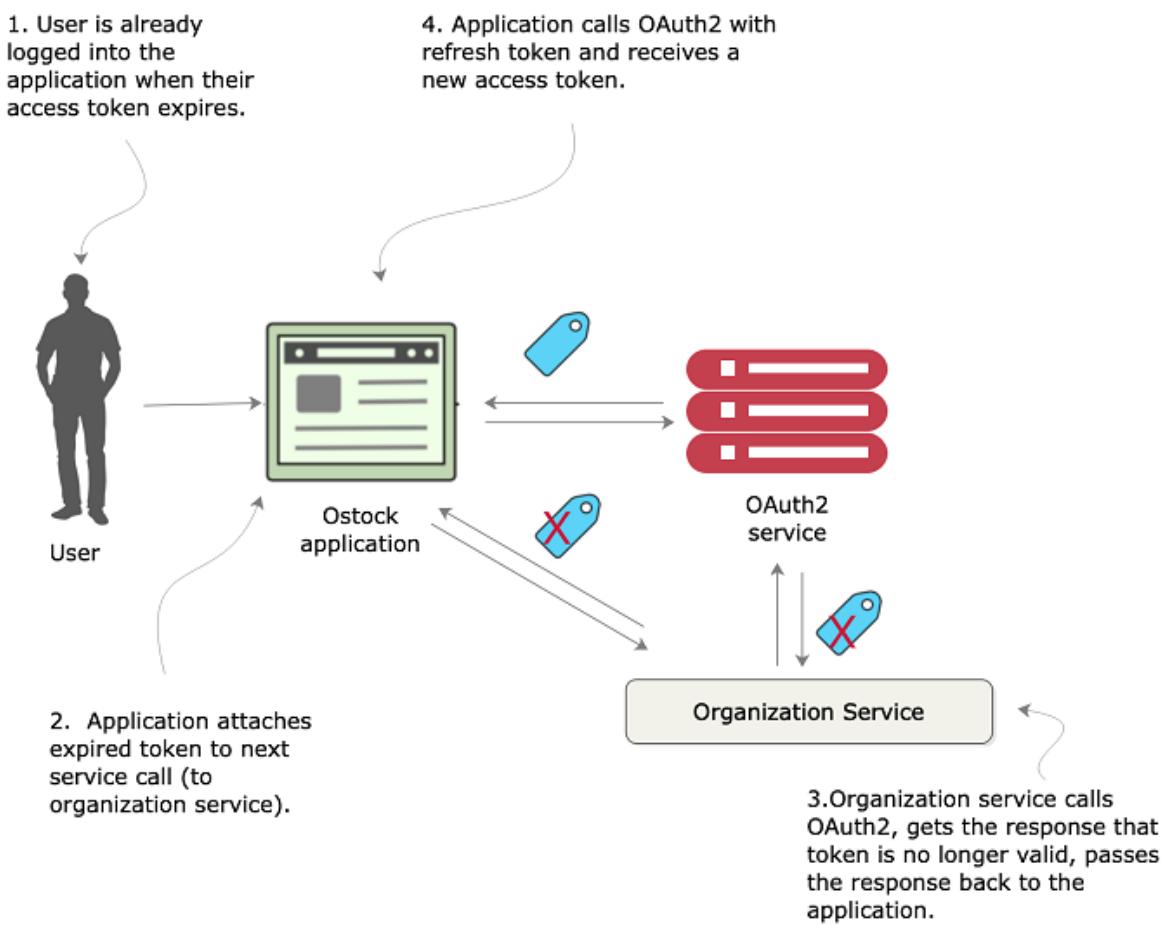
- The implicit grant is the only grant type where the OAuth2 access token is directly exposed to a public client (web browser). In the authorization grant, the client application gets an authorization code returned to the application server hosting the application. With an authorization code grant, the user is granted an OAuth2 access by presenting the authorization code. The

returned OAuth2 token is never directly exposed to the user's browser.

- In the client credentials grant, the grant occurs between two server-based applications. In the password grant, both the application requesting a service and the services are trusted and are owned by the same organization.
- OAuth2 tokens generated by the implicit grant are more vulnerable to attack and misuse because the tokens are made available to the browser. Any malicious JavaScript running in the browser can get access to the OAuth2 access token and call the services you retrieved the OAuth2 token for on your behalf and essentially impersonate you.
- The implicit grant type OAuth2 tokens should be short-lived (1-2 hours). Because the OAuth2 access token is stored in the browser, the OAuth2 spec (and Spring Cloud security) doesn't support the concept of a refresh token in which a token can be automatically renewed.

