

Back-end Java

Microserviços, Spring Boot e Kubernetes



Sumário

- ISBN
- Agradecimentos
- Sobre o livro
- 1. Introdução
- 2. Instalando o ambiente
- 3. Criando os primeiros serviços
- 4. Serviço de usuários (user-api)
- 5. Serviço de produtos (product-api)
- 6. Serviço de compras (shopping-api)
- 7. Buscas mais complexas na shopping-api
- 8. Comunicação entre os serviços
- 9. Exceções
- 10. Autenticação
- 11. Testes de unidade
- 12. Api-gateway
- 13. Executando a aplicação com Docker
- 14. Kubernetes
- 15. Instalando o Kubernetes
- 16. Implantando as aplicações no Kubernetes
- 17. Acesso externo ao cluster

ISBN

Impresso: 978-65-86110-61-6

Digital: 978-65-86110-62-3

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

Agradecimentos

Agradeço à minha família — sem eles, eu dificilmente teria chegado até aqui — e agradeço à minha namorada, Brianda, pelo companheirismo durante toda a minha carreira profissional.

Obrigado ao pessoal da editora Casa do Código, em especial à Vivian, pela grande ajuda na revisão do livro.

Sobre o autor

Eduardo Felipe Zambom Santana tem mais de 15 anos de experiência em Engenharia de Software. Trabalha principalmente como Java, já tendo trabalhado com os principais frameworks da linguagem, como Struts, JSF e Spring. Também tem bastante experiência em outras linguagens, como Python e Erlang. Formou-se em Ciência da Computação na UFSCar (2007), fez mestrado também na UFSCar (2010) e doutorado na USP (2019), trabalhando na área de sistemas distribuídos.

Sobre o livro

A arquitetura dos sistemas de software vem sofrendo diversas revoluções nos últimos anos, começando com os grandes monólitos, que dificultavam bastante a manutenção e a evolução das aplicações, passando pela arquitetura orientada a serviços, que era bastante dependente de arquivos XML de configuração, até chegar à arquitetura de microsserviços, que tenta resolver (ou minimizar) alguns dos vários problemas que as arquiteturas de software anteriores possuíam.

Diversas tecnologias têm surgido para o desenvolvimento de aplicações baseadas em microsserviços para a plataforma Java e, sem dúvida, o framework mais conhecido e utilizado atualmente para esse fim é o Spring Boot. O objetivo do Spring Boot é diminuir a quantidade de configurações necessárias para o desenvolvimento de aplicações. Nele, é possível utilizar diversos frameworks e bibliotecas para a construção rápida de microsserviços, com diferentes funcionalidades, como desenvolvimento de APIs REST, comunicação entre serviços e segurança.

Porém, na arquitetura de microsserviços, é mais complicado executar uma aplicação inteira, tanto em um ambiente local quanto em um ambiente de produção, pois é necessário executar e configurar todos os serviços e permitir que eles se comuniquem. Existem diversas ferramentas que facilitam essas tarefas, sendo as duas principais o Docker, para a criação de contêineres, e o Kubernetes, para a criação de clusters que executam esses contêineres Docker.

Atualmente, é essencial que um desenvolvedor ou desenvolvedora back-end conheça, além da linguagem de programação que vai utilizar, algumas dessas ferramentas para a execução da aplicação em um ambiente de produção. Embora seja possível desenvolver e executar os microsserviços separadamente em sua máquina, isso não é o ideal, pois não se terá um ambiente próximo do real para testar a aplicação.

Este livro vai mostrar como desenvolver uma aplicação baseada em microsserviços utilizando o Spring Boot, como criar imagens Docker dos

serviços desenvolvidos e, por fim, como executar a aplicação no Kubernetes.

Para quem é este livro?

Este livro foi escrito principalmente para quem já tem conhecimento na linguagem Java e deseja começar a trabalhar com desenvolvimento back-end nessa linguagem com o framework Spring. Também poderá ser bastante útil para desenvolvedores web que estão começando a trabalhar com APIs e com a arquitetura de microsserviços. Para quem já trabalha com back-end, serão proveitosas a explicação sobre o Kubernetes e a demonstração de como configurar um cluster Kubernetes para o ambiente de desenvolvimento.

Estrutura do livro

O livro é dividido em duas partes. A primeira, do capítulo 2 ao 12, mostra o desenvolvimento de uma aplicação de microsserviços com o Spring Boot. A aplicação é formada por três microsserviços, chamados de *user-api*, *product-api* e *shopping-api*, que terão as responsabilidades de gerenciar usuários, produtos e compras. O capítulo 2 apresentará o ambiente de programação utilizado; os capítulos de 3 a 7 mostrarão a criação dos microsserviços; o capítulo 8 apresentará a comunicação entre os serviços; o capítulo 9 fará o tratamento dos erros nas aplicações; o capítulo 10 mostrará um mecanismo de autenticação nos serviços; o capítulo 11 mostrará como criar testes de unidade em aplicações Spring; e o capítulo 12 mostrará como criar um *api-gateway* com o Spring-Cloud.

A segunda parte do livro, que vai do capítulo 13 ao 17, mostra como criar o cluster Kubernetes na máquina de desenvolvimento. O capítulo 13 mostra como criar as imagens Docker com os microsserviços desenvolvidos; o capítulo 14 apresenta os principais conceitos do Kubernetes; o 15 mostra como instalar a ferramenta no ambiente local; finalmente, os capítulos 16 e 17 fazem as configurações finais para executar as aplicações no cluster.

Atualizações da nova versão do livro

Esta é a segunda versão deste livro. As principais modificações em relação à primeira versão são:

- Atualização do Spring Boot em todas as aplicações, da versão 2.3.0.RELEASE para a 3.0.0;
- Atualização do Java em todas as aplicações, da versão 8 para a 17;
- Atualização da versão do Kubernetes, da versão 1.16 para 1.22;
- Melhorias gerais em todas as aplicações;
- Utilização do Lombok nas aplicações;
- Adição dos testes de unidade no capítulo 11;
- Adição do projeto do *api-gateway* no capítulo 12.

Código-fonte

Todo o código-fonte das aplicações e os arquivos para a configuração do cluster Kubernetes estão disponíveis no GitHub, no repositório:

<https://github.com/ezambomsantana/java-back-end-livro>

CAPÍTULO 1

Introdução

Até alguns anos atrás, a grande maioria dos sistemas web era desenvolvida em uma arquitetura monolítica, isto é, tudo ficava em apenas um projeto, incluindo o back-end e o front-end. Alguns frameworks como o Struts e o JavaServer Faces inclusive disponibilizam diversas bibliotecas para a criação de interfaces ricas, como Prime Faces e Rich Faces. Esse modelo tinha uma série de problemas, como alto acoplamento do front-end e do back-end da aplicação, projetos enormes com milhares de arquivos (HTMLs, JavaScript, CSS, scripts de banco de dados, Java) e diversas cópias dos mesmos trechos de código em várias aplicações.

Atualmente, a maioria dos sistemas está sendo desenvolvida utilizando APIs e a arquiteturas de microsserviços, que busca resolver exatamente os problemas mencionados anteriormente. Com as APIs, o back-end é totalmente isolado do front-end e, com os microsserviços, os projetos são muito menores, não passando de algumas dezenas ou centenas de arquivos. Além disso, evita-se a duplicação de código, já que cada serviço implementa uma funcionalidade específica e pode ser reutilizado por diversas aplicações.

O Java continua sendo a linguagem mais utilizada para o desenvolvimento de aplicações back-end. Isso se deve ao grande grau de maturidade da linguagem e da sua máquina virtual. Existem diversas formas de desenvolver microsserviços em Java, como utilizar bibliotecas nativas ou alguns frameworks, como o Quarkus e, principalmente, o Spring. Neste livro, utilizaremos o Spring Boot, com diversas funcionalidades do Spring Web, Spring Data e o Spring Cloud. Também mostraremos que essa API pode ser integrada com um front-end desenvolvido em JavaScript com o framework React.

Obviamente, essa arquitetura também possui algumas desvantagens, como maior complexidade das aplicações e a latência para a comunicação entre os serviços. Testar as aplicações é uma tarefa mais complexa porque um

microsserviço pode depender de vários outros. Por exemplo, um microsserviço de compras pode depender dos dados sobre o cliente e sobre os produtos que estarão em diferentes microsserviços. É possível testar os serviços localmente, mas, para isso, a pessoa desenvolvedora deve executar localmente todos os serviços.

Uma maneira mais simples de testar as aplicações é criar um cluster local utilizando o Kubernetes, já que o cluster pode ficar executando em background e apenas o microsserviço alterado necessita ser atualizado. Além de facilitar os testes, utilizar o Kubernetes no ambiente de desenvolvimento aumenta a confiabilidade da aplicação, já que o ambiente do programador é muito mais próximo dos ambientes de homologação e produção.

Para explicar todos esses conceitos, este livro começa apresentando o framework Spring e desenvolvendo um exemplo de uma aplicação com três microsserviços. A aplicação consiste em um serviço para cadastro de cliente, um para cadastro de produtos e um para compras. Para a execução das compras, os clientes e os produtos devem ser validados, o que requer a comunicação entre os serviços. Depois, serão mostradas algumas funcionalidades do Spring Cloud, como o *api-gateway* e o ZooKeeper.

Depois que a aplicação estiver pronta, veremos como criar um cluster no ambiente de desenvolvimento utilizando o Docker, docker-compose e o Kubernetes.

1.1 Framework Spring

O Spring é um framework Java que possui uma grande quantidade de projetos, como o Spring Boot, o Spring Data e o Spring Cloud, que podem ser utilizados em conjunto ou não. É também possível utilizar outros frameworks com o Spring e até mesmo as bibliotecas do Enterprise Java Beans (EJB). O Spring é bastante antigo — sua primeira versão foi publicada em 2002 — e é um projeto robusto e estável. Atualmente, o

framework está na versão 5.0, lançada em 2017. Os próximos tópicos descrevem brevemente os projetos que utilizaremos.

Spring Boot

O Spring Boot é uma forma de criar aplicações baseadas no framework Spring de forma simples e rápida. Nelas, já existe um contêiner web, que pode ser o Tomcat ou o Jetty, e a aplicação é executada com apenas um *run*, diferentemente de quando é necessário primeiro instalar e configurar um contêiner, gerar um arquivo WAR (*Web Application Resource*) e, por fim, implantá-lo no contêiner. O Spring Boot também facilita a configuração das aplicações através de arquivos *properties* ou diretamente no código, não sendo necessário o uso de arquivos XML. Há também uma grande quantidade de bibliotecas especialmente criadas para ele, como para acesso a dados em banco de dados relacionais e NoSQL, para geração de métricas sobre a aplicação, acesso a serviços e muito mais.

A primeira versão do livro utilizou a versão 2.3.0.RELEASE do Spring Boot. Nesta nova versão, todas as aplicações foram atualizadas para a versão 2.6.7, que foi lançada em abril de 2022.

CRIANDO UM PROJETO SPRING BOOT

O Spring Boot possui um site interessante para criar um projeto, o Spring Initializr. Nele, é possível selecionar diversas opções e bibliotecas para a geração do projeto.

Acesse: <https://start.spring.io/>.

Spring Data

O Spring Data é um projeto do Spring para facilitar a criação da camada de persistência de dados. Esse projeto tem abstrações para diferentes modelos de dados, como banco de dados relacionais e não relacionais, como o MongoDB e o Redis.

Relacionado a banco de dados relacionais, o Spring Data possibilita o acesso aos dados utilizando interfaces e definindo apenas o nome de um método. O framework, com isso, implementa todo o acesso ao banco de dados automaticamente. Por exemplo, a classe na listagem a seguir define três métodos que serão traduzidos para consultas em um banco de dados relacional:

```
List<Pessoa> findByName(String name);  
List<Pessoa> findByAge(int age);  
Pessoa findByCpf(String cpf);
```

Nesse exemplo, para uma classe `Pessoa`, o Spring Data automaticamente gera um método que fará a busca — no primeiro caso, pelo nome; no segundo, pela idade e, no terceiro, pelo CPF. Obviamente, existe uma sintaxe correta para que o Spring Boot consiga gerar os métodos. Isso será apresentado detalhadamente a partir do capítulo 4 deste livro.

Spring Cloud

O Spring Cloud é um projeto que disponibiliza diversas funcionalidades para a construção de sistemas distribuídos, como gerenciamento de configuração, serviços de descoberta, proxys, eleição de líderes etc. É bastante simples usá-lo junto a projetos Spring Boot.

Neste livro, vamos usá-lo para a criação de um *api-gateway* que será uma aplicação que fica na frente de todos os microsserviços recebendo todas as requisições da aplicação. Com isso, podemos centralizar essas requisições, não sendo necessário que um cliente dos serviços conheça o endereço e a porta onde estarão instaladas todas as aplicações.

1.2 Kubernetes

O Kubernetes, ou K8s, é uma ferramenta para facilitar a execução e a operação de contêineres inicialmente desenvolvida por engenheiros do Google, mas hoje aberta para ser utilizada em diversas plataformas. Nela, é possível definir diversas opções para a implantação, execução e

disponibilização dos contêineres de forma fácil e, em muitos casos, automática.

Atualmente, o K8s é uma das principais ferramentas de DevOps, pois é utilizada normalmente em ambientes de produção e homologação das aplicações. Porém, também é possível a criação de um cluster local em uma máquina de desenvolvimento, o que facilita bastante os testes locais de uma aplicação e também aumenta a confiabilidade de um novo desenvolvimento, já que o ambiente de programação é muito mais próximo do ambiente de produção.

Vamos detalhar os principais conceitos do Kubernetes no capítulo 12, mas a estrutura básica dessa ferramenta é criar definições para os contêineres que serão executados com algumas configurações básicas, como número de CPUs e quantidade de memória necessária para executar esse contêiner. A partir deles, é possível criar máquinas virtuais (chamadas Pods) que os executam. Assim, o K8s disponibiliza diversas funcionalidades para facilitar e otimizar a execução dos Pods, como aproveitar melhor o hardware da máquina alocando e desalocando recursos automaticamente, monitorar aplicações, escalar rapidamente e automaticamente serviços de acordo com o uso de hardware e garantir a autorrecuperação das aplicações de forma rápida e simples.

1.3 Lombok

O Lombok é uma biblioteca Java para a geração de trechos de código que normalmente são iguais para todas as classes, como os construtores e os métodos `get` e `set`. Ela não adiciona nenhuma nova funcionalidade, mas ela aumenta bastante a produtividade dos programadores e facilita a manutenção das aplicações. Veremos que apenas adicionando algumas anotações, como `@Getter` e `@Setter`, o Lombok gera uma grande quantidade de código.

CAPÍTULO 2

Instalando o ambiente

Nesta primeira parte do livro, será necessário instalar a IDE para o desenvolvimento da aplicação. Eu optei pelo IntelliJ, porém qualquer outra IDE, como o Eclipse, o VisualStudio Code ou o NetBeans, pode ser utilizada. Também usaremos o banco de dados PostgreSQL e o PGAdmin para a administração do banco de dados.

Para facilitar a instalação do PostgreSQL, será utilizado um contêiner Docker com o BD já instalado e configurado, então, para isso, também será necessário instalar o Docker. Para testar os serviços, pode ser usada qualquer ferramenta para fazer requisições REST. Neste livro, foi utilizada a versão gratuita do Postman (<https://www.getpostman.com/>). No GitHub do projeto (<https://github.com/ezambomsantana/java-back-end-livro>), está disponível uma coleção com todas as requisições que serão feitas para testar a aplicação desenvolvida.

IntelliJ

Para instalar o IntelliJ, basta baixar o instalador da IDE no site oficial (<https://www.jetbrains.com/idea/>) e seguir as instruções. A IDE é independente de plataforma e tem um processo de instalação bastante similar no Linux, Windows e MacOS. O IntelliJ possui uma licença gratuita que é suficiente para o que vamos desenvolver neste livro. Se você quiser, pode também adquirir uma licença paga da IDE.

Maven

Utilizaremos o Maven para a gerência de dependências e também para a construção das imagens do Docker. Instalar o Maven é simples nos três SOs. No Linux, basta executar o comando `sudo apt install maven` e, no MAC, `brew install maven`. No Windows, é necessário baixar no site oficial da ferramenta a última versão e descompactar o arquivo, o que criará

uma pasta com os arquivos do Maven. Depois, basta adicionar na variável `PATH` do SO o caminho para a pasta do Maven.

Também é possível utilizar o Maven diretamente da IDE. Tanto o IntelliJ como o Eclipse já possuem um Maven embutido, que é suficiente para a construção e execução dos projetos que serão desenvolvidos neste livro. Outra opção é a utilização do Maven Wrapper, que é um executável que pode ser colocado na raiz do projeto e que evita que o Maven tenha que ser instalado na máquina. Mais informações sobre o Maven Wrapper podem ser encontradas em <https://www.baeldung.com/maven-wrapper>.

Docker

O Docker é uma ferramenta para criar e executar contêineres. Neste livro, o Docker será usado para criar os contêineres de cada um dos microsserviços e também para executar o banco de dados PostgreSQL no ambiente de desenvolvimento.

No Linux, a instalação é um pouco mais complexa e deve ser feita via linha de comando; já no Windows e no Mac, existe a versão *Docker for Desktop*.

Docker no Linux

Para instalar o Docker no Linux, o caminho mais simples é utilizar um gerenciador de dependências. No desenvolvimento deste livro, utilizei Ubuntu, por isso usei o *apt*, mas qualquer outro gerenciador, como o *dnf* no Fedora, possui passos parecidos. O primeiro passo para a instalação é atualizar os pacotes do *apt*.

```
sudo apt update
```

O segundo passo é remover pacotes antigos do Docker, caso algum já esteja instalado. Se nenhum estiver instalado, esse passo não fará nenhuma alteração na máquina.

```
sudo apt remove docker docker-engine docker.io
```

O terceiro passo é instalar o Docker na máquina.

```
sudo apt install docker.io
```

Finalmente, o comando `systemctl` adiciona o Docker como um serviço do SO e faz com que ele seja iniciado sempre que a máquina for inicializada.

```
sudo systemctl start docker  
sudo systemctl enable docker
```

Para verificar se o Docker foi instalado corretamente, basta executar o comando `docker version`. O resultado será como esse:

```
Client: Docker Engine - Community  
Version:          20.10.12
```

Docker no Windows e no Mac

Para o Windows e o Mac, existe uma ferramenta que facilita bastante a instalação do Docker (e também do Kubernetes, como será visto mais à frente), que é o *Docker for Desktop*. Em ambos os sistemas operacionais, a instalação é bem simples, bastando entrar no site da ferramenta (<https://www.docker.com/products/docker-desktop/>), baixar os instaladores e seguir os passos que serão indicados.

Essa ferramenta já instala o Docker e permite a criação e execução de contêineres Docker. Inicialmente, basta instalar a ferramenta que o Docker já estará funcionando. Quando começarmos a falar do Kubernetes, veremos também que essa ferramenta já possui uma versão do Kubernetes embutida, aí veremos mais configurações e opções.

Assim como o IntelliJ, o Docker for Desktop possui uma licença gratuita que é suficiente para o que utilizaremos neste livro.

PostgreSQL e PGAdmin

O banco de dados utilizado neste livro é o PostgreSQL. Para instalá-lo, existem duas opções: ou baixar a versão mais recente do BD no site oficial (<https://www.postgresql.org/>) ou utilizar uma imagem Docker com o banco de dados já instalado. As duas opções funcionam da mesma forma para a aplicação que será desenvolvida neste livro. Eu vou utilizar a imagem do

Docker para evitar a instalação do banco de dados na máquina. Com o Docker, para criar um contêiner que execute o PostgreSQL, basta executar o seguinte comando:

```
docker run -d -p 5432:5432 -e POSTGRES_PASSWORD=postgres postgres
```

Esse comando cria um contêiner usando a imagem *postgres*. Se ela não existe em sua máquina, não há problema, o Docker baixará a imagem diretamente do DockerHub (<https://hub.docker.com/>) e a instalará em seu registro Docker local. A opção `-p` faz o mapeamento da porta local da máquina para a porta do contêiner, o que permitirá que o PostgreSQL seja acessado no endereço <http://localhost:5432>.

Para o gerenciamento do banco de dados, o PGAdmin é uma ferramenta bastante útil. Ela pode se conectar ao PostgreSQL, sendo o banco de dados instalado diretamente na máquina ou em um contêiner Docker. Essa ferramenta também está disponível para todos os sistemas operacionais e pode ser baixada de seu site oficial, <https://www.pgadmin.org/>. Depois da instalação do PostgreSQL e do PGAdmin, para executar as aplicações que serão desenvolvidas no livro, é necessário apenas criar um novo banco de dados chamado `dev`.

Com tudo isso instalado e configurado, podemos iniciar o desenvolvimento da aplicação.

CAPÍTULO 3

Criando os primeiros serviços

No decorrer do livro, criaremos uma aplicação que simula um e-commerce com três microsserviços: um para o cadastro de usuários (*user-api*), um para o cadastro de produtos (*product-api*) e, finalmente, um serviço de compras (*shopping-api*). Inicialmente eles funcionarão de forma independente e, depois, trabalharemos na comunicação entre os microsserviços.

A *user-api* será responsável por manter os dados dos usuários da aplicação. Alguns serviços que estarão disponíveis nela serão a criação e exclusão de usuários e a validação da existência de um usuário. A *product-api* conterá todos os produtos cadastrados em nossa aplicação e terá serviços como cadastrar produtos e recuperar informações de um produto, como preço e descrição. Finalmente, a *shopping-api* será utilizada para o cadastro de compras na aplicação. Assim, para realizar uma compra, a *shopping-api* receberá informações sobre o usuário que está fazendo a compra e uma lista de produtos, e todos esses dados precisarão ser validados na *user-api* e na *product-api*.

O desenvolvimento dos três microsserviços terá o mesmo padrão, que são as camadas *Controller*, *Service* e *Repository*. Além disso, teremos as entidades que representam as tabelas dos banco de dados e os *Data Transfer Objects (DTO)*, que são as classes utilizadas para receber e enviar informações entre os microsserviços e também para o front-end. Todas essas camadas serão explicadas no decorrer deste e dos próximos capítulos.

Neste capítulo, para apresentar os conceitos básicos do Spring Boot, será desenvolvida uma versão bem simplificada da *user-api*. Nesta versão, os dados estarão armazenados apenas em memória em uma lista de objetos. Porém, ao fim, já teremos visto boa parte dos conceitos necessários para o desenvolvimento de uma API REST com o Spring Boot.

3.1 Hello World com o Spring Boot

O primeiro passo para criar uma aplicação Spring Boot é configurar o projeto com suas dependências. Utilizaremos o Maven para gerenciá-las. Duas configurações são importantes nesta fase do projeto: primeiro, adicionar a versão do Spring Boot que será utilizada na tag `<parent>`, como é possível ver na listagem a seguir. Nesta edição do livro, foi utilizada a versão 3.0.0 do Spring Boot.

Depois, é necessário colocar a dependência `spring-boot-starter-web`, que configura o projeto para ser uma aplicação Web (com isso, o Spring Boot cria uma aplicação web simples com o servidor Tomcat já configurado), e, também, a dependência `spring-boot-starter-validation`, que será utilizada para fazer algumas validações básicas em algumas rotas (por exemplo, para verificar se todos os campos obrigatórios foram preenchidos em uma rota de cadastro).

Outra biblioteca que será utilizada nos projetos que serão desenvolvidos neste livro é o *Lombok*, para a geração de códigos repetitivos nos projetos Java, como os métodos `get`, `set` e os construtores. Ele funciona com o uso de anotações nas classes, que indicam quais métodos devem ser gerados. As anotações serão explicadas conforme forem aparecendo no código.

```
<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
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.santana.java.back.end</groupId>
  <artifactId>user-api</artifactId>
  <version>0.0.1</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.0</version>
```

```

</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.24</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

</project>

```

Em toda aplicação Spring Boot, também é necessário criar uma classe simples com o método `main`, que chama o método `run` da classe `SpringApplication`. Essa chamada configura uma aplicação Spring básica, criando todos os *beans* no projeto. Os beans são as classes com anotações especiais do Spring, como os `@RestController` e os `@Service`. É possível fazer configurações mais complexas, mas isso será visto nos próximos capítulos. Com essa configuração básica, rodando a aplicação, o Tomcat será iniciado, mas ainda nenhuma rota funcionará.

```

package com.santana.java.back.end;

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

@SpringBootApplication
public class App {

    public static void main(String[] args) {

```

```

        SpringApplication.run(App.class, args);
    }
}

```

Para criar o primeiro serviço, será criada uma classe com a anotação `@RestController`, que permite a criação de métodos que serão chamados via web utilizando o protocolo HTTP. Uma rota back-end é um caminho no servidor que recebe uma requisição HTTP de um usuário. Essa rota pode receber informações e retornar uma resposta.

Por exemplo, podemos criar um método simples que, quando chamado pelo browser, retornará uma mensagem para o usuário. Esse método deve ser anotado com `@GetMapping` e, como parâmetro, deve ser passado o caminho para acessar esse método. No exemplo seguinte, temos então um método chamado `getMensagem` que retornará uma mensagem simples para o usuário.

```

package com.santana.java.back.end.controller;

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

@RestController
public class UserController {

    @GetMapping("/")
    public String getMensagem() {
        return "Spring boot is working!";
    }

}

```

Porém, a maioria dos serviços retorna dados mais complexos, principalmente no formato *JSON (JavaScript Object Notation)*, que hoje é o principal padrão para a troca de mensagem entre aplicações. Esse formato é utilizado por praticamente todas as linguagens e serve tanto para a comunicação entre os serviços do back-end quanto para a integração do

back-end com o front-end. A tradução para esse formato é feita automaticamente pelo Spring, bastando que o método do Controller retorne um objeto Java. A listagem a seguir mostra a classe `UserDTO`, que será utilizada como exemplo para os próximos serviços.

Note também as anotações `@Getter`, `@Setter`, `@NoArgsConstructor` e `@AllArgsConstructor`. Essas anotações são do *Lombok* e indicam que devem ser gerados respectivamente os métodos `get`, os `set`, um construtor vazio e um construtor com todos os argumentos da classe. Com isso, não precisamos escrever nenhum desses métodos.

```
package com.santana.java.back.end.dto;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.LocalDateTime;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserDTO {

    private String nome;
    private String cpf;
    private String endereco;
    private String email;
    private String telefone;
    private LocalDateTime dataCadastro;

}
```

Essa classe é um DTO, que foi comentado anteriormente. Todas as classes que tiverem essa sigla no nome serão utilizadas como o retorno dos métodos da camada *Controller*. Elas possuem apenas os atributos da classe

que estamos criando, como nome, CPF e endereço dos usuários da aplicação, e os métodos *get* e *set* (que, nesse caso, foram gerados com o Lombok).

Antes de iniciar a implementação das rotas, a listagem a seguir mostra um método que cria e retorna uma lista com objetos do tipo `UserDTO`. Essa lista será utilizada em mais de um método, por isso é estática e, para inicializá-la apenas uma vez, foi criado um método chamado `initiateList` que insere três usuários na lista. Esse método foi anotado com `@PostConstruct`, que faz com que ele seja executado logo depois que o contêiner inicializa a classe `UserController`. Essa anotação pode ser utilizada em todas as classes gerenciadas pelo Spring, como `Controllers` e `Services`.

Além disso, note que foi adicionada a anotação `@RequestMapping("/user")` na classe. Essa anotação indica que todas as rotas que serão implementadas neste controlador possuirão o path iniciado com `/user`.

```
@RestController
@RequestMapping("/user")
public class UserController {

    public static List<UserDTO> usuarios = new ArrayList<UserDTO>
();

    @PostConstruct
    public void initiateList() {
        UserDTO userDTO = new UserDTO();
        userDTO.setNome("Eduardo");
        userDTO.setCpf("123");
        userDTO.setEndereco("Rua a");
        userDTO.setEmail("eduardo@email.com");
        userDTO.setTelefone("1234-3454");
        userDTO.setDataCadastro(LocalDate.now());

        UserDTO userDTO2 = new UserDTO();
        userDTO2.setNome("Luiz");
        userDTO2.setCpf("456");
        userDTO2.setEndereco("Rua b");
```

```

        userDT02.setEmail("luiz@email.com");
        userDT02.setTelefone("1234-3454");
        userDT02.setDataCadastro(LocalDateTime.now());

        UserDT0 userDT03 = new UserDT0();
        userDT03.setNome("Bruna");
        userDT03.setCpf("678");
        userDT03.setEndereco("Rua c");
        userDT03.setEmail("bruna@email.com");
        userDT03.setTelefone("1234-3454");
        userDT03.setDataCadastro(LocalDateTime.now());

        usuarios.add(userDT0);
        usuarios.add(userDT02);
        usuarios.add(userDT03);
    }
}

```

Agora, um método do controller pode retornar a lista de usuários que foi criada no método `initiateList`. Ele se chama `getUsers` e recebe a anotação `@GetMapping`, que indica que essa rota retornará dados do servidor. O endereço dessa rota será `http://localhost:8080/user`, pois o `/user` foi definido na classe com a anotação `@RequestMapping`.

```

@GetMapping
public List<UserDT0> getUsers() {
    return usuarios;
}

```

A resposta para a chamada desse serviço será o seguinte JSON:

```

[
  {
    "nome": "Eduardo",
    "cpf": "123",
    "endereco": "Rua a",
    "email": "eduardo@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  },

```

```

{
  "nome": "Luiz",
  "cpf": "456",
  "endereco": "Rua b",
  "email": "luiz@email.com",
  "telefone": "1234-3454",
  "dataCadastro": "2019-11-17T21:04:51.701+0000",
},
{
  "nome": "Bruna",
  "cpf": "678",
  "endereco": "Rua c",
  "email": "bruna@email.com",
  "telefone": "1234-3454",
  "dataCadastro": "2019-11-17T21:04:51.701+0000",
}
]

```

Note que o JSON tem exatamente os mesmos valores dos objetos da classe `UserDTO`. Com essa seção, as principais ideias de como desenvolver uma API com o Spring Boot já foram mostradas, que são a criação de um `Controller` e os métodos que retornam os dados no formato JSON. No restante deste capítulo, desenvolveremos os serviços para manipular a lista de usuários e, com isso, serão explicados os principais tipos de serviços e como eles devem ser chamados.

Para quem conhece o protocolo HTTP, deve ter percebido que a anotação `@GetMapping` tem o nome de um de seus verbos. Isso será utilizado em todos os `controllers`, já que o protocolo REST é uma extensão do HTTP e todos os métodos do serviço possuem uma chamada com um dos verbos do protocolo. O box a seguir contém uma pequena explicação sobre esse protocolo e seus principais verbos.

PROTOCOLO HTTP

O HTTP é o principal protocolo da Web. Ele define como deve ser feita uma requisição para uma aplicação, incluindo a URL, os parâmetros, os cabeçalhos e o tipo de resposta esperado. Os serviços que seguem o protocolo HTTP devem ser configurados com um verbo, que indica qual é o comportamento esperado do serviço. Os verbos HTTP mais utilizados são:

GET: os métodos GET devem recuperar dados, não afetando o estado da aplicação. Podem receber parâmetros, mas estes devem ser utilizados apenas para a recuperação de dados, nunca para uma atualização ou inserção.

POST: os métodos POST enviam dados para o servidor para serem processados. Os dados vão no corpo da requisição e não na URL. Normalmente são utilizados para criar novos recursos no servidor.

PATCH: funciona de modo similar ao POST, mas deve ser utilizado para atualizar as informações no servidor e não para novos registros.

DELETE: utilizado para excluir elementos do servidor.

3.2 Serviço de usuários (user-api)

Já temos um serviço que retorna uma lista de usuários, vamos criar agora um que recebe o identificador de um usuário e retorna apenas um usuário específico. Esse serviço também utiliza o verbo HTTP `GET`, com a diferença de que ele recebe um parâmetro na URL para filtrar o usuário a ser retornado. Para implementar esse método, a mesma lista de usuários foi utilizada, porém, agora, o retorno é apenas um dos usuários da lista ou uma exceção dizendo que o usuário não foi encontrado.

```
@GetMapping("/{cpf}")  
public UserDTO getUsersFiltro(@PathVariable String cpf) {
```

```

        return usuarios
            .stream()
            .filter(userDTO -> userDTO.getCpf().equals(cpf))
            .findFirst()
            .orElseThrow(() -> new RuntimeException("User not
found."))
    }

```

Para definir o parâmetro na URL, tivemos que colocar o valor {cpf} na definição da rota e também adicionar a anotação `@PathVariable` no parâmetro `cpf`. Note que tanto o valor na URL quanto o parâmetro possuem o mesmo nome — isso é obrigatório.

Esse serviço terá duas possíveis respostas: a primeira é caso o método encontre algum usuário com o CPF utilizado como filtro. Por exemplo, se a chamada para o serviço for `http://localhost:8080/user/123`, a resposta será:

```

{
  "nome": "Eduardo",
  "cpf": "123",
  "endereco": "Rua a",
  "email": "eduardo@email.com",
  "telefone": "1234-3454",
  "dataCadastro": "2019-11-17T21:04:51.701+0000",
}

```

A segunda é caso nenhum usuário seja encontrado. Por exemplo, para a chamada `http://localhost:8080/user/000`, a resposta será um erro indicando que nenhum usuário foi encontrado. A mensagem do erro não é a melhor possível, pois ela não indica claramente qual o problema. Veremos nos próximos capítulos como melhorar essa mensagem.

```

{
  "timestamp": "2022-09-26T12:31:32.640+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/user/000"
}

```

O próximo serviço salvará as informações de um novo usuário na lista de usuários. É possível utilizar um método `GET` para enviar dados para o servidor, porém isso não é recomendável. O correto é enviar os dados em uma requisição `POST` e no corpo da requisição, não na URL. Em um método `POST`, a anotação utilizada é a `@PostMapping`, também recebendo como parâmetro o caminho para a requisição. Para o método receber dados no corpo da requisição, é necessário utilizar a anotação `@RequestBody`. O seguinte método usa essas anotações para receber as informações de um usuário e inseri-lo na lista:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public UserDTO inserir(@RequestBody UserDTO userDTO) {
    userDTO.setDataCadastro(LocalDate.now());
    usuarios.add(userDTO);
    return userDTO;
}
```

Um outro fator importante desse método é o código de retorno. No REST, um dos dados que vão na resposta para o usuário é o código de retorno de uma rota. Por padrão, esse código é o `200 OK`, mas podemos mudar esse retorno dependendo do tipo de método. Normalmente um método `POST` retorna um código `201 CREATED`, que indica que um novo recurso foi criado no servidor. A definição do código pode ser feita usando a anotação `@ResponseStatus`. Para mais detalhes sobre os códigos HTTP, veja o quadro a seguir.

CÓDIGOS HTTP

Na resposta HTTP, um dado bastante importante é o código da resposta. Esse código indica resumidamente se a requisição foi tratada corretamente e o que aconteceu no servidor. Ela é muito utilizada para a comunicação entre serviços, para que o cliente saiba o que fazer depois de chamar o serviço. Os erros são divididos por faixa, que são:

- **200:** indica que a requisição foi tratada corretamente. Alguns dos códigos mais utilizados são o 200 OK , 201 CREATED , 202 ACCEPTED e 204 NO_CONTENT .
- **300:** indica um redirecionamento. Os códigos mais conhecidos nessa faixa são o 301 MOVED_PERMANENTLY e o 308 PERMANENT_REDIRECT .
- **400:** indica erros do cliente (por exemplo, o cliente enviou dados no formato errado). Alguns dos códigos mais utilizados são o 400 BAD_REQUEST , 401 UNAUTHORIZED , 404 NOT_FOUND e 408 TIMEOUT .
- **500:** indica que o servidor está com algum problema e não pode tratar a requisição. Os erros mais conhecidos nesta faixa são o 500 INTERNAL_SERVER_ERROR e 502 BAD_GATEWAY .

Para enviar os dados em uma requisição `post` , também será utilizado o formato JSON. Por exemplo, o seguinte JSON deve ser enviado no corpo da requisição para o cadastro de um novo usuário:

```
{
  "cpf": "987",
  "nome": "Carlos",
  "endereco": "Avenida 2",
  "email": "carlos@email.com",
  "telefone": "1234-3454"
}
```

Note que a data de cadastro não foi passada no JSON, pois ela é preenchida automaticamente com a data do servidor no método `inserir` . Se

chamarmos novamente o método que lista todos os usuários da lista, receberemos a seguinte resposta agora:

```
[
  {
    "nome": "Eduardo",
    "cpf": "123",
    "endereco": "Rua a",
    "email": "eduardo@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  },
  {
    "nome": "Luiz",
    "cpf": "456",
    "endereco": "Rua b",
    "email": "luiz@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  },
  {
    "nome": "Bruna",
    "cpf": "678",
    "endereco": "Rua c",
    "email": "bruna@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  },
  {
    "nome": "Carlos",
    "cpf": "987",
    "endereco": "Avenida 2",
    "email": "carlos@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  }
]
```

Nesse tipo de rota, outro requisito importante é fazer a validação dos dados de entrada. Por exemplo, podemos definir que o nome, o CPF e o e-mail são

dados obrigatórios para o cadastro de usuário, mas do jeito como a rota foi implementada, isso não está sendo validado.

O Spring possui um mecanismo bastante simples para fazer esse tipo de validação básica. Podemos fazer isso diretamente na classe `UserDTO` apenas adicionando a anotação `@NotBlank`.

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserDTO {

    @NotBlank(message = "Nome é obrigatório")
    private String nome;
    @NotBlank(message = "CPF é obrigatório")
    private String cpf;
    private String endereco;
    @NotBlank(message = "E-mail é obrigatório")
    private String email;
    private String telefone;
    private LocalDateTime dataCadastro;

}
```

Outra pequena mudança que deve ser feita para a validação funcionar é adicionar a anotação `@Valid` no parâmetro que recebe os dados na rota `POST`. Isso deve ser feito para indicar para o Spring que as validações que estão definidas no DTO devem ser realizadas antes da execução do método do controlador.

```
public UserDTO inserir(@RequestBody @Valid UserDTO userDTO) {
```

Caso algum dado esteja incorreto na chamada para a rota, o retorno será uma mensagem indicando que alguma coisa de errado aconteceu. Aqui, a resposta ainda não está muito boa também, pois ela não indica exatamente o que aconteceu de errado — mais para a frente, veremos como melhorar essa mensagem.

```
{
  "timestamp": "2022-09-26T12:26:38.723+00:00",
  "status": 400,
  "error": "Bad Request",
  "path": "/user"
}
```

O último serviço será para excluir um usuário da lista e será bem parecido com o serviço de busca. Ele também receberá um CPF como parâmetro na URL e fará uma busca pelo usuário. Caso ele seja encontrado, ele será removido da lista e o serviço retornará `true` ; caso contrário, retornará `false` . Uma diferença é que esse método usará o verbo DELETE do protocolo HTTP. A anotação `@DeleteMapping` cria um serviço com o verbo DELETE e a anotação `@PathVariable` funciona da mesma forma que no serviço GET anterior, o CPF será passado para o serviço na URL.

```
@DeleteMapping("/{cpf}")
public boolean remover(@PathVariable String cpf) {
    return usuarios
        .removeIf(userDTO -> userDTO.getCpf().equals(cpf));
}
```

Por exemplo, a chamada para esse serviço com o endereço `http://localhost:8080/user/123` removerá o usuário Eduardo da lista.