## B.5 How tokens are refreshed

When an OAuth2 access token is issued, it has a limited amount of time valid and will eventually expire. When the token expires, the calling application (and user) will need to re-authenticate with the OAuth2 service. However, in most of the OAuth2 grant flows, the OAuth2 server will issue both an access token and a refresh token. A client can present the refresh token to the OAuth2 authentication service, and the service will validate the refresh token and then issue a new OAuth2 access token. Let's look at figure B.5 and walk through the token refresh flow:



**Figure B.5 The token refresh flow allows an application to get a new access token without forcing the user to re-authenticate.**

1. The user has logged into Ostock and is already authenticated with the OAuth2 service. The user is happily working, but unfortunately, their token has expired.
2. The next time the user tries to call a service (say the organization service), the Ostock application will pass the expired token to the organization service.

3. The organization service will try to validate the token with the OAuth2 service, which returns an HTTP status code 401 (unauthorized) and a JSON payload, indicating that the token is no longer valid. The organization service will return an HTTP 401 status code to the calling service.
4. The Ostock application gets the 401 HTTP status code and the JSON payload, indicating that the call failed back from the organization service. The Ostock application will then call the OAuth2 authentication service with the refresh token. The OAuth2 authentication service will validate the refresh token and then send back a new access token.

## B.6 Summary

- OAuth2 is a flexible authorization framework that provides multiple mechanisms for applications to authenticate and authorize users without forcing them to share credentials.
- OAuth2 has four forms of authentication grants that client applications can use to authenticate users, receive an access token, and then validate that token. These grants are password, client credential, authorization code and implicit.
- The password grant is used when both the application and the services explicitly trust one another.
- The client credentials grant is typically used when an application needs to access an OAuth2 protected resource, but no human being is involved in the transaction. With the client credentials grant type, the OAuth2 server only authenticates based on the application name and the secret key provided by the resource owner.

- The authorization code grant allows different applications from different vendors to share data and services without having to expose a user's credentials across multiple applications.
- The implicit grant is a way a pure JavaScript application gets an access token without having any intermediate code exchange. In the other flows, the client communicates with an application server that's carrying out the user's requests, and the application server is interacting with any downstream services.

# Appendix C. Monitoring your microservices

This chapter covers

- Monitoring microservice performance
- Exposing application metrics with micrometer
- Storing application metrics with Prometheus
- Visualizing application metrics with Grafana Dashboards

Because microservices are distributed and fine-grained (small), they introduce a level of complexity to our application that wouldn't exist in monolithic applications. Microservice architectures require a high degree of operational maturity, and monitoring becomes a critical part of their administration. If we research about service monitoring, we will see that most of the people agree that it is a fundamental process, but what exactly is monitoring? I will define monitoring as the process of analysis, collection, and storage of data such as application, platform, and system events metrics, among others, that help us visualize failure patterns within an IT environment.

Only failures? It is essential to clarify that failure is one of the most apparent reasons why monitoring is fundamental, but it isn't the only one. The performance of the microservice also is considered another reason and a critical role in the application. We cannot describe the performance as a binary

concept that is just "up and running" or "down." Service architectures can operate with a specific state of degradation that may affect the performance of one or more services.

In the following sections, I will show you how to monitor our Spring Boot microservices using several technologies such as Spring Boot Actuator, Micrometer, Prometheus, and Grafana. So, let's start.

## **C.1 Introduction to monitoring with Spring Boot Actuator**

Spring Boot Actuator is a library that provides us with monitoring and administration tools for our REST API in a reasonably simple way by organizing and exposing a series of REST endpoints that allow us to access different monitoring information to check the status of our services. In other words, Spring Boot Actuator provides out-of-the-box operational endpoints that will help us understand and manage our service's health.

To use Spring Actuator, we need to follow two simple steps. The first one is to include the maven dependencies in our pom.xml file, and the second one is to enable the endpoints we are going to use in our application.

### ***ADDING THE SPRING BOOT ACTUATOR***

To include Spring Boot Actuator in our microservice, we need to add the following dependency to the pom.xml of the

microservice we are working on.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

## ***ENABLING ACTUATOR ENDPOINTS***

Setting up Spring Boot Actuator is a straightforward process; just by adding the dependency into our microservices, we now have a series of endpoints available to consume. Each endpoint can be enabled or disabled and exposed via HTTP or JMX. The following figure shows you all the actuator endpoints enabled by default.