Com o código desenvolvido neste capítulo, já temos uma primeira API já funcional. Obviamente, ainda temos bastante trabalho até termos uma aplicação pronta para produção. Nos próximos três capítulos, implementaremos a primeira versão de cada uma das três APIs e, em todas elas, já faremos o acesso ao banco de dados com o Spring Data.

CAPÍTULO 4

Serviço de usuários (user-api)

Neste capítulo, continuaremos a implementação da *user-api*, que é o serviço para o gerenciamento de usuários da aplicação. A mudança mais importante aqui será a utilização de um banco de dados em vez de uma simples lista. Para isso, será utilizado o PostgreSQL e também precisaremos configurar o Spring Data no Maven da aplicação. Depois, serão desenvolvidas as três camadas da aplicação: *Repository*, *Service* e *Controller*.

4.1 Configuração do Spring Data nos serviços

Para configurar a aplicação para acessar o banco de dados, será necessário adicionar três novas dependências. A primeira é o `spring-boot-starter-data-jpa`, que é a versão do Spring Data já pronta para ser utilizada com o Spring Boot. A segunda é o `org.flywaydb`, que faz as migrações do banco de dados. Se você não conhece as migrações, o box a seguir apresenta esse conceito. A última dependência é o `org.postgresql`, que possui o conector para o PostgreSQL. Essas configurações serão exatamente as mesmas nos três microserviços.

MIGRAÇÕES

A ideia de migrações é manter as mudanças do modelo de dados versionadas e reproduzíveis. A atualização de banco de dados sempre foi um problema, já que os scripts normalmente não são mantidos junto ao código-fonte da aplicação e a aplicação deles era feita de forma manual. Com as migrações, os scripts são mantidos junto ao código-fonte e as mudanças são aplicadas no banco de dados automaticamente assim que a aplicação for implantada. Neste livro, usaremos o Flyway (<https://flywaydb.org>) para o gerenciamento de migrações.


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

Além disso, no arquivo `application.properties`, é necessário adicionar as configurações de como acessar o banco de dados. Esse arquivo deve ficar na pasta `/src/main/resources` e serve para definir diversas configurações da aplicação — ele será mais utilizado nos próximos capítulos. Todos os projetos terão exatamente as mesmas configurações, com exceção de `spring.jpa.properties.hibernate.default_schema`, `spring.flyway.schemas` e `server.port`, pois cada microsserviço terá o seu *schema* de banco de dados e sua porta. Os schemas serão *users* na *user-api*, *products* na *product-api* e *shopping* na *shopping-api*. Com relação às portas da aplicação, quaisquer portas podem ser utilizadas. Eu utilizei a porta 8080 para a *user-api*, a 8081 para a *product-api* e a 8082 para a *shopping-api*.

`spring.jpa.properties.hibernate.default_schema` e `spring.flyway.schemas` são duas configurações que definem os schemas do banco de dados que serão utilizados em cada um dos microsserviços, mas elas configuram coisas diferentes. A primeira define qual schema a aplicação conectará quando for executada e a segunda, onde as tabelas do Flyway serão criadas. Se a propriedade `spring.flyway.schemas` não for definida, ele criará essa tabela no schema padrão, no caso do Postgres, o `public`.

```
## Application port
server.port=8080
```

```
## default connection pool
spring.datasource.hikari.connectionTimeout=20000
spring.datasource.hikari.maximumPoolSize=5

## PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/dev
spring.datasource.username=postgres
spring.datasource.password=postgres

## Default Schema
spring.flyway.schemas=users
spring.jpa.properties.hibernate.default_schema=users
```

Essas configurações são necessárias nos três projetos, lembre-se apenas de mudar o schema do banco de dados e a porta da aplicação.

Agora, vamos implementar as três camadas, *Repository*, *Service* e *Controller*.

4.2 Camada de dados (Repository)

Vamos usar o conceito de migrações para a criação do banco de dados. Para isso, devemos criar um arquivo `sql` com o nome

`V1__create_user_table.sql` dentro da pasta `/src/main/resources/db/migration`. O V1 é para indicar a ordem dos scripts, o que é importante porque o Spring verificará se uma migração já foi aplicada no banco de dados sempre que subirmos a nossa aplicação; se sim, a migração será ignorada; caso contrário, ela será executada no banco de dados. Por enquanto teremos apenas uma migração, mas nos próximos capítulos acrescentaremos mais scripts para explicar melhor como funciona o processo de migrações.

```
create schema if not exists users;

create table users.user (
    id bigserial primary key,
    nome varchar(100) not null,
```

```
    cpf varchar(100) not null,  
    endereco varchar(100) not null,  
    email varchar(100) not null,  
    telefone varchar(100) not null,  
    data_cadastro timestamp not null  
);
```

Como o script mostra, criaremos uma tabela chamada `user` com os campos `id`, `nome`, `cpf`, `endereco`, `email`, `telefone` e `data_cadastro`.

Para todas as classes DTOs que foram criadas no capítulo anterior, também será necessário criar uma entidade, que é o objeto que possui exatamente a mesma estrutura do banco de dados. Essas classes são anotadas com um `@Entity`, indicando que elas representam uma tabela do BD. Nessa classe, o `id` da entidade é marcado com a anotação `@Id` e a `@GeneratedValue`, que indica a forma com que o `id` é gerado — no nosso caso o `IDENTITY`. Além disso, os outros atributos podem ser marcados com a anotação `@Column`, que possui diversas propriedades, como se o atributo é obrigatório ou não e o tamanho máximo. Note também que, assim como na classe `UserDTO` do capítulo anterior, foram adicionadas as anotações do Lombok na classe `User`.

Se a anotação `@Column` não for adicionada, serão assumidas as propriedades padrões para essa coluna. Nesse caso, os nomes do atributo e da coluna do banco de dados devem ser iguais; se não, a aplicação não funcionará. A listagem a seguir mostra o código da entidade para a tabela `user`.

```
package com.santana.java.back.end.model;  
  
import jakarta.persistence.Entity;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Id;  
import java.time.LocalDateTime;  
  
@Getter
```

```

@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nome;
    private String cpf;
    private String endereco;
    private String email;
    private String telefone;
    private LocalDateTime dataCadastro;

    public static User convert(UserDTO userDTO) {
        User user = new User();
        user.setNome(userDTO.getNome());
        user.setEndereco(userDTO.getEndereco());
        user.setCpf(userDTO.getCpf());
        user.setEmail(userDTO.getEmail());
        user.setTelefone(userDTO.getTelefone());
        user.setDataCadastro(userDTO.getDataCadastro());
        return user;
    }
}

```

O método `convert`, na classe `User`, é importante porque precisaremos converter instâncias da entidade `User` para instâncias da classe `UserDTO`. Isso será utilizado em diversos serviços, então é melhor ter um método que realiza essa operação do que ficar duplicando código em vários locais. A mesma coisa será necessária na classe `UserDTO` — a listagem a seguir mostra a mesma implementação nessa classe.

```

public static UserDTO convert(User user) {
    UserDTO userDTO = new UserDTO();
    userDTO.setNome(user.getNome());
    userDTO.setEndereco(user.getEndereco());
}

```

```

        userDTO.setCpf(user.getCpf());
        userDTO.setEmail(user.getEmail());
        userDTO.setTelefone(user.getTelefone());
        userDTO.setDataCadastro(user.getDataCadastro());
        return userDTO;
    }

```

Com o Spring Data, agora basta criar um repositório para a entidade criada. O repositório é uma interface anotada com `@Repository`, que também é um *bean* do Spring e que será automaticamente instanciado na inicialização da aplicação. Essa interface deve estender a `JpaRepository`, passando a entidade e o tipo do id, no nosso caso, `User` e `Long`. Com isso, já estão disponíveis diversos métodos básicos para o acesso ao banco de dados, como busca, busca por id, inserção, exclusão etc. Também é possível adicionar algumas consultas simples apenas com o nome do método, como o `findByCpf` e o `queryByNameLike`, assim como construir consultas mais complexas, o que será demonstrado nos próximos capítulos.

```

package com.santana.java.back.end.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.santana.java.back.end.model.User;

import java.util.List;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    User findByCpf(String cpf);

    List<User> queryByNameLike(String name);

}

```

Como é possível verificar nos métodos `findByCpf` e `queryByNameLike`, algumas consultas podem ser criadas apenas com o nome do método. Esses métodos devem ter algumas palavras-chaves no nome como `find`, `and`,

or , like e o nome do campo. Nos próximos serviços, criaremos algumas consultas mais complexas, e mais algumas dessas palavras-chaves serão apresentadas.

4.3 Camada de serviços (Service)

O *controller* poderia chamar o repositório diretamente, porém a maioria das aplicações possui uma camada intermediária, chamada de *service*, que é onde ficam as regras de negócio da aplicação. Essas classes devem ser anotadas com `@Service` e normalmente são responsáveis por fazer chamadas ao repositório e também a outros serviços. A listagem a seguir mostra um exemplo de uma classe `Service` que será utilizada para acessar os dados na tabela `user`.

```
package com.santana.java.back.end.service;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import org.springframework.stereotype.Service;

import com.santana.java.back.end.dto.UserDTO;
import com.santana.java.back.end.model.User;
import com.santana.java.back.end.repository.UserRepository;

@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;

    public List<UserDTO> getAll() {
        List<User> usuarios = userRepository.findAll();
        return usuarios
            .stream()
            .map(UserDTO::convert)
    }
}
```

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

Na listagem, é possível ver que a classe foi anotada com `@Service`, indicando que uma instância dela será criada na criação da aplicação. A classe também possui um atributo do tipo `UserRepository`. Na primeira edição do livro, esse atributo estava anotado com `@Autowired`, que serve para fazer injeção de dependências. Porém, usando o Lombok, poderemos remover todas essas anotações e substituí-las pela anotação `@RequiredArgsConstructor`, que fica na classe. Com ela, evitamos ter que utilizar essa anotação `@Autowired` em classes que possuem a injeção de diversas dependências.

Além disso, a classe possui o método `getAll`, que faz as seguintes operações:

1. Chama o método `findAll`, do `UserRepository`, que retorna uma lista de usuários, sendo instâncias da entidade `User`;
2. Transforma a lista em um stream e chama o método `map` para transformar a lista de entidades em uma lista de DTOs;
3. Retorna a lista de DTOs.

Além do método `getAll`, adicionaremos mais seis métodos nesta classe: `findById`, `save`, `delete`, `findByCpf`, `queryByName` e `editUser`. O primeiro busca um usuário por um id específico; o segundo salva um usuário no banco de dados; o terceiro exclui um usuário do banco de dados. O `findByCpf` faz a busca de um usuário por seu CPF e o `queryByName` faz uma busca pelo nome do usuário, mas, diferentemente da busca por id ou pelo CPF, a busca não será exata, mas sim pela inicial do nome passada no parâmetro. Por exemplo, se o parâmetro `nome` tiver valor `Mar%`, a busca retornará pessoas com o nome Marcela, Marcelo ou Marcos.

O último método, o `editUser`, receberá o id de um usuário e um objeto da classe `UserDTO`. Com o id, será buscado um usuário no banco de dados e, caso ele exista, os dados desse usuário serão atualizados com o que foi

recebido no objeto do tipo `UserDTO` ; caso o usuário não exista, será retornado um erro de que o usuário não foi encontrado.

```
public UserDTO findById(long userId) {
    return userRepository
        .findById(userId).orElseThrow(() -> new
RuntimeException());
}

public UserDTO save(UserDTO userDTO) {
    userDTO.setDataCadastro(LocalDateTime.now());
    User user = userRepository.save(User.convert(userDTO));
    return UserDTO.convert(user);
}

public UserDTO delete(long userId) {
    User user = userRepository
        .findById(userId).orElseThrow(() -> new
RuntimeException());
    userRepository.delete(user.get());
    return user;
}

public UserDTO findByCpf(String cpf) {
    User user = userRepository.findByCpf(cpf);
    if (user != null) {
        return UserDTO.convert(user);
    }
    return null;
}

public List<UserDTO> queryByName(String name) {
    List<User> usuarios = userRepository.queryByNameLike(name);
    return usuarios
        .stream()
        .map(UserDTO::convert)
        .collect(Collectors.toList());
}

public UserDTO editUser(Long userId, UserDTO userDTO) {
    User user = userRepository
```



```

        .findById(userId).orElseThrow(() -> new
RuntimeException());
        if (userDTO.getEmail() != null &&
            !user.getEmail().equals(userDTO.getEmail())) {
            user.setEmail(userDTO.getEmail());
        }
        if (userDTO.getTelefone() != null &&
            !user.getTelefone().equals(userDTO.getTelefone())) {
            user.setTelefone(userDTO.getTelefone());
        }
        if (userDTO.getEndereco() != null &&
            !user.getEndereco().equals(userDTO.getEndereco())) {
            user.setEndereco(userDTO.getEndereco());
        }

        user = userRepository.save(user);
        return UserDTO.convert(user);
    }

```

O Spring possui uma funcionalidade bastante interessante, que é a paginação. É bastante simples de implementar utilizando o Spring Data, que já disponibiliza duas interfaces prontas para serem usadas nesses casos: o `Page` e o `Pageable`. O `Page` é uma coleção de objetos já paginados, que possui dados como o tamanho de uma página, quantos existem no total e qual a página que o usuário está visualizando. Já o `Pageable` recebe os parâmetros que o usuário pode usar para acessar uma página, como qual o tamanho e qual a página que deve ser acessada.

Por último, a interface `JpaRepository` já possui um método `findAll` que recebe um objeto `Pageable` e faz a consulta fazendo paginação. O método a seguir mostra a implementação da consulta com paginação.

```

public Page<UserDTO> getAllPage(Pageable page) {
    Page<User> users = userRepository.findAll(page);
    return users
        .map(UserDTO::convert);
}

```

4.4 Camada dos controladores (Controllers)

Os *controllers* mudarão pouco em relação ao capítulo anterior. Basicamente, eles chamarão a classe da camada de serviço. A classe continua com a mesma anotação `@RestController`, e os métodos com as anotações `@GetMapping`, `@PostMapping` e `@DeleteMapping`. Uma diferença é a injeção da dependência da classe de serviços `UserService`. Os métodos nela são os mesmos, com a diferença de que agora são chamados os métodos da camada de serviço em vez de manipular a lista em memória. Eu também mudei o nome de algumas rotas para ficar mais próximo do que usaremos nos outros serviços; por exemplo, a `DELETE /removeUser/{cpf}` virou `DELETE /user/{id}`.

Outro ponto importante nas rotas são os códigos de retorno. Por padrão, as rotas REST retornam o código 200, porém é uma boa prática mudar o tipo de retorno dependendo do tipo da rota. Por exemplo, nas rotas que criam um objeto, o melhor código de retorno é o 201, que indica que um recurso foi criado no servidor; para uma rota `DELETE`, normalmente é retornado um código 204, indicando que a rota não terá nenhum dado de retorno.

```
package com.santana.java.back.end.controller;

import java.util.List;

import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

import com.santana.java.back.end.dto.UserDTO;
import com.santana.java.back.end.exception.UserNotFoundException;
import com.santana.java.back.end.service.UserService;

@RestController
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {
```

```

private final UserService userService;

@GetMapping
public List<UserDTO> getUsers() {
    return userService.getAll();
}

@GetMapping("/{id}")
public UserDTO findById(@PathVariable Long id) {
    return userService.findById(id);
}

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public UserDTO newUser(@RequestBody @Valid UserDTO userDTO) {
    return userService.save(userDTO);
}

@GetMapping("/{cpf}/cpf")
public UserDTO findByCpf(
    @RequestParam(name="key") String key,
    @PathVariable String cpf) {
    return userService.findByCpf(cpf, key);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable Long id) throws
UserNotFoundException {
    userService.delete(id);
}

@GetMapping("/search")
public List<UserDTO> queryByName(
    @RequestParam(name="nome", required = true) String
nome) {
    return userService.queryByName(nome);
}
}

```

Uma novidade aqui é a rota `/user/search`, que fará a busca pelo nome recebido como parâmetro - o nome pode ser completo ou apenas parte do nome. Se o nome for completo, a rota retornará apenas um usuário; se o usuário passar apenas parte do nome, pode ser retornada uma lista de usuários. Outra novidade é a anotação `@RequestParam`, que deve ser usada quando queremos passar parâmetros na URL para a rota. Veja que a anotação recebeu que o parâmetro é obrigatório. A chamada para essa rota pode ser feita pela URL `http://localhost:8080/user/search?nome=mar%`; nesse caso, a resposta para a chamada será:

```
[
  {
    "nome": "marcela",
    "cpf": "123",
    "endereco": "Rua abc",
    "email": "marcela@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  },
  {
    "nome": "marcelo",
    "cpf": "123",
    "endereco": "Rua abc",
    "email": "marcelo@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2019-11-17T21:04:51.701+0000",
  }
]
```

Como o parâmetro `nome` foi anotado como obrigatório, caso a rota seja chamada sem ele, apenas com `http://localhost:8080/user/search`, retornará o erro mostrado a seguir.

```
{
  "timestamp": "2020-05-30T01:22:41.581+0000",
  "status": 400,
  "error": "Bad Request",
  "message": "Required String parameter 'nome' is not present",
  "path": "/user/search"
}
```

Além da rota de `search` , também foram adicionadas outras duas rotas no `UserController` : as rotas de edição de usuário e a rota que retorna os usuários usando paginação. A listagem a seguir mostra a implementação da rota de edição de usuários.

```
@PatchMapping("/{id}")
public UserDTO editUser(@PathVariable Long id,
                        @RequestBody UserDTO userDTO) {
    return userService.editUser(id, userDTO);
}
```

A rota de edição é parecida com a de criação de usuário. Ela recebe um `JSON` que representa um usuário e atualiza os dados do usuário conforme o que foi recebido. Por exemplo, se quisermos atualizar o endereço do usuário que tem o CPF `123` , podemos chamar a rota `POST`

<http://localhost:8080/user/123>, passando apenas os dados que queremos atualizar, como o endereço e o telefone:

```
{
    "endereco": "Rua abc",
    "telefone": "1234-3454"
}
```

A outra rota adicionada é a busca de usuários usando paginação. A chamada é feita na rota `GET` <http://localhost:8080/user/pageable?size=2&page=1> . Note os parâmetros `size` e `page` . O `size` indica que as páginas terão tamanho 2, e que deve ser retornada a página 1 (lembrando que as páginas começam no índice 0). A listagem a seguir mostra o código dessa rota.

```
@GetMapping("/pageable")
public Page<UserDTO> getUsersPage(Pageable pageable) {
    return userService.getAllPage(pageable);
}
```

O retorno dessa rota é uma lista de usuários e também os dados sobre a paginação, como mostrado na listagem a seguir.

```
{
    "content": [
```

```
{
  "nome": "Marcelo",
  "cpf": "123",
  "endereco": "Rua abc",
  "key": "b4a05433-cf63-447f-bfc4-535b313d99a9",
  "email": "marcelo@email.com",
  "telefone": "1234",
  "dataCadastro": "2022-10-01T14:30:41.48008"
},
{
  "nome": "Bruna",
  "cpf": "123",
  "endereco": "Rua abc",
  "key": "a124a203-a8c1-40d1-9a08-ae907f63bf9b",
  "email": "bruna@email.com",
  "telefone": "1234",
  "dataCadastro": "2022-10-01T14:30:46.979584"
}
],
"pageable": {
  "sort": {
    "empty": true,
    "sorted": false,
    "unsorted": true
  },
  "offset": 2,
  "pageNumber": 1,
  "pageSize": 2,
  "paged": true,
  "unpaged": false
},
"totalPages": 2,
"totalElements": 4,
"last": true,
"size": 2,
"number": 1,
"sort": {
  "empty": true,
  "sorted": false,
  "unsorted": true
},
}
```

```
"numberOfElements": 2,  
"first": false,  
"empty": false  
}  
,
```

Note que, no retorno, existe um objeto chamado *content*, que possui a lista de usuários; depois, existem diversos dados sobre paginação, como o número de página, o número total de elementos e o tamanho da página. Esses dados podem ser utilizados no *front-end* para a criação de uma tabela com paginação.

Ainda existem diversas melhorias nos serviços que serão feitas nos próximos capítulos, mas essa versão já é funcional e salva os dados no banco de dados. As chamadas e as respostas dos outros serviços são idênticas às apresentadas no capítulo anterior.

CAPÍTULO 5

Serviço de produtos (product-api)

O segundo serviço gerenciará os produtos da aplicação. A configuração do Maven é a mesma do serviço anterior, com a exceção óbvia do `artifactId`, que, nesse caso, será *product-api*. O arquivo `application.properties` desse projeto também será igual ao do *user-api*, as únicas diferenças são o *schema* (products) e a porta (8081). Para este serviço, criaremos duas tabelas: a dos produtos e a das categorias de produtos. Isso será feito para mostrar o relacionamento entre duas tabelas e também implementar algumas buscas mais complexas no banco de dados.

5.1 Camada de dados (Repository)

A primeira nova implementação desse serviço é a criação das tabelas. Primeiro, adicionaremos a tabela `category`, que será uma forma de agrupar os produtos. Ela possui apenas o campo `nome`. A listagem a seguir mostra sua criação. Esse script deve ter o nome `v1__create_category_table.sql`.

```
create schema if not exists products;
```

```
create table products.category (  
    id bigserial primary key,  
    nome varchar(100) not null  
);
```

Depois, é definida a tabela `product`, com um arquivo que deve ter o nome `v2__create_product_table.sql`. Os campos da tabela são o `id`, o `nome`, o `preco` e a `descrição` do produto. Além disso, existe o `category_id`, que é uma chave estrangeira que relaciona a tabela `product` e `category`.

```
create table products.product (  
    id bigserial primary key,  
    product_identifier varchar not null,
```



```
    nome varchar(100) not null,  
    descricao varchar not null,  
    preco float not null,  
    category_id bigint REFERENCES products.category(id)  
);
```

Finalmente, vamos adicionar algumas categorias predefinidas em nosso sistema. Para isso, criaremos uma migração que insere três categorias no banco de dados. O arquivo dessa migração é o

V3__insert_categories.sql .

```
INSERT INTO products.category(id, nome) VALUES(1, 'Eletrônico');  
INSERT INTO products.category(id, nome) VALUES(2, 'Móveis');  
INSERT INTO products.category(id, nome) VALUES(3, 'Brinquedos');
```

Observação: é possível criar as duas tabelas em apenas uma migração, mas, para mostrar como criar várias migrações, as duas tabelas e os *inserts* foram feitos em arquivos diferentes.

O primeiro passo para a implementação do serviço é a criação das entidades, as classes Java que representam as tabelas do banco de dados. Assim como no serviço anterior, essas classes têm os mesmos atributos que a tabela. A listagem a seguir mostra a classe `Product` .

```
package com.santana.java.back.end.model;  
  
import jakarta.persistence.Entity;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Id;  
import jakarta.persistence.JoinColumn;  
import jakarta.persistence.ManyToOne;  
  
import com.santana.java.back.end.dto.CategoryDTO;  
import com.santana.java.back.end.dto.ProductDTO;  
import lombok.AllArgsConstructor;  
import lombok.Getter;  
import lombok.NoArgsConstructor;
```

```

import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity(name="product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nome;
    private Float preco;
    private String descricao;
    private String productIdentifier;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    public static Product convert(ProductDTO productDTO) {
        Product product = new Product();
        product.setNome(productDTO.getNome());
        product.setPreco(productDTO.getPreco());
        product.setDescricao(productDTO.getDescricao());
        product.setProductIdentifier(
            productDTO.getProductIdentifier());
        if (productDTO.getCategoryDTO() != null) {
            product.setCategory(
                Category.convert(productDTO.getCategoryDTO()));
        }
        return product;
    }
}

```

A listagem a seguir mostra a classe `category`. Ela é bem simples, tem apenas o nome da categoria.

```

package com.santana.java.back.end.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

import com.santana.java.back.end.dto.CategoryDTO;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity(name="category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nome;

    public static Category convert(CategoryDTO categoryDTO) {
        Category category = new Category();
        category.setId(categoryDTO.getId());
        category.setNome(categoryDTO.getNome());
        return category;
    }
}

```

Além das entidades, também criaremos os DTOs para as classes `Product` e `Category`, que são a `ProductDTO` e a `CategoryDTO`. Note as anotações `@NotBlank` e `@NotNull`, elas validarão se os campos possuem valores válidos quando formos salvar um novo produto. A diferença entre as anotações é que a `@NotBlank` verifica se uma `String` é diferente de nulo e

também não é vazia; a `@NotNull` deve ser utilizada para campos de outros tipos, como o `Float`. A listagem a seguir mostra o código `ProductDTO`.

```
package com.santana.java.back.end.dto;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

import com.santana.java.back.end.model.Product;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ProductDTO {

    @NotBlank
    private String productIdentifier;
    @NotBlank
    private String nome;
    @NotBlank
    private String descricao;
    @NotNull
    private Float preco;
    @NotNull
    private CategoryDTO category;

    public static ProductDTO convert(Product product) {
        ProductDTO productDTO = new ProductDTO();
        productDTO.setNome(product.getNome());
        productDTO.setPreco(product.getPreco());
        productDTO.setProductIdentifier(
            product.getProductIdentifier());
        productDTO.setDescricao(product.getDescricao());
        if (product.getCategory() != null) {
            productDTO.setCategoryDTO(
                CategoryDTO.convert(product.getCategory()));
        }
    }
}
```

```

        }
        return productDTO;
    }
}

```

A listagem a seguir mostra o código CategoryDTO .

```

package com.santana.java.back.end.dto;

import jakarta.validation.constraints.NotNull;

import com.santana.java.back.end.model.Category;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class CategoryDTO {

    @NotNull
    private Long id;
    private String nome;

    public static CategoryDTO convert(Category category) {
        CategoryDTO categoryDTO = new CategoryDTO();
        categoryDTO.setId(category.getId());
        categoryDTO.setNome(category.getNome());
        return categoryDTO;
    }
}

```

Para as classes Product e Category , também serão criadas as classes dos repositórios. O categoryRepository não tem nenhuma consulta complexa, por isso tem apenas a definição básica da interface.

```

package com.santana.java.back.end.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.santana.java.back.end.model.Category;

@Repository
public interface CategoryRepository
    extends JpaRepository<Category, Long> {

}

```

Já o `ProductRepository` tem uma consulta para recuperar todos os produtos de uma determinada categoria. Essa consulta foi implementada no método `getProductByCategory` e foi escrita na anotação `@Query`. Note que não é preciso fazer nenhum tipo de implementação dentro do método, o Spring Data faz tudo automaticamente, criando as instâncias da classe `Product` sozinho. Além disso, existe também o método `findByProductIdentifier`, que faz uma busca simples pelo identificador do produto.

```

package com.santana.java.back.end.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import com.santana.java.back.end.model.Product;

@Repository
public interface ProductRepository extends JpaRepository<Product,
Long> {

    @Query(value = "select p "
        + "from product p "
        + "join category c on p.category.id = c.id ")

```

```

        + "where c.id = :categoryId ")
    public List<Product> getProductByCategory(
        @Param("categoryId") long categoryId);

    public Product findByProductIdentifier(
        String productIdentifier);
}

```

5.2 Camada de serviços (Service)

A classe `ProductService` contém todos os serviços relacionados à entidade `Product`. Os serviços implementados são: `getAll`, que retorna todos os produtos cadastrados; `getProductByCategoryId`, que retorna todos os produtos de uma determinada categoria; `findByProductIdentifier`, que retorna um produto para o id selecionado; `save`, que salva um novo produto; e `delete`, que exclui um produto do banco de dados.

```

package com.santana.java.back.end.service;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.santana.java.back.end.dto.ProductDTO;
import
com.santana.java.back.end.exception.ProductNotFoundException;
import com.santana.java.back.end.model.Product;
import com.santana.java.back.end.repository.ProductRepository;

@Service
@RequiredArgsConstructor
public class ProductService {

```

```

private ProductRepository productRepository;

public List<ProductDTO> getAll() {
    List<Product> products = productRepository.findAll();
    return products
        .stream()
        .map(ProductDTO::convert)
        .collect(Collectors.toList());
}

public List<ProductDTO> getProductByCategoryId(
    Long categoryId) {

    List<Product> products =
        productRepository.getProductByCategory(categoryId);
    return products
        .stream()
        .map(ProductDTO::convert)
        .collect(Collectors.toList());
}

public ProductDTO findByProductIdentifier(
    String productIdentifier) {

    Product product =
productRepository.findByProductIdentifier(productIdentifier);
    if (product != null) {
        return ProductDTO.convert(product);
    }
    return null;
}

public ProductDTO save(ProductDTO productDTO) {
    Product product =
        productRepository.save(Product.convert(productDTO));
    return ProductDTO.convert(product);
}

public void delete(long productId) {
    Optional<Product> product =
        productRepository.findById(productId);
}

```



```

        if (product.isPresent()) {
            productRepository.delete(product.get());
        }
    }
}

```

Também foram adicionados mais dois métodos para a edição de um produto e o serviço para a listagem de produtos, mas agora utilizando paginação. O primeiro é o método `editProduct`, que recebe um objeto da classe `ProductDTO` com os dados para a atualização do produto e o `id` do produto que se deseja alterar. Note que os únicos dados que podem ser alterados são o nome do produto e o preço.

```

public ProductDTO editProduct(long id, ProductDTO dto) {
    Product product = productRepository.findById(id).orElseThrow(()
-> new RuntimeException("Product not found"));

    if (dto.getNome() != null || !dto.getNome().isEmpty()) {
        product.setNome(dto.getNome());
    }
    if (dto.getPreco() != null) {
        product.setPreco(dto.getPreco());
    }
    return ProductDTO.convert(productRepository.save(product));
}

```

O segundo método é o `getAllPage`, que vai retornar todos os produtos cadastrados, mas usando paginação.

```

public Page<ProductDTO> getAllPage(Pageable page) {
    Page<Product> users = productRepository.findAll(page);
    return users
        .map(ProductDTO::convert);
}

```

5.3 Camada dos controladores (Controllers)

A classe `ProductController` também não tem nenhuma grande novidade. Na *product-api*, foram criadas as seguintes rotas: criar um novo produto, excluir um produto, recuperar as informações do produto por id, listar todos os produtos e listar todos os produtos de uma determinada categoria. A listagem a seguir mostra o código completo dessa classe.

```
package com.santana.java.back.end.controller;

import java.util.List;

import jakarta.validation.Valid;

import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.web.bind.annotation.*;

import com.santana.java.back.end.dto.ProductDTO;
import com.santana.java.back.end.exception.ProductNotFoundException;
import com.santana.java.back.end.service.ProductService;

@RestController
@RequestMapping("/product")
@RequiredArgsConstructor
public class ProductController {

    private ProductService productService;

    @GetMapping
    public List<ProductDTO> getProducts() {
        return productService.getAll();
    }

    @GetMapping("/category/{categoryId}")
    public List<ProductDTO> getProductByCategory(
        @PathVariable Long categoryId) {
        return productService.getProductByCategoryId(categoryId);
    }
}
```

```

@GetMapping("/{productIdentifier}")
public ProductDTO findById(
    @PathVariable String productIdentifier) {
    return productService
        .findByProductIdentifier(productIdentifier);
}

@PostMapping
public ProductDTO newProduct(
    @Valid @RequestBody ProductDTO userDTO) {
    return productService.save(userDTO);
}

@DeleteMapping("/{id}")
public ProductDTO delete(@PathVariable Long id) {
    return productService.delete(id);
}
}

```

Além das rotas anteriores, foram adicionadas as rotas para a edição de um produto, a POST /product/{id} , e, também, a que lista os produtos com paginação, a GET /product/pageable .

```

@PostMapping("/{id}")
public ProductDTO editProduct(@PathVariable Long id,
    @RequestBody ProductDTO productDTO) {
    return productService.editProduct(id, productDTO);
}

@GetMapping("/pageable")
public Page<ProductDTO> getProductsPage(Pageable pageable) {
    return productService.getAllPage(pageable);
}

```

5.4 Testando os serviços

Com o *controller* criado, podemos chamar os serviços para testá-los. O primeiro a ser testado é o que cria um novo produto. Ele pode ser chamado na URL <http://localhost:8081/product/> com o método HTTP POST, e, no corpo da requisição, deve ser passado um JSON com a definição de um produto, como no exemplo a seguir.

```
{
  "productIdentifier": "tv",
  "nome": "TV",
  "preco": 1000,
  "descricao": "Uma TV",
  "category": {
    "id": 1
  }
}
```

Para essa requisição funcionar, o id da categoria deve ser válido, se não, a chamada retornará o seguinte erro:

```
{
  "timestamp": "2019-11-04T19:09:07.005+0000",
  "status": 500,
  "error": "Internal Server Error",
  "message": "could not execute statement; SQL [n/a]; constraint [product_category_id_fkey]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement",
  "path": "/newProduct"
}
```

Esse erro não é muito amigável, mas é possível perceber a `ConstraintViolationException`, indicando que não existe uma categoria com o id passado. No capítulo 9, veremos como retornar mensagens de erro mais legíveis para o usuário.

Se um dos campos que foram definidos como obrigatórios não for passado no JSON, a validação no DTO será feita e a requisição retornará o seguinte erro:

```
{
  "timestamp": "2020-06-14T15:42:32.768+00:00",
  "status": 400,
  "error": "Bad Request",
  "message": "",
  "path": "/product/"
}
```

Nesse caso, seria útil informar qual campo obrigatório não está sendo retornado — no capítulo 9, retornaremos uma mensagem melhor para esse erro também.

Outro serviço disponível é o que lista todos os produtos do banco de dados. A URL desse serviço é <http://localhost:8081/product> com o método HTTP GET. A listagem a seguir mostra a resposta para ele:

```
[
  {
    "productIdentifier": "tv",
    "nome": "TV",
    "preco": 1000.0,
    "descricao": "Uma televisão",
    "category": {
      "id": 1,
      "nome": "Eletrônico"
    }
  },
  {
    "productIdentifier": "video-game",
    "nome": "Video Game",
    "preco": 2000.0,
    "descricao": "Um video game",
    "category": {
      "id": 1,
      "nome": "Eletrônico"
    }
  },
  {
    "productIdentifier": "carro-controle-remoto",
    "nome": "Carrinho de Controle Remoto",
    "preco": 100.0,
```

```
    "descricao": "Um carrinho de controle remoto",
    "category": {
      "id": 1,
      "nome": "Brinquedos"
    }
  }
]
```

Agora já temos implementados os dois microsserviços que armazenam os dados necessários para efetuar uma compra: os usuários na *user-api* e os produtos na *product-api*. Esses dados serão importantes para validar uma compra, já que ela deve ser efetuada por um usuário cadastrado no sistema e conter uma lista de produtos que existem no catálogo da loja.

CAPÍTULO 6

Serviço de compras (shopping-api)

O serviço de compras também será parecido com os anteriores, mas serão adicionadas algumas buscas mais complexas: buscaremos todas as compras de um certo usuário ou de um determinado período de tempo. Esse microsserviço será importante também no próximo capítulo, onde veremos a comunicação entre os serviços. O arquivo `application.properties` desse projeto também será igual aos anteriores, sendo que as únicas diferenças são o *schema* (shopping) e a porta (8082).

6.1 Camada de dados (Repository)

Vamos iniciar com a criação das tabelas para esse serviço. Primeiro, adicionaremos a tabela `shop`, que armazenará todas as compras registradas em nosso sistema. Além disso, criaremos a tabela `item`, que conterá todos os itens de uma compra. A listagem a seguir mostra a criação dessas tabelas. Esse script deve ter o nome `v1__create_shop_table.sql`.

```
create schema if not exists shopping;
```

```
create table shopping.shop (  
    id bigserial primary key,  
    user_identifier varchar(100) not null,  
    date timestamp not null,  
    total float not null  
);
```

```
create table shopping.item (  
    shop_id bigserial REFERENCES shopping.shop(id),  
    product_identifier varchar(100) not null,  
    price float not null  
);
```

Refletindo as tabelas `shop` e `item`, temos também as entidades `Shop` e `Item`. Ambas as classes têm os mesmos atributos que as tabelas. Uma novidade nessas entidades é o relacionamento de uma coleção dependente, pois uma compra tem uma coleção de itens. A melhor forma de mapear esse tipo de relação é usando a anotação `@ElementCollection`. Ela tem o atributo `fetch`, que pode ter os valores `Eager`, o que indica que os valores devem ser recuperados do BD junto da entidade principal, ou `Lazy`, que indica que a lista só deve ser recuperada quando for chamada.