To configure the enablement of a specific endpoint, we only need to use the following format property.

```
management.endpoint.<id>.enabled= true or false
```

For example, to disable the beans endpoint, we need the following property

```
management.endpoint.beans.enabled = false
```

For this book's purposes, I will be enabling all the default actuator endpoints for all our microservices, but feel free to make the changes that best fit your needs. Remember, you can always add security to the HTTP Endpoints by using Spring Security. The following code shows the spring boot

actuator configuration for our licensing and organization service.

```
management.endpoints.web.exposure.include=*
management.endpoints.enabled-by-default=true
```

With this configuration, if we hit the <http://localhost:8080/actuator>, we can see a list like the one shown in figure C.1 with all the endpoints exposed by Spring Boot Actuator.

```
{
 "_links": {
 "self": {↔},
 "archaius": {↔},
 "beans": {↔},
 "caches-cache": {↔},
 "caches": {↔},
 "health": {
 "href": "http://localhost:8080/actuator/health",
 "templated": false
 },
 "health-path": {↔},
 "info": {↔},
 "conditions": {↔},
 "shutdown": {↔},
 "configprops": {↔},
 "env": {↔},
 "env-toMatch": {↔},
 "integrationgraph": {↔},
 "loggers": {↔},
 "loggers-name": {↔},
 "heapdump": {↔},
 "threaddump": {↔},
 "metrics-requiredMetricName": {↔},
 "metrics": {↔},
 "scheduledtasks": {↔},
 "mappings": {↔},
 "refresh": {↔},
 "restart": {↔},
 "pause": {↔},
 "resume": {↔},
 "features": {↔},
 "service-registry": {↔},
 "bindings-name": {↔},
 "bindings": {↔},
 "channels": {↔},
 "hystrix.stream": {↔}
 }
}
```

## **Figure C.1 Spring Boot Actuator default endpoints.**

## **C.2 Setting up micrometer and Prometheus**

Spring Boot Actuator offers metrics that allow us to monitor our application. However, if we want to have more precise metrics for our application and want to obtain those metrics, we will need to use some additional technologies, such as Micrometer and Prometheus.

### **C.2.1 Understanding Micrometer and Prometheus**

Micrometer is a library that provides application metrics and is designed to add little or no overhead at all to the metrics collection activity in our applications.

Micrometer allows metric data to be exported to any of the most popular monitoring systems. Using Micrometer, the application abstracts from the metric system used and may change in the future if desired. One of the most popular monitoring systems is Prometheus, which is responsible for collecting and storing the data of the metrics exposed by the applications and offers a data query language with which other applications can visualize them in graphs and control panels. Grafana is one of these tools that allows you to view the data provided by Prometheus. Micrometer can also be used with other monitoring systems such as Datadog, SignalFx, Influx, New Relic, Google Stackdriver, Wavefront, and more.

One of the advantages of using Spring boot for our microservices is that Spring boot allows us to choose one or more monitoring systems containing different views to analyze and project our results. With Micrometer, we will be able to measure metrics that allow us to understand the performance of our systems as a whole through several components of a single application or instances of clusters, and more.

The following list contains some of the metrics we can obtain with Micrometer:

- Statistics related to garbage collection.
- CPU Usage.
- Memory Usage.
- Threads utilization
- Data source usage.
- Spring MVC request latencies.
- Kafka connection factories.
- Use of cache
- Number of events logged in Logback
- Uptime

For purposes of this book, I chose Prometheus as the monitoring system because it has a simple integration with Micrometer and Spring Boot 2. Prometheus is an in-memory dimensional time-series database, and also monitoring and alert system. Remember, when we talk about storing time series, we refer to storing data in chronological order, measuring variables over time, and time-series focused databases are exceptionally efficient in storing and querying this data. Prometheus's main objective is to work in a pull model scraping the metrics from the application instances periodically.

Some of the main characteristics of Prometheus are:

- Flexible Query Language: It contains a custom query language that allows us to query data straightforwardly.
- Efficient Storage: Stores the time series in memory and on a local disk efficiently.
- Multidimensional data models: All the time series are identified by metric name and a set of key-value pairs.
- Multiple Integrations: It allows integration with third parties such as Docker, JMX, and more.

## C.2.2 Implementing Micrometer and Prometheus

To export the data to Prometheus using Spring Boot 2, we just need to add the following dependencies.

```
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-core</artifactId>
</dependency>
```

For this book, I enabled all of the default actuator endpoints for all our microservices, but if you want only to expose the Prometheus endpoint, you should include the following property to the licensing and organization service application properties.

```
management.endpoints.web.exposure.include=prometheus
```

Just by executing those steps, you should now be able to see a list like the one shown in figure C.2 by hitting the <http://localhost:8080/actuator> URL.

```
{
 "_links": {
 "self": {↳},
 "archaius": {↳},
 "beans": {↳},
 "caches": {↳},
 "caches-cache": {↳},
 "health": {
 "href": "http://localhost:8080/actuator/health",
 "templated": false
 },
 "health-path": {↳},
 "info": {↳},
 "conditions": {↳},
 "shutdown": {↳},
 "configprops": {↳},
 "env-toMatch": {↳},
 "env": {↳},
 "loggers": {↳},
 "loggers-name": {↳},
 "heapdump": {↳},
 "threaddump": {↳},
 "prometheus": {
 "href": "http://localhost:8080/actuator/prometheus",
 "templated": false
 },
 },
}
```

## Figure C.2 Spring Boot Actuator endpoints including the actuator/Prometheus endpoint.

There are several ways to set up Prometheus. In this book, I will be using a Docker container to run the Prometheus

services using the official images that are ready to use. But feel free to use the one that best suits your needs. If you want to use Docker, make sure you have the service defined in the Docker compose file, as shown in the following code listing C.1.