Além disso, podemos utilizar a anotação `@CollectionTable` para definir qual é a tabela onde os itens estarão armazenados — no caso, a tabela `shop_item`. A anotação `@JoinColumn` define qual coluna da tabela `shop_item` será unida (*join*) à tabela `shop` — no caso, a coluna `shop_id`. Assim como nos outros projetos, as entidades possuem um método chamado `convert`, que converte um DTO para uma entidade.

```
package com.santana.java.back.end.model;

import java.time.LocalDateTime;
import java.util.List;
import java.util.stream.Collectors;

import jakarta.persistence.CollectionTable;
import jakarta.persistence.ElementCollection;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;

import com.santana.java.back.end.dto.ShopDTO;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
```



```

@NoArgsConstructor
@AllArgsConstructor
@Entity(name="shop")
public class Shop {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String userIdentifier;
    private float total;
    private LocalDateTime date;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "item",
        joinColumns = @JoinColumn(name = "shop_id"))
    private List<Item> items;

    public static Shop convert(ShopDTO shopDTO) {
        Shop shop = new Shop();
        shop.setUserIdentifier(shopDTO.getUserIdentifier());
        shop.setTotal(shopDTO.getTotal());
        shop.setDate(shopDTO.getDate());
        shop.setItems(shopDTO
            .getItems()
            .stream()
            .map(Item::convert)
            .collect(Collectors.toList()));
        return shop;
    }
}

```

A classe `Item` é um pouco diferente. Nela, utilizamos a anotação `@Embeddable`, indicando que ela pode ser embutida em uma entidade. Uma classe com a anotação `@Embeddable` não tem vida própria, ela sempre depende de uma entidade, isto é, de uma classe que tenha a anotação `@Entity`.

```
package com.santana.java.back.end.model;
```

```

import jakarta.persistence.Embeddable;

import com.santana.java.back.end.dto.ItemDTO;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Embeddable
public class Item {

    private String productIdentifier;
    private Float price;

    public static Item convert(ItemDTO itemDTO) {
        Item item = new Item();
        item.setProductIdentifier(
            itemDTO.getProductIdentifier());
        item.setPrice(itemDTO.getPrice());
        return item;
    }
}

```

Assim como nos outros projetos, para as classes `Shop` e `Item` criaremos também os DTOs.

```

package com.santana.java.back.end.dto;

import java.util.Date;
import java.util.List;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

import com.santana.java.back.end.model.Shop;
import lombok.AllArgsConstructor;

```

```

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ShopDTO {

    @NotBlank
    private String userIdentifier;
    @NotNull
    private Float total;
    @NotNull
    private LocalDateTime date;
    @NotNull
    private List<ItemDTO> items;

    public static ShopDTO convert(Shop shop) {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setUserIdentifier(shop.getUserIdentifier());
        shopDTO.setTotal(shop.getTotal());
        shopDTO.setDate(shop.getDate());
        return shopDTO;
    }
}

package com.santana.java.back.end.dto;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

import com.santana.java.back.end.model.Item;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter

```

```

@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ItemDTO {

    @NotBlank
    private String productIdentifier;
    @NotNull
    private Float price;

    public static ItemDTO convert(Item item) {
        ItemDTO itemDTO = new ItemDTO();
        itemDTO.setProductIdentifier(
            item.getProductIdentifier());
        itemDTO.setPrice(item.getPrice());
        return itemDTO;
    }
}

```

Para a classe `Shop`, será criada a classe `ShopRepository`, onde serão adicionadas algumas consultas um pouco mais complexas para mostrar algumas das capacidades do Spring Data. Esses métodos são o `findAllByUserIdentifier`, o `findAllByTotalGreaterThan` e o `findAllByDateGreaterThanEquals`. O primeiro método vai recuperar todas as compras de um usuário específico, o segundo vai buscar todas as compras que tenham um valor total maior do que o valor passado como parâmetro, e o terceiro vai retornar todas as compras a partir de uma data específica.

Note os padrões importantes para essas buscas. O primeiro é o `findAll`, indicando que a busca será por um ou mais resultados, o segundo é o `By{Atributo}`, que indica por qual atributo será feita a busca, e o terceiro é o `GreaterThan`, que faz um filtro para apenas valores maiores do que o passado como parâmetro serem buscados.

```

package com.santana.java.back.end.repository;

import java.util.Date;

```

```

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.santana.java.back.end.model.Shop;

@Repository
public interface ShopRepository
    extends JpaRepository<Shop, Long> {

    public List<Shop> findAllByUserIdentifier(
        String userIdentifier);

    public List<Shop> findAllByTotalGreaterThan(
        Float total);

    List<Shop> findAllByDateGreaterThan(
        LocalDateTime date);

}

```

6.2 Camada de serviços (Service)

A classe `ShopService` contém todos os serviços relacionados à entidade `Shop`, e são eles os principais da nossa aplicação, já que a ideia é permitir que usuários façam compras. Assim como nas outras classes de serviço, temos os métodos para retornar todas as compras, o `getAll`, o método para retornar uma compra pelo id, o `findById`, e o método para salvar uma compra, o `save`. Note que o método `save` calcula o preço total da compra a partir da lista de itens e também salva a data da compra com a data corrente do servidor.

Além dos métodos já comentados, essa classe tem dois métodos novos, o `getUser` e o `getDate`, que recuperam todas as compras de um determinado usuário ou de uma determinada data. A listagem a seguir mostra todos os métodos da classe `ShopService`.

```

package com.santana.java.back.end.service;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import com.santana.java.back.end.dto.ShopDTO;
import com.santana.java.back.end.model.Shop;
import com.santana.java.back.end.repository.ShopRepository;

@Service
@RequiredArgsConstructor
public class ShopService {

    private final ShopRepository shopRepository;

    public List<ShopDTO> getAll() {
        List<Shop> shops = shopRepository.findAll();
        return shops
            .stream()
            .map(ShopDTO::convert)
            .collect(Collectors.toList());
    }

    public List<ShopDTO> getByUser(String userIdentifier) {
        List<Shop> shops = shopRepository
            .findAllByUserIdentifier(userIdentifier);
        return shops
            .stream()
            .map(ShopDTO::convert)
            .collect(Collectors.toList());
    }

    public List<ShopDTO> getByDate(ShopDTO shopDTO) {
        List<Shop> shops = shopRepository
            .findAllByDateGreaterThan(shopDTO.getDate());
        return shops
    }

```

```

        .stream()
        .map(ShopDTO::convert)
        .collect(Collectors.toList());
    }

    public ShopDTO findById(long ProductId) {
        Optional<Shop> shop = shopRepository
            .findById(ProductId);
        if (shop.isPresent()) {
            return ShopDTO.convert(shop.get());
        }
        return null;
    }

    public ShopDTO save(ShopDTO shopDTO) {
        shopDTO.setTotal(shopDTO.getItems()
            .stream()
            .map(x -> x.getPrice())
            .reduce((float) 0, Float::sum));

        Shop shop = Shop.convert(shopDTO);
        shop.setDate(LocalDate.now());

        shop = shopRepository.save(shop);
        return ShopDTO.convert(shop);
    }
}

```

6.3 Camada dos controladores (Controllers)

O *controller* desse serviço também segue o padrão das outras APIs, como é possível observar no código. Existem quatro rotas `GET` para fazer buscas

das compras e uma rota POST para cadastrar uma nova compra. A listagem a seguir mostra o código da classe ShopController completo.

```
package com.santana.java.back.end.controller;

import java.util.List;

import lombok.RequiredArgsConstructor;
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.RestController;

import com.santana.java.back.end.dto.ShopDTO;
import com.santana.java.back.end.service.ShopService;

@RestController
@RequiredArgsConstructor
public class ShopController {

    private final ShopService shopService;

    @GetMapping("/shopping")
    public List<ShopDTO> getShops() {
        return shopService.getAll();
    }

    @GetMapping("/shopping/shopByUser/{userIdentifier}")
    public List<ShopDTO> getShops(@PathVariable String
userIdentifier) {
        return shopService.getByUser(userIdentifier);
    }

    @GetMapping("/shopping/shopByDate")
    public List<ShopDTO> getShops(@RequestBody ShopDTO shopDTO) {
        return shopService.getByDate(shopDTO);
    }

    @GetMapping("/shopping/{id}")
    public ShopDTO findById(@PathVariable Long id) {
```



```

        return shopService.findById(id);
    }

    @PostMapping("/shopping")
    @ResponseStatus(HttpStatus.CREATED)
    public ShopDTO newShop(@Valid @RequestBody ShopDTO shopDTO) {
        return shopService.save(shopDTO);
    }
}

```

6.4 Testando os serviços

O principal serviço da *shopping-api* é o `/shopping` com o método HTTP POST, que pode ser acessado pela URL <http://localhost:8082/shopping>. Seu objetivo é salvar novas compras feitas por um usuário.

Seguindo a estrutura da classe `ShopDTO`, uma requisição para esse serviço deve conter um JSON com os atributos `userIdentifier` e uma lista de `items`, sendo que cada item deve conter os atributos `productIdentifier` e `price`. A listagem a seguir mostra um exemplo de um JSON para essa requisição.

```

{
  "userIdentifier": "teste",
  "items": [
    {
      "productIdentifier": "a1",
      "price": "100"
    },
    {
      "productIdentifier": "a2",
      "price": "299"
    },
    {

```

```

        "productIdentifier": "a3",
        "price": "50"
    }
]
}

```

Esse serviço responderá um JSON parecido, confirmando que a compra foi efetuada com sucesso. A única diferença é que na resposta será informado o preço total da compra e a data em que ela foi salva. A listagem a seguir mostra a resposta da requisição.

```

{
  "userIdentifier": "teste",
  "total": 449.0,
  "date": "2019-11-17T21:04:19.828+0000",
  "items": [
    {
      "productIdentifier": "a1",
      "price": 100.0
    },
    {
      "productIdentifier": "a2",
      "price": 299.0
    },
    {
      "productIdentifier": "a3",
      "price": "50"
    }
  ]
}

```

Outro serviço disponível nessa API é o de listar todas as compras de um usuário. Para isso, chamaremos a rota `/shopping/shopByUser/{userIdentifier}`. Se chamarmos a URL <http://localhost:8082/shopping/shopByUser/eduardo> com o método HTTP GET, por exemplo, o retorno serão todas as compras do usuário `eduardo`. A listagem a seguir mostra a resposta para essa requisição.

```
[
  {
    "userIdentifier": "eduardo",
    "total": 399.0,
    "date": "2019-11-17T21:04:51.701+0000",
    "items": [
      {
        "productIdentifier": "p1",
        "price": 100.0
      },
      {
        "productIdentifier": "p2",
        "price": 299.0
      }
    ]
  },
  {
    "userIdentifier": "eduardo",
    "total": 599.0,
    "date": "2019-11-17T21:04:55.324+0000",
    "items": [
      {
        "productIdentifier": "p3",
        "price": 300.0
      },
      {
        "productIdentifier": "p2",
        "price": 299.0
      }
    ]
  }
]
```

Os microsserviços estão funcionando, porém não temos nenhuma consulta realmente complexa no banco de dados. Por exemplo, ainda não podemos recuperar todas as compras de mais de 100 reais em um intervalo de tempo de um mês, ou verificar quantas vendas foram feitas em uma semana e qual o total das vendas. No próximo capítulo, adicionaremos na *shopping-api* buscas mais complexas, nas quais teremos que escrever o código SQL e também o código Java para definir os filtros de uma busca.

CAPÍTULO 7

Buscas mais complexas na shopping-api

Fizemos diversas buscas no banco de dados nas três APIs até agora, como buscar um usuário pelo nome ou pelo CPF, um produto pelo seu identificador, ou compras a partir de uma data. Porém, às vezes precisamos fazer *queries* mais complexas, que tenham filtros dinâmicos, por exemplo, buscar as compras de um mês que tiveram custo total acima de mil reais, ou podemos buscar o total de vendas para um mês utilizando as funções de agregação do SQL, como `count`, `sum` ou `avg`. Não é possível implementar esse tipo de consulta com apenas o nome do método em um repositório; para isso, precisaremos implementar um método que utiliza a *API Criteria*, que permite a construção de consultas dinâmicas no banco de dados.

Para mostrar a utilização desse tipo de consulta, vamos implementar na *shopping-api* duas novas rotas, uma que listará todas as compras feitas, filtrando a consulta com diversos parâmetros, como preço da compra, intervalo de data, ou compras de um usuário específico, e outra que calculará a quantidade, o total e o preço médio das vendas.

7.1 Implementando as consultas

Desta vez, teremos que implementar os comandos SQL, diferentemente do que fizemos na maioria das nossas consultas até agora. Uma das consultas retornará um objeto que ainda não temos, que é o relatório com a contagem, valor total e médio de todas as compras. Para isso, criaremos um DTO simples, chamado `ShopReportDTO`. Esse objeto será usado no retorno de uma das consultas que criaremos.

```
package com.santana.java.back.end.dto;
```

```
import lombok.AllArgsConstructor;
```

```

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ShopReportDTO {

    private Integer count;
    private Double total;
    private Double mean;

}

```

Para implementar as novas consultas, vamos criar uma interface chamada `ReportRepository` , que terá a definição de dois métodos, o `getShopByFilters` e `getReportByDate` , cada um deles recebendo os filtros para a consulta. O primeiro método retornará uma lista de compras que respeite os filtros passados e o segundo, um relatório das compras para um período de tempo.

```

package com.santana.java.back.end.repository;

import java.time.LocalDate;
import java.util.List;

import com.santana.java.back.end.dto.ShopReportDTO;
import com.santana.java.back.end.model.Shop;

public interface ReportRepository {

    public List<Shop> getShopByFilters(
        LocalDate dataInicio,
        LocalDate dataFim,
        Float valorMinimo);

    public ShopReportDTO getReportByDate(
        LocalDate dataInicio,

```

```

        LocalDate dataFim);
    }
}

```

Agora, na interface `ShopRepository` , que desenvolvemos no capítulo anterior, temos que adicionar um `extends` para essa nova interface. Isso serve para que os métodos onde vamos implementar as consultas possam ser injetados sempre que formos utilizar a `ShopRepository` . Não podemos fazer uma classe implementar a interface `ShopRepository` , se não seríamos obrigados a implementar diversos métodos dessa interface como o `findById` , por exemplo.

```

@Repository
public interface ShopRepository
    extends JpaRepository<Shop, Long>, ReportRepository {
    // código desenvolvido no capítulo anterior
}

```

Finalmente podemos implementar as consultas em uma classe concreta. Vamos criar a classe `ReportRepositoryImpl` , que implementa a interface `ReportRepository` . Essa classe deve ter um atributo do tipo `EntityManager` , que é anotado com `@PersistenceContext` . É esse objeto quem faz a conexão com o banco de dados. Como estávamos usando apenas consultas diretamente com o Spring Data, ainda não tínhamos precisado dele; porém, como vamos ter que escrever a consulta nesse caso, precisamos da conexão com o banco de dados diretamente.

```

package com.santana.java.back.end.repository;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

public class ReportRepositoryImpl
    implements ReportRepository {

    @PersistenceContext
    private EntityManager entityManager;
}

```

```
// a implementação das consultas vai aqui  
  
}
```

Precisaremos criar um método para cada uma das consultas que faremos. A começar com a `getShopByFilters`, a primeira coisa que faremos nesse método é montar a consulta SQL e para isso utilizaremos um objeto do tipo `StringBuilder`. Note que o início da consulta é inteiro escrito em uma linha, o `select`, o `from` e o primeiro filtro do `where`. Essa parte da consulta é obrigatória, indicando que o filtro `dataInicio` deve ser obrigatório. Além da `dataInicio`, existem ainda mais dois filtros, a `dataFim` e o `valorMinimo`, porém, eles não são obrigatórios, e por isso existe um `if` para verificar se eles devem fazer parte da consulta ou não.

Note que, nos filtros de data, é chamado o método `atTime`. Isso é feito para converter os dados que foram recebidos com o tipo `LocalDate` para o `LocalDateTime`, que é o tipo usado no banco de dados. Isso evita que o usuário tenha que digitar as horas e os minutos quando for chamar a rota que fará a geração do relatório. Note também que no campo `dataInicio` foi colocado para iniciar na hora 0 e minuto 0, e na `dataFim` foi colocada a hora 23 e o minuto 59, para incluir todas as compras que foram feitas tanto na `dataInicio` quanto na `dataFim`.

Quando o comando SQL está completo, podemos criar um objeto do tipo `Query` usando o método `createNativeQuery` da classe `EntityManager`. Temos também que passar os valores dos filtros que criamos, com o método `setParameter` da classe `Query`. Por fim, executamos o método `getResultList`, que retorna a lista de objetos que são retornados na consulta. A listagem a seguir mostra a implementação completa desse método.

```
@Override  
public List<Shop> getShopByFilters(LocalDate dataInicio, LocalDate  
dataFim, Float valorMinimo) {  
    StringBuilder sb = new StringBuilder();  
    sb.append("select s ");  
    sb.append("from shop s ");
```



```

sb.append("where s.date >= :dataInicio ");

if (dataFim != null) {
    sb.append("and s.date <= :dataFim ");
}

if (valorMinimo != null) {
    sb.append("and s.total <= :valorMinimo ");
}

Query query = entityManager.createQuery(sb.toString());
query.setParameter("dataInicio",
    dataInicio.atTime(0, 0));

if (dataFim != null) {
    query.setParameter("dataFim",
        dataFim.atTime(23, 59));
}

if (valorMinimo != null) {
    query.setParameter("valorMinimo", valorMinimo);
}

return query.getResultList();
}

```

Eu escrevi a consulta inteira em apenas um método para facilitar a explicação, porém, pode ser uma boa prática separá-la em outros dois ou três métodos, por exemplo, um para criar o SQL e outro para criar o objeto query e definir os parâmetros.

O segundo método é o `getReportByDate`, com o qual a consulta é criada da mesma forma que no método anterior, porém, aqui, todos os filtros são obrigatórios. A grande diferença é como pegamos o resultado da consulta, utilizando o método `getSingleResult` da classe `Query`. Quando retornamos apenas um objeto, não é possível fazer a conversão direta para uma classe de entidade, como foi feito no método anterior. Agora precisaremos pegar cada um dos valores retornados e criar o objeto do tipo `ShopReportDTO` manualmente, pois esse é o objeto que será retornado no fim do método.

```

@Override
public ShopReportDTO getReportByDate(LocalDate dataInicio,
LocalDate dataFim) {
    StringBuilder sb = new StringBuilder();
    sb.append("select count(sp.id), sum(sp.total), avg(sp.total)
");
    sb.append("from shopping.shop sp ");
    sb.append("where sp.date >= :dataInicio ");
    sb.append("and sp.date <= :dataFim ");

    Query query = entityManager.createNativeQuery(sb.toString());
    query.setParameter("dataInicio", dataInicio
        .atTime(0, 0));
    query.setParameter("dataFim", dataFim
        .atTime(23, 59));

    Object[] result = (Object[]) query.getSingleResult();
    ShopReportDTO shopReportDTO = new ShopReportDTO();
    shopReportDTO.setCount(((BigInteger) result[0]).intValue());
    shopReportDTO.setTotal((Double) result[1]);
    shopReportDTO.setMean((Double) result[2]);
    return shopReportDTO;
}

```

Uma coisa importante é que a consulta retorna sempre `BigInteger` para consultas com a função `count`, e `Double` para consultas com as funções `sum` e `avg`. Por isso foi necessário fazer o cast para esses tipos antes de definir os valores no objeto `shopReportDTO`.

7.2 Camada de serviços (Service)

A camada de serviços será bastante simples, ela vai apenas chamar um método do repositório, converter os objetos da classe `Shop` para `ShopDTO` e retorná-los para os controladores. Eu adicionei os métodos dos serviços no `ShopService` que desenvolvemos no capítulo anterior, mas, se você preferir, também é possível criar uma nova classe de serviço, por exemplo, uma `ReportService`.

```

public List<ShopDTO> getShopsByFilter(
    LocalDate dataInicio,
    LocalDate dataFim,
    Float valorMinimo) {

    List<Shop> shops =
        reportRepository
            .getShopByFilters(dataInicio, dataFim, valorMinimo);
    return shops
        .stream()
        .map(ShopDTO::convert)
        .collect(Collectors.toList());
}

public ShopReportDTO getReportByDate(
    LocalDate dataInicio,
    LocalDate dataFim) {

    return reportRepository
        .getReportByDate(dataInicio, dataFim);
}

```

7.3 Camada dos controladores (Controllers)

Os controladores também são bem simples. Eu adicionei duas rotas novas: a `/shopping/search` para a busca de compras e a `/shopping/report` para a geração do relatório. Essas rotas são bem parecidas com as criadas nos capítulos anteriores. Elas recebem as requisições com o método `GET` e recebem alguns parâmetros na própria URL. Uma novidade são os parâmetros do tipo `Date`, para os quais devemos definir o padrão com que os dados serão digitados pelo usuário com a anotação `@DateTimeFormat` e o padrão definido no atributo `pattern`.

```

@GetMapping("/shopping/search")
public List<ShopDTO> getShopsByFilter(
    @RequestParam(name = "dataInicio", required=true)
    @DateTimeFormat(pattern = "dd/MM/yyyy") LocalDate

```

```

dataInicio,
    @RequestParam(name = "dataFim", required=false)
    @DateTimeFormat(pattern = "dd/MM/yyyy") LocalDate dataFim,
    @RequestParam(name = "valorMinimo", required=false)
    Float valorMinimo) {
    return shopService.getShopsByFilter(dataInicio, dataFim,
valorMinimo);
}

@GetMapping("/shopping/report")
public ShopReportDTO getReportByDate(
    @RequestParam(name = "dataInicio", required=true)
    @DateTimeFormat(pattern = "dd/MM/yyyy") LocalDate
dataInicio,
    @RequestParam(name = "dataFim", required=true)
    @DateTimeFormat(pattern = "dd/MM/yyyy") LocalDate dataFim)
{
    return shopService.getReportByDate(dataInicio, dataFim);
}

```

Note também a obrigatoriedade dos parâmetros: na primeira rota, apenas a `dataInicio` é obrigatória, já na segunda, tanto a `dataInicio` quanto a `dataFim` são obrigatórias.