### **Listing C.1 Setting up Prometheus in the docker-compose file**

```
prometheus:
 image: prom/prometheus:latest
 ports:
 - "9090:9090"
 volumes:
 - ./prometheus.yml:/etc/prometheus/prometheus.yml
 container_name: prometheus
 networks:
 backend:
 aliases:
 - "prometheus"
```

For the Prometheus container, I've created a volume for the Prometheus configuration file called `prometheus.yml`. This file contains the endpoints that Prometheus should use to pull the data. The following code listing C.2 shows the content of this file.

### **Listing C.2 Setting up the `prometheus.yml` file.**

```
global:
 scrape_interval: 5s #A
 evaluation_interval: 5s #B
 scrape_configs:
 - job_name: 'licensingservice'
 metrics_path: '/actuator/prometheus' #C
 static_configs:
 - targets: ['licensingservice:8080'] #D
 - job_name: 'organizationservice'
 metrics_path: '/actuator/prometheus'
 static_configs:
 - targets: ['organizationservice:8081'] #E
```

```

#A Sets the scrape interval to every 5 seconds
#B Sets the evaluate rules time to every 5 seconds
#C URL for the actuator/Prometheus endpoint in charge of exposing the metrics
 information in the format Prometheus expects to collect it.
#D URL for the licensing service.
#E URL for the organization

```

With the `prometheus.yml` file defines all the configuration the Prometheus service is going to have, for example, with the previous values, the service will scrape all the /Prometheus endpoints every 5 seconds and add the data to its time-series database.

Now that we have all the configuration set up let's run our services and verify that the scraping was successful. To confirm this visit, the following URL <http://localhost:9090/targets> by doing you should see a similar page to the one shown in figure C.3.

The screenshot shows the Prometheus Targets page. At the top, there are navigation links: Prometheus, Alerts, Graph, Status ▾, and Help. Below this, a section titled "Targets" has a "All" tab selected and an "Unhealthy" tab. Under "licensingservice (1/1 up)", there is a "show less" button. A table displays the following data:

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://licensingservice:8080/actuator/prometheus">http://licensingservice:8080/actuator/prometheus</a>	UP	instance="licensingservice:8080" job="licensingservice"	1.333s ago	16.77ms	

Under "organizationservice (1/1 up)", there is a "show less" button. Another table displays the following data:

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://organizationservice:8081/actuator/prometheus">http://organizationservice:8081/actuator/prometheus</a>	UP	instance="organizationservice:8081" job="organizationservice"	3.493s ago	14.17ms	

**Figure C.3 Prometheus targets configured in the `prometheus.yml` file.**

## C.3 Configuring Grafana

Grafana is an open-source tool that displays time series data. It offers a set of dashboards that allows us to visualize, explore, alert, and understand the application data. Grafana also enables us to create, explore, and share dashboards. The main reason I chose Grafana in this book is that it is one of the best options to display dashboards. It contains incredible graphics, multiple functions, very flexible, and most important is easy to use.

There are several ways to set up Grafana as well. In this book, I will be using the official Grafana Docker image to run the service. But feel free to use the one that best suits your needs. If you want to use Docker, make sure you have the service defined in the Docker compose file, as shown in the following code listing C.3.

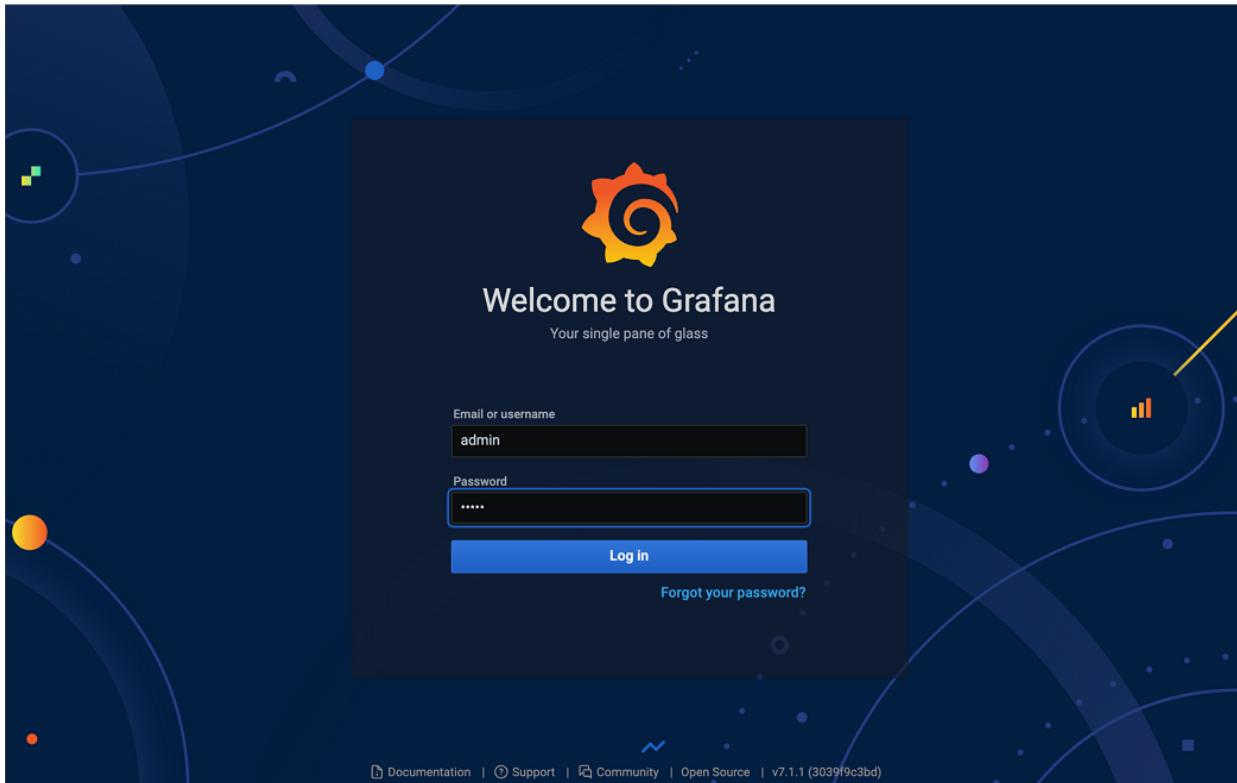
### **Listing C.3 Configure Grafana in the docker-compose file**

```
grafana:
image: "grafana/grafana:latest"
ports:
- "3000:3000"
container_name: grafana
networks:
backend:
aliases:
- "grafana"
```

Once added the configuration to the docker-compose file, execute the following command to run your services.

```
docker-compose -f docker/docker-compose.yml up
```

When it finishes, Grafana should be up and running on port 3000, so visiting the following link <http://localhost:3000/login>; we should be able to see the page shown in figure C.4.



**Figure C.4 Grafana log in page. The default username and password are admin admin.**

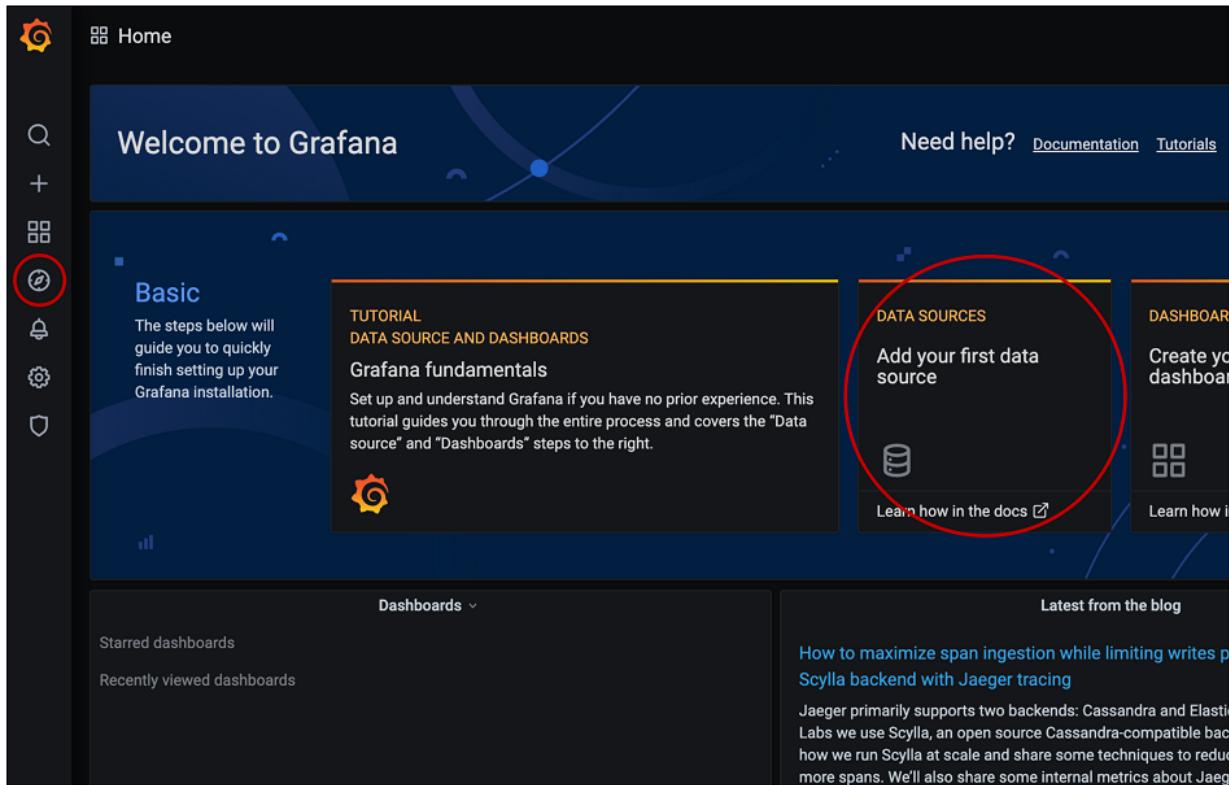
The default username and password for Grafana are admin and admin. We can change this once we logged in, or we can also define a username and password on the docker-compose file. The following code listing C.4 shows you how.