7.4 Testando os serviços

Agora podemos testar ambos os serviços desenvolvidos. Primeiro, vamos testar a rota `/shopping/search/`. Ela tem três possíveis parâmetros: as datas de início e fim da busca e o valor mínimo da compra (lembrando que apenas a data de início é obrigatória). Se fizermos a seguinte requisição <http://localhost:8082/shopping/search?dataInicio=01/01/2020&dataFim=01/01/2021&valorMinimo=50>, buscaremos todas as compras efetuadas entre 2020 e 2021 e que tenham o valor mínimo de 50 reais. A resposta para essa requisição terá o seguinte formato:

```
[
  {
    "userIdentifier": "123",
    "total": 100.0,
    "date": "2020-05-31T21:11:51.176+00:00",
    "items": [
      {
        "productIdentifier": "p3",
        "price": 100.0
      }
    ]
  },
  {
    "userIdentifier": "123",
    "total": 100.0,
    "date": "2020-05-31T21:26:48.267+00:00",
    "items": [
      {
        "productIdentifier": "p3",
        "price": 100.0
      }
    ]
  }
]
```

Como o parâmetro `dataInicio` é obrigatório, caso façamos a requisição sem ele (por exemplo, apenas com <http://localhost:8082/shopping/search>), teremos como resposta um erro, como mostrado na listagem a seguir.

```
{
  "timestamp": "2020-05-31T22:20:14.054+00:00",
  "status": 400,
  "error": "Bad Request",
  "message": "",
  "path": "/shopping/search"
}
```

A chamada para o segundo serviço é parecida. Por exemplo, se fizermos a requisição <http://localhost:8082/shopping/report?>

[dataInicio=01/01/2020&dataFim=01/01/2021](#), faremos o relatório de todas as compras efetuadas entre 2020 e 2021, e o resultado para essa busca será:

```
{
  "count": 2,
  "total": 200.0,
  "mean": 100.0
}
```

As funcionalidades básicas dos microserviços estão implementadas. Todos possuem seu próprio conjunto de serviços e seu banco de dados. Porém, eles ainda funcionam independentes um dos outros — e essa não é a nossa ideia. É importante que, para o cadastro de uma compra, todos os dados dos usuários e dos produtos estejam corretos. Para isso, será necessário comunicar a *shopping-api* com os outros dois microserviços. No próximo capítulo, faremos essa implementação e, assim, as funcionalidades dos microserviços estarão completas.

CAPÍTULO 8

Comunicação entre os serviços

Até agora as aplicações estão funcionando isoladamente, porém, para o cadastro da compra, é necessário validar se o usuário e os produtos selecionados existem. Também é preciso recuperar o preço de cada item para calcular o valor total da compra. Para isso, teremos que chamar a *user-api* e a *product-api* a partir da *shopping-api*. Além disso, teremos que reutilizar os DTOs de um microserviço em outro; por exemplo, a *shopping-api* precisará utilizar as classes `UserDTO` e `ProductDTO`, que hoje estão nos projetos *user-api* e *product-api*, então vamos ter que criar um novo projeto que contenha as partes comuns entre as aplicações.

8.1 Reutilizando DTOs

Para fazer a comunicação entre os serviços, o *shopping-api* precisará de todos os DTOs definidos na *user-api* e na *product-api*. Temos duas formas de fazer isso. A mais simples e direta seria copiar o código dessas classes na *shopping-api*. Porém, isso não seria o ideal, pois o código ficaria duplicado nos diferentes projetos, o que dificulta a manutenção e a evolução da aplicação. A melhor forma de fazer isso é criar um projeto Java que contenha apenas os DTOs de todos os microserviços e importá-lo em todas as nossas APIs. Com o Maven, isso é bem fácil de fazer.

O primeiro passo é criar um novo projeto Java simples — vamos chamá-lo de *shopping-client*. Esse projeto será um JAR simples, que será importado nos outros. O arquivo `pom.xml` desse JAR deve ser definido como a listagem a seguir.

```
<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
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>
<groupId>com.santana.java.back.end</groupId>
<artifactId>shopping-client</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.0</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.24</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

</project>

```

As únicas dependências que foram adicionadas são o `spring-boot-starter-validation` (pois as anotações com as validações do Spring ficam nos DTOs) e o `lombok`. Depois de configurado o projeto, copiamos todos os DTOs de todos os microsserviços anteriores para esse novo projeto, que são o `UserDTO` do projeto *user-api*, o `ProductDTO` e o `CategoryDTO` da *product-api*, e o `ShopDTO` e o `ItemDTO` da *shopping-api*.

Você verá que em todos os DTOs teremos problemas com os métodos `convert` que criamos anteriormente. Isso porque esses métodos dependem também das classes do `Model`, que continuarão dentro dos projetos anteriores. Isso é importante porque a estrutura do banco de dados deve ficar apenas dentro do microsserviço. Para resolver o problema, basta deletar os métodos `convert` de todos os DTOs. Para que o projeto

`shopping-client` fique disponível para ser importado em outros, execute o comando `mvn clean install` para que o Maven crie a biblioteca desse projeto e o disponibilize localmente.

8.2 Mudanças nos projetos

Depois de criar o projeto que contém os DTOs, precisamos importá-lo nos microsserviços. Para fazer isso, basta adicionar a dependência do *shopping-client* no `pom.xml` de todos os outros projetos. A listagem a seguir mostra como adicionar essa dependência.

```
<dependency>
  <groupId>com.santana.java.back.end</groupId>
  <artifactId>shopping-client</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

Além disso, cada projeto terá uma classe nova, chamada `DTOConverter`, que conterá os conversores necessários (os que excluímos dos DTOs algumas linhas atrás). Por exemplo, a *product-api* precisará converter os DTOs `CategoryDTO` e `ProductDTO` em entidades, por isso a classe `DTOConverter` terá esses dois conversores.

```
package com.santana.java.back.end.converter;

import com.santana.java.back.end.dto.CategoryDTO;
import com.santana.java.back.end.dto.ProductDTO;
import com.santana.java.back.end.model.Category;
import com.santana.java.back.end.model.Product;

public class DTOConverter {

    public static CategoryDTO convert(Category category) {
        CategoryDTO categoryDTO = new CategoryDTO();
        categoryDTO.setId(category.getId());
        categoryDTO.setNome(category.getNome());
        return categoryDTO;
    }
}
```

```

    }

    public static ProductDTO convert(Product product) {
        ProductDTO productDTO = new ProductDTO();
        productDTO.setNome(product.getNome());
        productDTO.setPreco(product.getPreco());
        if (product.getCategory() != null) {
            productDTO.setCategory(
                DTOConverter.convert(product.getCategory()));
        }
        return productDTO;
    }
}

```

A *user-api* precisará converter apenas a classe `UserDTO` , por isso apenas esse conversor é necessário.

```

package com.santana.java.back.end.converter;

import com.santana.java.back.end.dto.UserDTO;
import com.santana.java.back.end.model.User;

public class DTOConverter {

    public static UserDTO convert(User user) {
        UserDTO userDTO = new UserDTO();
        userDTO.setNome(user.getNome());
        userDTO.setEndereco(user.getEndereco());
        userDTO.setCpf(user.getCpf());
        return userDTO;
    }
}

```

Finalmente, a *shopping-api* precisará converter os DTOs `ShopDTO` e `ItemDTO` .

```

package com.santana.java.back.end.converter;

import java.util.stream.Collectors;

```

```

import com.santana.java.back.end.dto.ItemDTO;
import com.santana.java.back.end.dto.ShopDTO;
import com.santana.java.back.end.model.Item;
import com.santana.java.back.end.model.Shop;

public class DTOConverter {

    public static ItemDTO convert(Item item) {
        ItemDTO itemDTO = new ItemDTO();
        itemDTO.setProductIdentifier(
            item.getProductIdentifier());
        itemDTO.setPrice(item.getPrice());
        return itemDTO;
    }

    public static ShopDTO convert(Shop shop) {
        ShopDTO shopDTO = new ShopDTO();
        shopDTO.setUserIdentifier(shop.getUserIdentifier());
        shopDTO.setTotal(shop.getTotal());
        shopDTO.setDate(shop.getDate());
        shopDTO.setItems(shop
            .getItems()
            .stream()
            .map(DTOConverter::convert)
            .collect(Collectors.toList()));
        return shopDTO;
    }
}

```

8.3 Comunicação entre os serviços

Agora vamos implementar a comunicação entre os serviços. Basicamente, quando recebermos uma nova compra no serviço `/shop`, verificaremos se o usuário existe (pelo CPF) e se o produto existe (pelo `productIdentifier`). A *product-api* também enviará o preço do produto,

caso ele exista. Se o usuário ou o produto não existir, a compra não será efetuada e um erro será enviado para o usuário. Nenhuma mudança será necessária na *product-api* e na *user-api*, pois os serviços necessários já foram criados nos capítulos anteriores.

WebClient

Para fazer a comunicação entre os serviços, utilizaremos o WebClient, uma classe do Spring WebFlux que usa algumas ideias de programação reativa, embora ela funcione normalmente com aplicações Spring que não usam programação reativa. Basicamente, o WebClient é uma classe do framework Spring que facilita a criação de clientes REST na linguagem Java. Usando essa classe, a chamada para um serviço fica bastante simplificada, pois ela permite a utilização dos diferentes métodos HTTP (GET , PUT , DELETE ...) e também faz a conversão do JSON de requisição ou de resposta para objetos Java automaticamente.

SPRING WEBFLUX

O WebFlux é uma versão do Spring que usa programação reativa para a construção de aplicações web e serviços REST. Ele ainda não é tão usado como o Spring Web tradicional, mas algumas partes desse framework, como o próprio WebClient, vêm ganhando bastante popularidade nos últimos anos.

RESTTEMPLATE

Até poucas versões atrás do Spring, e na primeira edição deste livro, a classe utilizada para a comunicação entre os serviços era a RestTemplate, mas ela foi depreciada e o seu uso deve ser evitado, pois ela pode ser removida do framework nas próximas versões.

Para utilizar o WebClient na *shopping-api*, será necessário acrescentar a dependência `spring-boot-starter-webflux` no `pom.xml` do projeto. O

código a seguir mostra a dependência que deve ser adicionada.

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

Comunicação com a user-api

Implementar a comunicação entre os serviços será bastante simples. Criaremos uma nova classe `Service` para cada uma das APIs com que faremos a comunicação. A primeira classe será a `UserService`, que fará a comunicação entre a *shopping-api* e a *user-api*. Essa classe contém o método `getUserByCpf`, que recebe como parâmetro o CPF de um usuário e, utilizando a classe `WebClient`, faz a chamada para a *user-api*.

```
package com.santana.java.back.end.service;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;

import com.santana.java.back.end.dto.UserDTO;
import com.santana.java.back.end.exception.UserNotFoundException;
import reactor.core.publisher.Mono;

@Service
public class UserService {

    private String userApiURL = "http://localhost:8080";

    public UserDTO getUserByCpf(String cpf) {
        try {
            WebClient webClient = WebClient.builder()
                .baseUrl(userApiURL)
                .build();
```

```

        Mono<UserDTO> user = webClient.get()
            .uri("/user/" + cpf + "/cpf")
            .retrieve()
            .bodyToMono(UserDTO.class);

        return user.block();
    } catch (Exception e) {
        throw new RuntimeException("User not found");
    }
}
}

```

Para a utilização do `webClient`, são necessários dois trechos de código: o primeiro é o que cria o objeto do tipo `WebClient` e que possui os dados básicos do servidor em que será feita a conexão, que são a URL e a porta. Depois, usando o objeto da classe `WebClient`, é feita a chamada para o serviço específico — no nosso caso, o `/user/{cpf}/cpf` da *user-api*. Essa chamada retorna um objeto da classe `UserDTO` ou um erro indicando que o usuário não foi encontrado. Note que, internamente, o Spring já converteu o JSON para um objeto `UserDTO`, o que facilita bastante a implementação de clientes REST com esse framework. Note que o endereço para a chamada do *user-api* está *hardcoded* no método. Obviamente, essa não é a melhor solução, mas vamos resolver isso quando formos implantar os microsserviços no Kubernetes.

Comunicação com a *product-api*

A comunicação com a *product-api* é bem parecida com a comunicação com a *user-api*. Também criaremos uma classe chamada `ProductService` que conectará ao serviço que retorna os dados de um produto a partir de um identificador. Essa implementação está no método `getProductByIdentifier`.

```

package com.santana.java.back.end.service;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

```

```

import com.santana.java.back.end.dto.ProductDTO;
import
com.santana.java.back.end.exception.ProductNotFoundException;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class ProductService {

    private String productApiURL = "http://localhost:8081";

    public ProductDTO getProductByIdentifier(String
productIdentifier) {

        try {
            WebClient webClient = WebClient.builder()
                .baseUrl(productApiURL)
                .build();

            Mono<ProductDTO> product = webClient.get()
                .uri("/product/" + productIdentifier)
                .retrieve()
                .bodyToMono(ProductDTO.class);

            return product.block();
        } catch (Exception e) {
            throw new RuntimeException("Product not found");
        }

    }

}

```

Depois, os mesmos passos são necessários para a chamada ao serviço que verificará se um produto existe ou não.

Mudanças no serviço de inclusão de compra

Finalmente, mudaremos o serviço que salva uma compra no banco de dados. Na versão dos capítulos anteriores, a compra estava sendo salva diretamente no banco, sem validar se os usuários e produtos existiam. Também estávamos utilizando um valor para os produtos passados na chamada ao serviço, mas agora mudaremos isso para utilizar o valor que está salvo na *product-api*.

Essas mudanças serão feitas na classe `ShopService`. A primeira será adicionar as referências para a `ProductService` e `UserService` na classe com a anotação `@Autowired`. Além disso, no método `save`, vamos chamar os métodos criados anteriormente para verificar se os usuários e produtos existem.

Verificar se o usuário existe é bem simples, já que basta chamar o método `getUserByCpf` passando o CPF passado na requisição para o serviço de compra. Caso o usuário não seja encontrado, a *user-api* responderá `null`; já se encontrar, retornará todos os dados dele.

```
package com.santana.java.back.end.service;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.santana.java.back.end.converter.DTOConverter;
import com.santana.java.back.end.dto.ItemDTO;
import com.santana.java.back.end.dto.ProductDTO;
import com.santana.java.back.end.dto.ShopDTO;
import com.santana.java.back.end.model.Shop;
import com.santana.java.back.end.repository.ShopRepository;

@Service
public class ShopService {

    @Autowired
```



```

private ShopRepository shopRepository;

@Autowired
private ProductService productService;

@Autowired
private UserService userService;

// ...
// os outros métodos não sofreram alterações

public ShopDTO save(ShopDTO shopDTO) {

    if (userService
        .getUserByCpf(shopDTO.getUserIdentifier()) == null) {
        return null;
    }

    if (!validateProducts(shopDTO.getItems())) {
        return null;
    }

    shopDTO.setTotal(shopDTO.getItems()
        .stream()
        .map(x -> x.getPrice())
        .reduce((float) 0, Float::sum));

    Shop shop = Shop.convert(shopDTO);
    shop.setDate(LocalDate.now());

    shop = shopRepository.save(shop);
    return DTOConverter.convert(shop);
}

private boolean validateProducts(List<ItemDTO> items) {
    for (ItemDTO item : items) {
        ProductDTO productDTO = productService
            .getProductByIdentifier(
                item.getProductIdentifier());
        if (productDTO == null) {
            return false;
        }
    }
}

```

```
        }
        item.setPrice(productDTO.getPreco());
    }
    return true;
}
}
```

A verificação dos produtos é um pouco mais complicada, já que na requisição é passada uma lista de produtos. Precisamos também salvar o preço do produto que foi retornado na lista de produtos. Para fazer essas duas coisas, foi criado o método `validateProducts`. Note que esse método itera sobre toda a lista de produtos verificando um a um se eles existem, e se um deles não existir, o método retorna `false`.

Agora temos os nossos três microsserviços com a funcionalidade completa e, mais importante, eles estão integrados, o que permite verificar os dados dos produtos e do comprador quando um usuário fizer uma compra na aplicação. O exemplo desenvolvido até aqui, apesar de relativamente simples, mostra as principais vantagens da arquitetura de microsserviços, pois temos três serviços independentes, mas que podem ser integrados para a criação de uma funcionalidade mais complexa, como o serviço de compra.

O próximo passo na implementação da nossa aplicação será fazer um tratamento melhor para os erros que podem ocorrer. Por exemplo, o usuário enviar um código de produto inválido em uma compra ou enviar o CPF de um usuário que não está cadastrado no sistema.

CAPÍTULO 9

Exceções

Em uma API REST, é interessante utilizar os códigos HTTP para o retorno das chamadas corretamente para facilitar a integração entre diferentes aplicações. Por exemplo, o código 200 indica que a requisição foi executada com sucesso; o 201 indica que um recurso foi criado no servidor; o 404 indica que um recurso não foi encontrado e o 500 indica um erro genérico no servidor. Assim, neste capítulo, adicionaremos o tratamento de exceções nos serviços para o retorno de uma mensagem explicativa para o usuário e o código HTTP correto.

9.1 Criando as exceções

As exceções poderão ocorrer em todos os microsserviços. O erro de usuário não encontrado pode ocorrer tanto na *user-api* quanto na *shopping-api*, e o mesmo vale para o erro de um produto não encontrado, que pode ocorrer tanto na *product-api* quanto na *shopping-api*. Então, criaremos as exceções no projeto *shopping-client* para que as classes sejam compartilhadas por todos os microsserviços.

Criar uma exceção é bastante simples, basta criar uma nova classe e estender as classes `Exception` ou `RuntimeException`. No nosso caso, usaremos a classe `RuntimeException`, que é a mais indicada para erros que ocorrem normalmente por dados inválidos, como um usuário buscar por um produto ou usuário que não existe. A próxima listagem mostra a implementação da exceção `UserNotFoundException`.

```
package com.santana.java.back.end.exception;

public class UserNotFoundException extends RuntimeException {

}
```

A mesma coisa acontece com o caso de o usuário tentar pesquisar por um produto que não existe, então criaremos a classe `ProductNotFoundException`. E para quando o usuário informar uma categoria inexistente na hora de criar um novo produto, criaremos a classe `CategoryNotFoundException`.

```
package com.santana.java.back.end.exception;

public class ProductNotFoundException extends RuntimeException {

}

package com.santana.java.back.end.exception;

public class CategoryNotFoundException extends RuntimeException {

}
```

9.2 Implementando as exceções na user-api

Agora, é possível utilizar essa exceção em diversos lugares do código dos microsserviços. Vamos considerar o caso de um usuário fazer uma busca por usuário com um CPF inválido, por exemplo. A listagem a seguir mostra essa mudança nos métodos da classe de serviço desenvolvida anteriormente.

```
public UserDTO findByCpf(String cpf) {
    User user = userRepository.findByCpf(cpf);
    if (user != null) {
        return UserDTO.convert(user);
    }
    throw new UserNotFoundException();
}
```

Em vez de apenas retornar `null`, esses métodos retornarão uma exceção. Porém, isso ainda não é o suficiente, porque o erro retornado será um "Internal Server Error" com status 500, o que não é uma mensagem muito

clara para o usuário. A listagem a seguir mostra a mensagem de erro padrão quando uma exceção do tipo `UserNotFoundException` é retornada.

```
{
  "timestamp": "2019-10-17T20:23:31.045+0000",
  "status": 500,
  "error": "Internal Server Error",
  "message": "No message available",
  "path": "/user/cpf/123"
}
```

O ideal é mostrar uma mensagem mais amigável para o usuário, que contenha um código mais condizente com o erro ocorrido. Nesse caso, seria o erro 404, que indica que um recurso não foi encontrado no servidor. Para isso, criaremos uma classe que também é um DTO chamado `ErrorDTO`, pois será utilizado para enviar os dados no serviço. O melhor lugar para criar essa classe é também no *shopping-client*, assim ela já fica disponível para todos os outros projetos.

```
package com.santana.java.back.end.dto;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.LocalDateTime;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class ErrorDTO {
    private int status;
    private String message;
    private LocalDateTime timestamp;
}
```

Agora criamos uma classe que será executada sempre que uma exceção for lançada. Essa classe deve ter a anotação `@ControllerAdvice`, que informa

o pacote dos controllers da aplicação. Ela possui um método para cada tipo de exceção que o sistema pode gerar. No exemplo a seguir, vamos capturar a exceção `UserNotFoundException` no método `handleUserNotFound`.

```
package com.santana.java.back.end.exception.advice;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import com.santana.java.back.end.dto.ErrorDTO;
import com.santana.java.back.end.exception.UserNotFoundException;

@ControllerAdvice(basePackages =
"com.santana.java.back.end.controller")
public class UserControllerAdvice {

    @ResponseBody
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(UserNotFoundException.class)
    public ErrorDTO handleUserNotFound(
        UserNotFoundException userNotFoundException) {
        ErrorDTO errorDTO = new ErrorDTO();
        errorDTO.setStatus(HttpStatus.NOT_FOUND.value());
        errorDTO.setMessage("Usuário não encontrado.");
        errorDTO.setTimestamp(LocalDateTime.now());
        return errorDTO;
    }
}
```

Note alguns atributos importantes nas anotações. O atributo `basePackages` da anotação `@ControllerAdvice` indica que ela deve verificar as exceções retornadas em todos os controllers. O valor `HttpStatus.NOT_FOUND` passado na anotação `@ResponseStatus` indica que deve ser retornado o erro 404 como status da resposta. A exceção `UserNotFoundException.class` na

anotação `@ExceptionHandler` indica que esse método deve capturar esse tipo de exceções. Finalmente, a anotação `@ResponseBody` define que o retorno desse método será retornado no corpo da resposta.

Com esse método implementado, quando um serviço lançar a exceção `UserNotFoundException`, a seguinte resposta será exibida para o usuário:

```
{
  "status": 404,
  "message": "Usuário não encontrado.",
  "timestamp": "2019-10-18T17:33:45.996+0000"
}
```

9.3 Implementando as exceções na *product-api*

Na *product-api*, a implementação será bem parecida. Basicamente, quando um produto não for encontrado no banco de dados, retornaremos uma `ProductNotFoundException` — como no exemplo a seguir, nos métodos `findByProductIdentifier` e `delete`:

```
public ProductDTO findByProductIdentifier(
    String productIdentifier) {

    Product product = productRepository
        .findByProductIdentifier(productIdentifier);
    if (product != null) {
        return DTOConverter.convert(product);
    }
    throw new ProductNotFoundException();
}

public ProductDTO delete(long ProductId)
```

```

        throws ProductNotFoundException {

        Optional<Product> Product =
            productRepository.findById(ProductId);
        if (Product.isPresent()) {
            productRepository.delete(Product.get());
        }
        throw new ProductNotFoundException();
    }
}

```

Além disso, se na hora da criação de um novo produto o usuário informar uma categoria que não existe, será retornado o erro

`CategoryNotFoundException` . Para isso faremos uma pequena alteração no método `save` da classe `ProductService` :

```

public ProductDTO save(ProductDTO productDTO) {
    Boolean existsCategory = categoryRepository
        .existsById(productDTO.getCategory().getId());
    if (!existsCategory) {
        throw new CategoryNotFoundException();
    }
    Product product = productRepository
        .save(Product.convert(productDTO));
    return DTOConverter.convert(product);
}

```

Agora, estamos verificando se uma categoria existe antes de salvar o produto e, se ela não existir, retornaremos a exceção

`CategoryNotFoundException` . Utilizamos um método interessante do Spring Data nessa implementação, o `existsById` , que verifica se um determinado Id existe no banco de dados, retornando apenas `true` ou `false` . Esse método é útil quando só é necessário saber se o objeto existe, mas ele não será utilizado. No nosso caso, verificamos se uma categoria existe antes de tentar cadastrar um produto.

Também teremos uma classe anotada com `@ControllerAdvice` indicando como deve ser o tratamento para as exceções `ProductNotFoundException` e `CategoryNotFoundException` .


```

package com.santana.java.back.end.exception.advice;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import com.santana.java.back.end.dto.ErrorDTO;
import
com.santana.java.back.end.exception.ProductNotFoundException;

@ControllerAdvice(
    basePackages = "com.santana.java.back.end.controller")
public class ProductControllerAdvice {

    @ResponseBody
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(ProductNotFoundException.class)
    public ErrorDTO handleUserNotFound(
        ProductNotFoundException userNotFoundException) {

        ErrorDTO errorDTO = new ErrorDTO();
        errorDTO.setStatus(HttpStatus.NOT_FOUND.value());
        errorDTO.setMessage("Produto não encontrado.");
        errorDTO.setTimestamp(LocalDateTime.now());
        return errorDTO;
    }

    @ResponseBody
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(CategoryNotFoundException.class)
    public ErrorDTO handleCategoryNotFound(
        CategoryNotFoundException categoryNotFoundException) {

        ErrorDTO errorDTO = new ErrorDTO();
        errorDTO.setStatus(HttpStatus.NOT_FOUND.value());
        errorDTO.setMessage("Categoria não encontrada.");
        errorDTO.setTimestamp(LocalDateTime.now());
    }
}

```

```

        return errorDTO;
    }
}

```

Com esse método implementado, quando um serviço lançar a exceção `ProductNotFoundException`, a seguinte resposta será exibida para o usuário:

```

{
  "status": 404,
  "message": "Produto não encontrado.",
  "timestamp": "2019-10-18T17:33:45.996+0000"
}

```

E quando uma `CategoryNotFoundException` for lançada, a resposta será:

```

{
  "status": 404,
  "message": "Categoria não encontrada.",
  "timestamp": "2019-10-18T17:33:45.996+0000"
}

```

Outro erro que indiquei no capítulo 5 é quando não é informado um campo obrigatório na hora de salvar um novo produto. Quando isso acontece, o Spring retorna o erro `MethodArgumentNotValidException`, por isso também podemos adicionar um método na classe `ProductControllerAdvice` que trata esse erro. Nesse caso, retornaremos uma mensagem indicando quais campos possuem valores inválidos.

```

@ResponseBody
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public ErrorDTO processValidationError(
    MethodArgumentNotValidException ex) {

    ErrorDTO errorDTO = new ErrorDTO();
    errorDTO.setStatus(HttpStatus.BAD_REQUEST.value());
    BindingResult result = ex.getBindingResult();
    List<FieldError> fieldErrors = result.getFieldErrors();
    StringBuilder sb =

```

```

        new StringBuilder("Valor inválido para o(s) campo(s):");
    for (FieldError fieldError : fieldErrors) {
        sb.append(" ");
        sb.append(fieldError.getField());
    }
    errorDTO.setMessage(sb.toString());
    errorDTO.setTimestamp(LocalDateTime.now());
    return errorDTO;
}

```

Assim, se tentarmos salvar um produto sem alguns campos, por exemplo, o preço e o identificador do produto, como no seguinte JSON:

```

{
  "nome": "TV 2",
  "category": {
    "id": 1
  }
}

```

Obteremos a seguinte resposta:

```

{
  "status": 400,
  "message": "Valor inválido para o(s) campo(s):  
productIdentifier preco",
  "timestamp": "2020-06-14T16:22:33.661+00:00"
}

```

9.4 Implementando as exceções na shopping-api

Na *shopping-api*, também poderão ser lançadas as mesmas duas exceções: quando um usuário não cadastrado tentar fazer uma compra ou quando for passado um identificador de um produto inválido. As exceções serão lançadas logo após a chamada para os outros serviços. Agora, como estamos tratando os erros corretamente, tanto o serviço para encontrar um produto pelo identificador quanto o de encontrar um usuário pelo CPF estão retornando o erro 404 quando os recursos não são encontrados.

As listagens a seguir mostram as modificações feitas no código desenvolvido no capítulo 8. A principal diferença é que, em vez de usar a exceção genérica `RuntimeException`, usamos as exceções que indicam exatamente o erro que aconteceu nos serviços. O código a seguir mostra as mudanças no método `getProductByIdentifier` da classe `ProductService`.

```
public ProductDTO getProductByIdentifier(String productIdentifier)
{
    try {
        WebClient webClient = WebClient.builder()
            .baseUrl(productApiURL)
            .build();

        Mono<ProductDTO> product = webClient.get()
            .uri("/product/" + productIdentifier)
            .retrieve()
            .bodyToMono(ProductDTO.class);

        return product.block();
    } catch (Exception e) {
        throw new ProductNotFoundException();
    }
}
```

O código a seguir mostra as mudanças no método `getUserByCpf` da classe `UserService`.

```
public UserDTO getUserByCpf(String cpf) {
    try {
        WebClient webClient = WebClient.builder()
            .baseUrl(userApiURL)
            .build();

        Mono<UserDTO> user = webClient.get()
            .uri("/user/" + cpf + "/cpf")
            .retrieve()
            .bodyToMono(UserDTO.class);
    }
```

```

        return user.block();
    } catch (Exception e) {
        throw new UserNotFoundException();
    }
}

```

Assim como nos serviços anteriores, também temos que criar a classe anotada com o `@ControllerAdvice` com as duas possíveis exceções que podem ocorrer nesse serviço.

```

package com.santana.java.back.end.exception.advice;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import com.santana.java.back.end.dto.ErrorDTO;
import
com.santana.java.back.end.exception.ProductNotFoundException;
import com.santana.java.back.end.exception.UserNotFoundException;

@ControllerAdvice(
    basePackages = "com.santana.java.back.end.controller")
public class ShoppingControllerAdvice {

    @ResponseBody
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(ProductNotFoundException.class)
    public ErrorDTO handleUserNotFound(
        ProductNotFoundException userNotFoundException) {
        ErrorDTO errorDTO = new ErrorDTO();
        errorDTO.setStatus(HttpStatus.NOT_FOUND.value());
        errorDTO.setMessage("Produto não encontrado.");
        errorDTO.setTimestamp(LocalDateTime.now());
        return errorDTO;
    }
}

```

```

@ResponseBody
@ResponseStatus(HttpStatus.NOT_FOUND)
@ExceptionHandler(UserNotFoundException.class)
public ErrorDTO handleUserNotFound(
    UserNotFoundException userNotFoundException) {
    ErrorDTO errorDTO = new ErrorDTO();
    errorDTO.setStatus(HttpStatus.NOT_FOUND.value());
    errorDTO.setMessage("Usuário não encontrado.");
    errorDTO.setTimestamp(LocalDateTime.now());
    return errorDTO;
}
}

```

Agora, com os erros implementados, vamos desenvolver um mecanismo simples de autenticação para um usuário poder efetivar a sua compra. Criaremos uma chave de acesso para cada usuário e, quando ele for fazer uma compra, ele deverá passar a chave no serviço.

CAPÍTULO 10

Autenticação

Implementaremos neste capítulo um mecanismo simples de autenticação no serviço de compras. Existem várias formas de implementar mecanismos para verificar se o usuário pode ou não executar um serviço. Uma das maneiras mais utilizadas em uma aplicação REST é a verificação de uma chave de acesso que pode ser incluída no cabeçalho de uma requisição.

Para essa implementação, faremos duas mudanças principais: a primeira será na *user-api*, onde mudaremos o serviço de inclusão de usuários para a criação de uma chave utilizando um algoritmo de geração de chaves aleatórias. Um dos métodos mais usados para isso é o que gera UUID (Universally Unique Identifier), uma string alfanumérica com probabilidade quase zero de geração de valores iguais.

A segunda mudança é na chamada para a rota que insere uma nova compra no banco de dados. Atualmente essa rota recebe no corpo da requisição o usuário e a lista de produtos da compra. Além desses dados, adicionaremos no cabeçalho da requisição um parâmetro que será a chave gerada para o usuário. Ela será utilizada para autenticar o usuário na aplicação.

10.1 Gerando o UUID na user-api

Para salvar o UUID para cada usuário, inicialmente adicionaremos o campo `key` na tabela `user` e nas classes `User` e `UserDTO`. A listagem a seguir mostra a migração para adicionar o campo na tabela.

```
alter table users.user add column key varchar(100);
```

Vale lembrar aqui que a migração deve ser salva na pasta `src/main/resources/db`, assim como fizemos nos capítulos 4, 5 e 6. O nome do arquivo deve ser `v2__Add_key_column_user.sql`, sendo que a parte `v2__` é obrigatória, indicando que essa será a segunda migração a ser

executada no banco de dados; o restante do nome do arquivo você pode alterar.

A segunda alteração é na classe `User`, que é a classe que contém exatamente os mesmos campos da tabela. A listagem a seguir mostra o novo código dessa classe com o campo adicionado. Note que o campo também foi adicionado no método `convert`.

```
package com.santana.java.back.end.model;

import java.time.LocalDateTime;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

import com.santana.java.back.end.dto.UserDTO;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity(name="user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nome;
    private String cpf;
    private String endereco;
    private String key;
    private String email;
    private String telefone;
    private LocalDateTime dataCadastro;
```



```

        public static User convert(UserDTO userDTO) {
            User user = new User();
            user.setNome(userDTO.getNome());
            user.setEndereco(userDTO.getEndereco());
            user.setCpf(userDTO.getCpf());
            user.setKey(userDTO.getKey());
            user.setEmail(userDTO.getEmail());
            user.setTelefone(userDTO.getTelefone());
            user.setDataCadastro(userDTO.getDataCadastro());
            return user;
        }
    }
}

```

Agora vamos adicionar o campo na classe UserDTO .

```

package com.santana.java.back.end.dto;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.LocalDateTime;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserDTO {
    private String nome;
    private String cpf;
    private String endereco;
    private String key;
    private String email;
    private String telefone;
    private LocalDateTime dataCadastro;
}

```

E também no método que faz a conversão da classe `User` para a classe `UserDTO` .

```
package com.santana.java.back.end.converter;

import com.santana.java.back.end.dto.UserDTO;
import com.santana.java.back.end.model.User;

public class DTOConverter {

    public static UserDTO convert(User user) {
        UserDTO userDTO = new UserDTO();
        userDTO.setNome(user.getNome());
        userDTO.setEndereco(user.getEndereco());
        userDTO.setCpf(user.getCpf());
        userDTO.setKey(user.getKey());
        return userDTO;
    }

}
```

Esse campo será gerado automaticamente sempre que criarmos um novo usuário na aplicação. Gerar um UUID em Java é bastante simples, basta chamar o método `randomUUID` da classe `UUID` . Faremos a chamada no método `save` da classe `UserService` , como mostrado na listagem a seguir.

```
public UserDTO save(UserDTO userDTO) {
    userDTO.setKey(UUID.randomUUID().toString());
    userDTO.setDataCadastro(LocalDate.now());

    User user = userRepository.save(User.convert(userDTO));
    return DTOConverter.convert(user);
}
```

As alterações na *user-api* estão prontas. Se você quiser verificar o UUID, faça a chamada para o serviço de criação de um usuário. A chamada continua a mesma, um POST para a URL `http/localhost:8081/user`, com a única diferença na resposta, que agora terá também o campo `key` , como no exemplo a seguir.

```
{
  "nome": "Eduardo",
  "cpf": "123",
  "endereco": "Rua abc",
  "key": "28805e07-f886-48ec-89dc-48614c58402c",
  "email": "eduardo@email.com",
  "telefone": "1234",
  "dataCadastro": "2022-10-01T12:24:26.324566"
}
```

Além disso, o campo `key` será retornado em todas as rotas GET que retornam as informações dos usuários.

Agora, para fazer o login, na rota em que recebemos o CPF do usuário, vamos também enviar a chave do usuário. Enviaremos essa chave como um parâmetro de requisição; assim, a chamada para a rota será

`http://localhost:8080//user/{cpf}/cpf?key={key}` . Para essa mudança, vamos começar pelo repositório. Antes a busca era feita apenas pelo CPF, agora vamos fazer a busca pelo CPF e pela chave, então basta mudar o nome do método de `findByCpf` para `findByCpfAndKey` .

```
User findByCpfAndKey(String cpf, String key);
```

Na camada de serviço, na classe `UserService` , apenas precisamos passar a chave como parâmetro para o método `findByCpf` e mudar o nome do método que chamávamos anteriormente para o novo método do repositório.

```
public UserDTO findByCpf(String cpf, String key) {
    User user = userRepository.findByCpfAndKey(cpf, key);
    if (user != null) {
        return DTOConverter.convert(user);
    }
    throw new UserNotFoundException();
}
```

Finalmente, na camada de `controller` , passamos para o método a chave que será passada como parâmetro. Para fazer isso, utilizamos a anotação `@RequestParam` , que indica que o atributo será passado na URL. Como esse

atributo será usado para a busca do usuário, ele é obrigatório, por isso passamos o valor `true` no atributo `required` da anotação.

```
@GetMapping("/user/{cpf}/cpf")
UserDTO findByCpf(
    @RequestParam(name="key", required=true) String key,
    @PathVariable String cpf) {
    return userService.findByCpf(cpf, key);
}
```

10.2 Validando o usuário na shopping-api

Na *shopping-api* agora temos que receber a chave do usuário quando recebermos uma requisição na rota `POST shopping`. Existem várias formas de passar essa informação, mas como ela é relacionada à autenticação, eu passarei como um campo no cabeçalho (header) da requisição, chamado de `key`. Para receber um valor no cabeçalho, basta criar um parâmetro no método do *controller* e anotá-lo com `@RequestHeader`. Essa anotação possui dois atributos principais: o nome do parâmetro e se ele é obrigatório ou não. Se ele for obrigatório e o valor não for passado, será retornado um erro para o usuário. A listagem a seguir mostra a adição desse parâmetro na *shopping-api*.