## **Listing C.4 Configure the admin username and password for Grafana in the docker-compose file**

```
grafana:
 image: "grafana/grafana:latest"
 ports:
 - "3000:3000"
 environment:
 - GF_SECURITY_ADMIN_USER=admin #A
 - GF_SECURITY_ADMIN_PASSWORD=password #B
 container_name: grafana
 ... rest of the file removed for conciseness
```

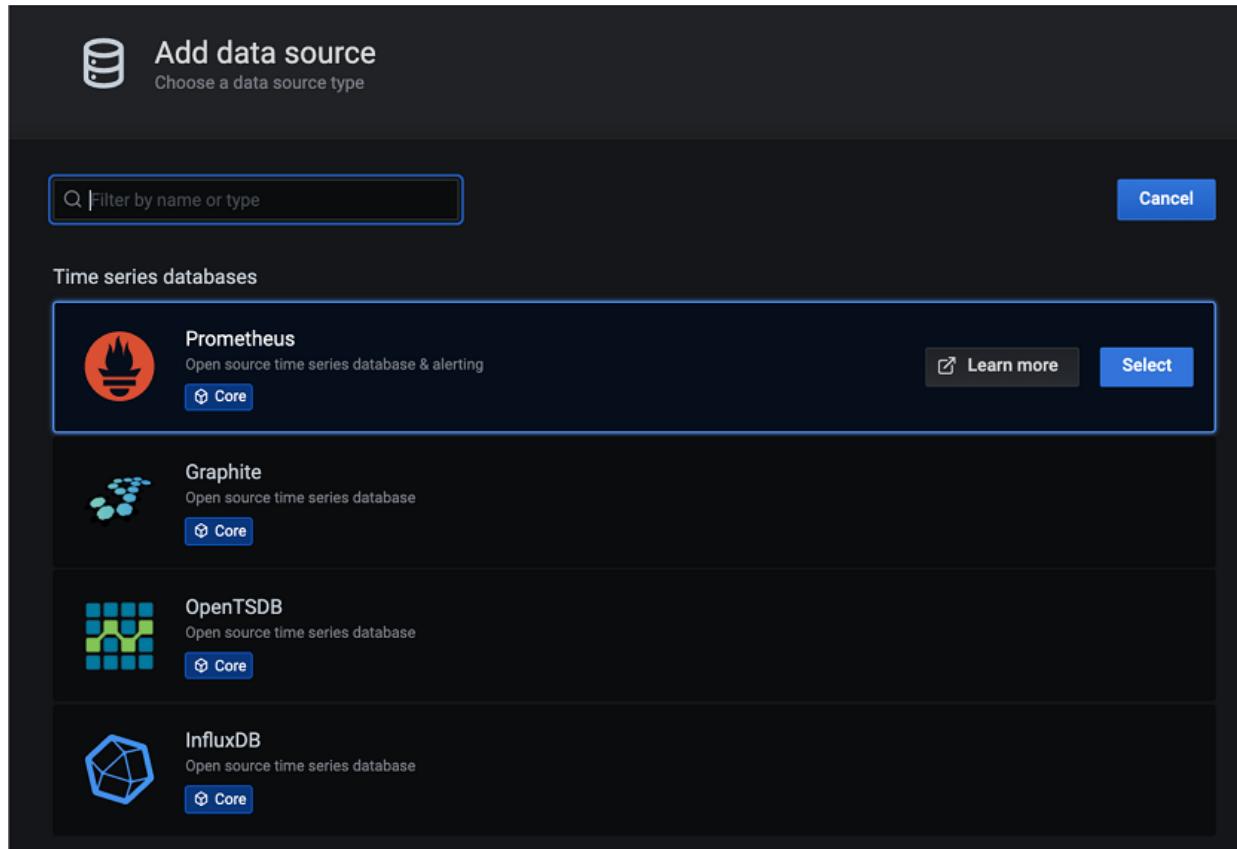
#A Sets the admin username  
#B Sets the admin password

To finish the configuration in Grafana, we need to create a data source and create a dashboard configuration. Let's start with the data source. To create the data source, click the data source section on the main page, click the explore icon on the left, and select the add data source option. The following figure C.5 shows you these options.



**Figure C.5 Grafana welcome page. This page shows the links to set the initial configuration.**

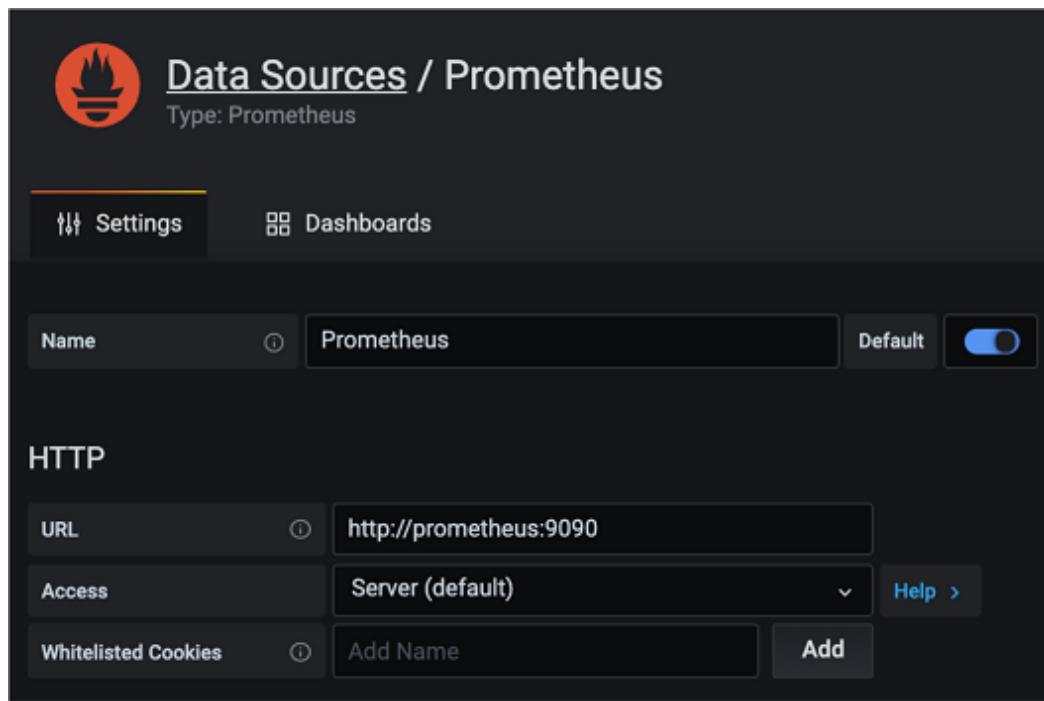
When you are at the data source page, select Prometheus as the time series database as shown in figure C.6



**Figure C.6 Select the time series database in the Add data source page.**

The final step is to configure the Prometheus URL <http://localhost:9090> or <http://prometheus:9090> for our data source. The following figure C.7 shows you how.

**NOTE** Remember, we use localhost when we are running the services locally, but if we are running the services with docker, we need to use the Prometheus service backend alias we defined on the docker-compose file. For the Prometheus service, we defined the prometheus alias. You can see this on the previous code listing C.1.



## Figure C.7 Configure the Prometheus datasource using the local or the docker Prometheus URL.

Once filled, click the Save and test button at the bottom of the page. Now that we have our data source let's import a dashboard for our Grafana.

In order to import a dashboard, click the Dashboard icon on the left menu, select the manage option, and click the import button.

On the import page, you will see the following options:

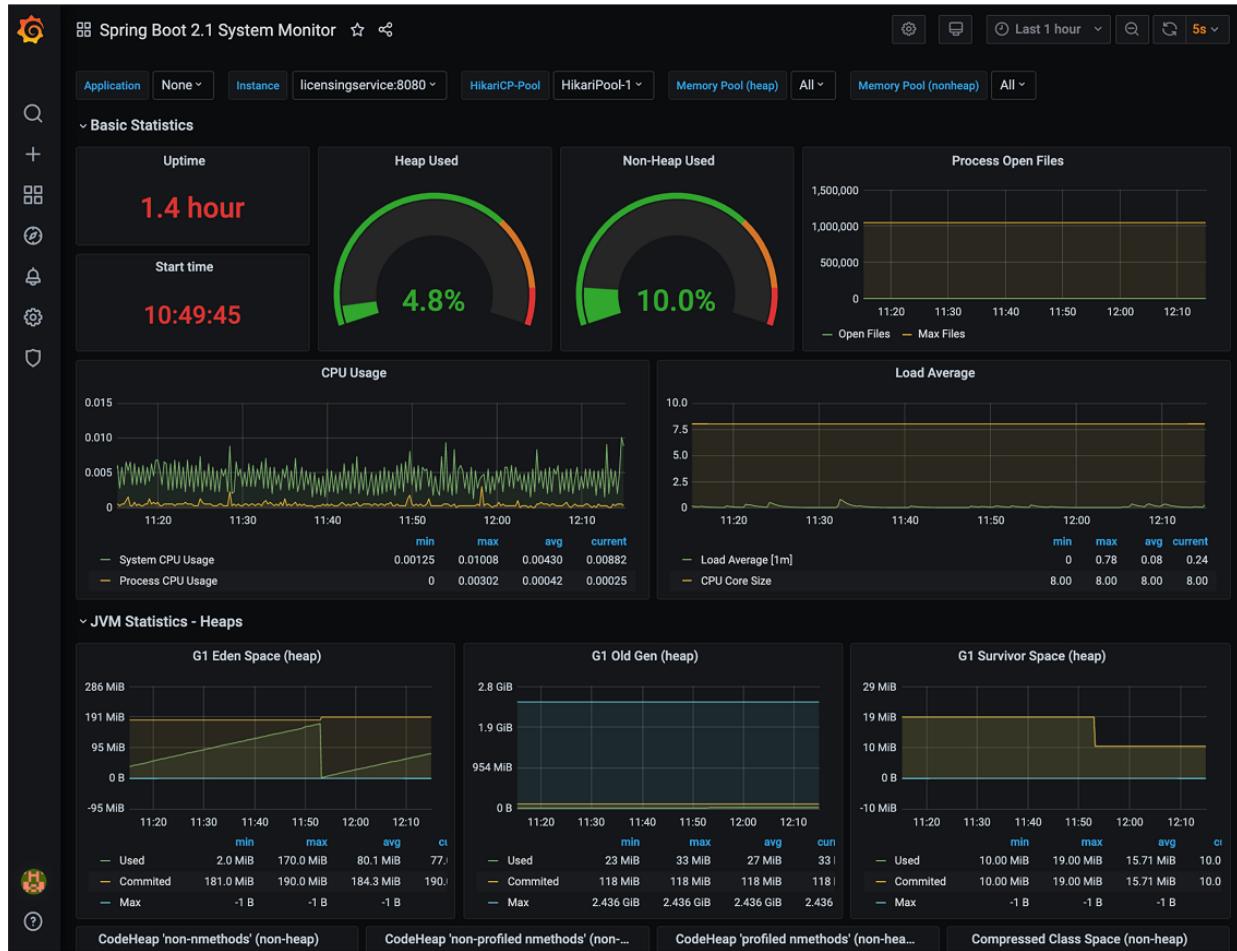
- Upload a JSON file.
- Import via grafana.com
- Import via panel JSON.

For purposes of this example, we will import the following dashboard from grafana.com.

<https://grafana.com/grafana/dashboards/11378/>. This dashboard contains the Spring Boot 2.1 statistics by micrometer-prometheus.

To import this dashboard just copy the URL or the ID to the clipboard, paste it in the Import via Grafana.com field and click the load button. This load button will redirect you to the dashboard configuration page where you can rename, move to a folder, and select the Prometheus data source. To continue, select the Prometheus data source and click the import button.

If the configuration is successful, you should now see your dashboard page with all the micrometer metrics. Figure C.8 shows the Spring Boot 2.1 System Monitor dashboard.



**Figure C.8 Grafana Spring Boot 2.1 System Monitor dashboard.**

**NOTE** I recommend you visit the official documentation links if you want to know more about Grafana and Prometheus.

<https://grafana.com/docs/grafana/latest/> and  
<https://prometheus.io/docs/introduction/overview/>

## C.4 Summary

The microservices architecture needs to be monitored for the same reasons as any other type of distributed system. The more complex our architecture gets, the more challenging it

is to understand the performance and troubleshoot the issues.

When we talk about monitoring an application, we might think of failure, and yes, failure is one of the most common reasons why adequate monitoring is essential. Still, it is not the only reason. Performance is another excellent reason for considering monitoring in our architecture. As I mentioned in the first's chapters, services do not only are up or down. Instead, they can be up and running but with degraded states that can damage our architecture.

With a reliable monitoring system like the one I explained in this appendix, you will prevent performance failures and visualize possible errors in your architecture. It is essential to highlight that you can add this monitoring code to any of your application development stages. The only requirements you need to execute this code are Docker and a Spring Boot application.