```
@PostMapping("/shopping")
public ShopDTO newShop(
    @RequestHeader(name = "key", required=true) String key,
    @RequestBody ShopDTO shopDTO) {
    return shopService.save(shopDTO, key);
}
```

As alterações na camada de serviço são bastante simples, basta adicionar um novo parâmetro, que chamamos `key`. Esse valor será passado também para o `UserService` e assim será enviado para a *user-api* quando formos validar o usuário.

```
public ShopDTO save(ShopDTO shopDTO, String key) {
    UserDTO userDTO = userService
```

```

        .getUserByCpf(shopDTO.getUserIdentifier(), key);
    validateProducts(shopDTO.getItems());

    shopDTO.setTotal(shopDTO.getItems()
        .stream()
        .map(x -> x.getPrice())
        .reduce((float) 0, Float::sum));

    Shop shop = Shop.convert(shopDTO);
    shop.setDate(LocalDate.now());

    shop = shopRepository.save(shop);
    return DTOConverter.convert(shop);
}

```

Nesse serviço, será necessário mudar a chamada para o método que fará a validação do usuário, porque, agora, além de enviar o CPF do usuário, deve ser enviada também a chave de autenticação (lembrando que a passaremos na URL). No código, será necessário concatenar a chave na chamada da rota que busca o usuário com o CPF. A chamada antes era `"/user/" + cpf + "/cpf"` e agora será `"/user/" + cpf + "/cpf?key="+key`. Nenhuma outra mudança é necessária, o código a seguir mostra a implementação completa do método `getUserByCpf`.

```

public UserDTO getUserByCpf(String cpf, String key) {
    try {
        WebClient webClient = WebClient.builder()
            .baseUrl(userApiURL)
            .build();

        Mono<UserDTO> user = webClient.get()
            .uri("/user/" + cpf + "/cpf?key="+key)
            .retrieve()
            .bodyToMono(UserDTO.class);

        return user.block();
    } catch (Exception e) {
        throw new UserNotFoundException();
    }
}

```

A chamada para o serviço de compra continua igual à que fizemos no capítulo 6, basta agora adicionar no header da requisição o atributo `key` com a chave do usuário. Como definimos que a chave é obrigatória, caso ela não seja passada, o servidor responderá com o seguinte erro:

```
{
  "timestamp": "2020-05-16T16:16:49.833+0000",
  "status": 400,
  "error": "Bad Request",
  "message": "Missing request header 'key' for method parameter
of type String",
  "path": "/shopping/"
}
```

Agora os três microsserviços estão completos e os erros estão sendo tratados corretamente. No próximo capítulo, vamos ver como escrever testes de unidade para aumentar a confiança no nosso código e evitar que mudanças tenham efeitos indesejados na aplicação.

CAPÍTULO 11

Testes de unidade

Os teste automatizados são essenciais para se ter a segurança de que as aplicações estão funcionando corretamente e a de que outras funcionalidades da aplicação não pararão de funcionar quando uma mudança for feita em uma parte do código. Existem diversos tipos de testes, e todo projeto Spring deve ter no mínimo os testes de unidade, que são os testes que verificam se a implementação de um trecho de código, normalmente um método, está funcionando corretamente. Implementar esse tipo de teste em microsserviços com o Spring Boot é bastante simples. Existem diversas bibliotecas que podem ser usadas para isso, as mais conhecidas são o JUnit e o Mockito.

Neste capítulo, serão mostrados os testes de unidade desenvolvidos para os microsserviços *user-api* e *shopping-api*. Os da *product-api* estão no repositório do GitHub, mas não foram colocados no livro, pois eles são praticamente iguais aos testes dos outros dois microsserviços. Além disso, não serão colocados todos os testes desenvolvidos para esses microsserviços, pois foram adicionados muitos testes. Caso o leitor tenha interesse em verificar todos os testes desenvolvidos, consulte o código no repositório do GitHub.

11.1 Configuração

A configuração das bibliotecas de teste em um projeto Spring Boot é fácil, basta adicionar a dependência `spring-boot-starter-test`. Essa dependência já adiciona todas as principais bibliotecas que serão necessárias para os testes dos microsserviços, que são o JUnit — que é a biblioteca mais utilizada para o desenvolvimento de testes de unidade em Java — e o Mockito — que é a biblioteca usada para a utilização de *mocks*. Se você não conhece o conceito de *mocks*, leia o quadro a seguir.

Mocks

O objetivo de um teste de unidade é testar apenas aquele pequeno trecho de código que está sendo testado e, por isso, evitamos chamar as dependências daquele código. Para isso, utilizamos um *mock*, que é um objeto que simula o objeto real.

Por exemplo, quando implementamos um teste para o método *x* da classe *X*, e nele há uma chamada para o método *y* da classe *Y*, se utilizarmos um objeto *Y* real, vamos testar os métodos *x* e *y*, mas caso aconteça um erro, não saberemos exatamente onde está o erro. Para evitar isso, implementamos um *mock* do método *y* e indicamos exatamente o que ele deve fazer. Assim, garantimos que o teste verificou apenas se o método *x* está correto. O Mockito é a principal biblioteca para a utilização de *mocks* no Java.

O código a seguir mostra a dependência que deve ser adicionada em todos os microsserviços:

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

11.2 User-api

Vamos implementar testes para duas classes na *user-api*: a *UserService*, que possui os métodos que implementam as regras de negócio da aplicação, como os que criam, listam e buscam usuários — normalmente esse tipo de classe é a mais importante de testar, pois ela é a que tem mais probabilidade de ser alterada quando novas regras de negócio são adicionadas ao projeto; e vamos implementar os testes para a *UserController*, que é a classe que possui as rotas REST da aplicação. É importante também testar esse tipo de

classe, pois elas são as interfaces do serviço com o mundo exterior; por isso, é importante ter segurança nas mudanças delas.

Normalmente, para testar uma aplicação, criamos classes com o mesmo nome da classe normal, mas adicionando a palavra `Test` no final — por exemplo, a classe de teste da `UserService` será a `UserServiceTest`. Em um projeto Spring criado com o Maven, essas classes devem ficar no diretório `test/java`, no mesmo pacote da classe normal. Então, por exemplo, se a classe `UserService` está no diretório

`src/java/com/santana/java/back/end/service`, a `UserServiceTest` deverá estar no `test/java/com/santana/java/back/end/service`.

UserService

Vamos começar testando a `UserService`. Para isso, teremos que criar uma nova classe chamada `UserServiceTest` e, nela, vamos implementar diversos testes para as regras de negócio da API de usuários. Algumas informações importantes sobre essa classe:

- Toda classe de teste que utilizar o Mockito deve ter a anotação `@ExtendWith(MockitoExtension.class)` ;
- Normalmente a classe que está sendo testada — nesse caso, a `UserService` — deve ser anotada com `@InjectMocks` ;
- As dependências da classe que estamos testando — nesse caso, a `UserRepository` — deve ser anotada com `@Mock` ;
- Foi adicionado um método `getUser` para criar objetos da classe `Usuário`, pois isso será necessário em diversos testes. Esse método é público e estático, porque ele será utilizado por outras classes de teste.

```
package com.santana.java.back.end.service;
```

```
import com.santana.java.back.end.dto.UserDTO;  
import com.santana.java.back.end.model.User;  
import com.santana.java.back.end.repository.UserRepository;  
import org.junit.jupiter.api.Assertions;  
import org.junit.jupiter.api.Test;
```

```

import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.ArrayList;
import java.util.List;

@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @InjectMocks
    private UserService userService;

    @Mock
    private UserRepository userRepository;

    // os testes serão implementados aqui

    public static User getUser(Integer id, String nome, String cpf)
    {
        User user = new User();
        user.setId(id);
        user.setNome(nome);
        user.setCpf(cpf);
        user.setEndereco("endereco");
        user.setTelefone("5432");
        return user;
    }
}

```

Um exemplo de teste é verificar se o método que lista todos os usuários está funcionando corretamente; para isso, criamos dois usuários e fazemos um *mock* do método `findAll` da interface `userRepository`, para que, quando ele for chamado, ele retorne uma lista com os dois usuários criados. Por fim, é feita a chamada para o método `getAll`, que é o método que está sendo testado, e verificamos se foram retornados os dois usuários corretamente.

```

@Test
public void testListAllUsers() {

    List<User> users = new ArrayList<>();
    users.add(getUser(1, "User Name", "123"));
    users.add(getUser(2, "User Name 2", "321"));

    Mockito.when(userRepository.findAll()).thenReturn(users);

    List<UserDTO> usersReturn = userService.getAll();
    Assertions.assertEquals(2, usersReturn.size());

}

```

Note que no *mock* indicamos exatamente o que deve acontecer quando o método `findAll` da `UserRepository` é chamado; assim, estamos testando apenas o comportamento do método que queremos testar e não suas dependências.

Outro teste importante é verificar se o método que salva um novo usuário está funcionando corretamente. Nesse teste, devemos fazer um *mock* do método `save` da classe `userRepository`. Depois, podemos chamar o método que salva um usuário da classe `userService` e verificar se os dados do usuário foram retornados corretamente.

```

@Test
public void testSaveUser() {
    User userDB = getUser(1, "User Name", "123");
    UserDTO userDTO = DTOConverter.convert(userDB);

    Mockito.when(userRepository.save(Mockito.any()))
        .thenReturn(userDB);

    UserDTO user = userService.save(userDTO);
    Assertions.assertEquals("User Name", user.getNome());
    Assertions.assertEquals("123", user.getCpf());
}

```

O último teste dessa classe é o que verifica a edição do usuário. Ele é bastante parecido com o teste anterior, mas ele deve verificar se os dados de

endereço e telefone foram corretamente modificados.

```
@Test
public void testEditUser() {
    User userDB = getUser(1, "User Name", "123");

    Mockito.when(userRepository.findById(1L))
        .thenReturn(Optional.of(userDB));
    Mockito.when(userRepository.save(Mockito.any()))
        .thenReturn(userDB);

    UserDTO userDTO = DTOConverter.convert(userDB);
    userDTO.setEndereco("Novo endereco");
    userDTO.setTelefone("1234");

    UserDTO user = userService.editUser(1L, userDTO);
    Assertions.assertEquals("Novo endereco", user.getEndereco());
    Assertions.assertEquals("1234", user.getTelefone());
}
```

UserController

Além da camada de serviços, é importante também testar a camada de controle, para verificar se os serviços REST estão recebendo e retornando os dados corretamente. Para esse tipo de teste, também utilizaremos o Mockito e o JUnit, mas com algumas classes especiais desses frameworks para o teste de controladores REST. Vamos testar a classe UserController; para isso, temos que criar a classe UserControllerTest. No geral, essa classe é parecida com a que testa a camada de serviços; a grande diferença é o uso da classe MockMvc, que é uma classe que simula uma chamada REST para um serviço.

O código a seguir mostra a implementação inicial dessa classe, ainda sem os métodos de testes, que serão mostrados um a um a seguir.

```
package com.santana.java.back.end.controller;

import com.santana.java.back.end.service.UserService;
```

```

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

@ExtendWith(MockitoExtension.class)
public class UserControllerTest {

    @InjectMocks
    private UserController userController;

    @Mock
    private UserService userService;

    private MockMvc mockMvc;

    // os testes serão implementados aqui

    @BeforeEach
    public void setup() {
        mockMvc =
MockMvcBuilders.standaloneSetup(userController).build();
    }
}

```

O primeiro teste é o que verifica a rota que lista todos os usuários. Note que assim como nos testes da camada de serviço, é feito um `mock` da camada inferior — lá, foi o da camada `repository`; aqui, o da camada `service`. A grande diferença nesse teste é que o método não é chamado diretamente, mas, sim, a rota REST, usando o objeto `mockMvc`. Por fim, basta verificar se a resposta da rota é o formato esperado — no caso, comparando o JSON de resposta.

```

@Test
public void testListUsers() throws Exception {
    List<UserDTO> users = new ArrayList<>();
}

```

```

users.add(DTOConverter
    .convert(UserServiceTest.getUser(1, "Nome 1", "123")));

Mockito.when(userService.getAll()).thenReturn(users);

MvcResult result = mockMvc
    .perform(MockMvcRequestBuilders.get("/user"))
    .andExpect(MockMvcResultMatchers.status().isOk())
    .andReturn();

String resp = result.getResponse().getContentAsString();
Assertions.assertEquals("[{\"nome\":\"Nome 1\", \" +
    \"\"cpf\":\"123\", \"\"endereco\":\"endereco\", \"\"key\":null, \" +
    \"\"email\":null, \"\"telefone\":\"5432\", \"\"dataCadastro\":null}]]\"
    , resp);
}

```

11.3 Shopping-api

Os testes da *shopping-api* são bastante parecidos com a da *user-api*, mas vamos destacar aqui três testes importantes. O primeiro é o teste do método `save` da classe `ShopService`, que é o método mais importante da aplicação inteira, pois é ele que junta todas as peças da aplicação, verificando se um usuário e os produtos existem e, depois, salvando uma compra no banco de dados.

Os outros dois testes importantes são das classes `UserService` e `ProductService`, que fazem a comunicação com as outras APIs. Esses testes são diferentes dos outros, pois *mockar* a classe `WebDriver` é mais complicado; por isso, usaremos uma biblioteca interessante que serve para criar um servidor de teste que responde um objeto de teste para qualquer chamada.

Vamos começar com o teste mais simples, que é o do método `save`, pois ele é parecido com os testes desenvolvidos anteriormente. O primeiro passo

para essa implementação é construir a classe `ShopServiceTest` e inicializar todos os objetos que são necessários para o teste, assim como fizemos nos testes anteriores.

```
package com.santana.java.back.end.service;

import com.santana.java.back.end.dto.ItemDTO;
import com.santana.java.back.end.dto.ProductDTO;
import com.santana.java.back.end.dto.ShopDTO;
import com.santana.java.back.end.dto.UserDTO;
import com.santana.java.back.end.model.Shop;
import com.santana.java.back.end.repository.ShopRepository;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.ArrayList;

@ExtendWith(MockitoExtension.class)
public class ShopServiceTest {

    @InjectMocks
    private ShopService shopService;

    @Mock
    private UserService userService;

    @Mock
    private ProductService productService;

    @Mock
    private ShopRepository shopRepository;

}
```

Agora podemos implementar o teste do método que salva uma compra.

```

@Test
public void test_saveShop() {

    ItemDTO itemDTO = new ItemDTO();
    itemDTO.setProductIdentifier("123");
    itemDTO.setPrice(100F);

    ShopDTO shopDTO = new ShopDTO();
    shopDTO.setUserIdentifier("123");
    shopDTO.setItems(new ArrayList<>());
    shopDTO.getItems().add(itemDTO);
    shopDTO.setTotal(100F);

    ProductDTO productDTO = new ProductDTO();
    productDTO.setProductIdentifier("123");
    productDTO.setPreco(100F);

    Mockito.when(userService.getUserByCpf("123", "123"))
        .thenReturn(new UserDTO());
    Mockito.when(productService.getProductByIdentifier("123"))
        .thenReturn(productDTO);
    Mockito.when(shopRepository.save(Mockito.any()))
        .thenReturn(Shop.convert(shopDTO));

    shopDTO = shopService.save(shopDTO, "123");
    Assertions.assertEquals(100F, shopDTO.getTotal());
    Assertions.assertEquals(1, shopDTO.getItems().size());
    Mockito.verify(shopRepository,
Mockito.times(1)).save(Mockito.any());
}

```

Nesse método, eu fiz uma verificação diferente e que pode ser bastante útil em alguns casos, que é verificar se um método de um mock foi chamado uma determinada quantidade de vezes no método testado. Isso é feito usando o `Mockito.verify` — note que, na última linha do teste, verifica-se que o método `save` da `shopRepository` é chamado exatamente uma vez.

Vamos agora fazer os testes da classe que faz a comunicação com a *product-api*, que é a `ProductService`. Para esse teste, teremos que adicionar duas bibliotecas no arquivo `pom.xml`: a `okhttp` e a

mockwebserver , que são as bibliotecas que criam o servidor de testes que será utilizado para simular a chamada para o *product-api*. Note que ambas as bibliotecas estão no escopo de teste, pois elas só serão utilizadas para a execução dos testes.

```
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
  <version>4.0.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>mockwebserver</artifactId>
  <version>4.0.1</version>
  <scope>test</scope>
</dependency>
```

Agora, podemos implementar a classe `ProductServiceTest` , que terá os testes da classe `ProductService` . A grande novidade dessa classe é o objeto `mockBackEnd` , que simulará o serviço da *product-api*. Para configurar esse objeto, é necessário executar um método antes e depois de cada teste — o de antes inicializará o servidor de teste, e o que executa depois finalizará o servidor. A listagem a seguir mostra o código inicial dessa classe, ainda sem o teste do método.

```
package com.santana.java.back.end.service;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.santana.java.back.end.dto.ProductDTO;
import okhttp3.mockwebserver.MockResponse;
import okhttp3.mockwebserver.MockWebServer;
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.test.util.ReflectionTestUtils;

import java.io.IOException;
```

```

@ExtendWith(MockitoExtension.class)
public class ProductServiceTest {

    public static MockWebServer mockBackEnd;

    @InjectMocks
    private ProductService productService;

    @BeforeEach
    void setUp() throws IOException {
        mockBackEnd = new MockWebServer();
        mockBackEnd.start();

        String baseUrl = String.format("http://localhost:%s",
mockBackEnd.getPort());
        ReflectionTestUtils.setField(productService,
"productApiURL", baseUrl);
    }

    @AfterEach
    void tearDown() throws IOException {
        mockBackEnd.shutdown();
    }
}

```

Agora podemos implementar o teste do método `getProductByIdentifier`. A grande novidade desse teste é o *mock* da resposta do servidor que simula a *product-api*. Isso foi feito com o objeto `mockBackEnd`; note que, com ele, foi possível simular a resposta que será enviada quando o servidor é chamado, fazendo que ele retorne um JSON de um produto e adicionando um *header* na resposta indicando o *content-type*.

```

@Test
void test_getProductByIdentifier() throws Exception {
    ProductDTO productDTO = new ProductDTO();
    productDTO.setPreco(1000F);
    productDTO.setProductIdentifier("prod-identifier");

    ObjectMapper objectMapper = new ObjectMapper();
}

```

```

mockBackend.enqueue(new MockResponse()
    .setBody(objectMapper.writeValueAsString(productDTO))
    .addHeader("Content-Type", "application/json"));

productDTO = productService.getProductByIdentifier("prod-
identifier");

Assertions.assertEquals(1000F, productDTO.getPreco());
Assertions.assertEquals("prod-identifier",
productDTO.getProductIdentifier());
}

```

11.4 Cobertura de Código

Uma métrica interessante para verificar a quantidade do código que está sendo testado na aplicação é a cobertura de testes. Existem diversas ferramentas para verificar a cobertura. Uma das mais conhecidas é o JaCoCo (Java Code Coverage), que pode ser configurado para rodar sempre que o Maven é executado para construir a aplicação.

É bastante simples utilizar o JaCoCo em uma aplicação Maven. Basta adicionar o seguinte plugin no arquivo `pom.xml` no projeto.

```

<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.8</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>prepare-package</phase>
      <goals>

```

```

        <goal>report</goal>
    </goals>
</execution>
</executions>
</plugin>

```

Com esse plugin, toda vez que o projeto for construído, será gerado um conjunto de arquivos HTML que mostrarão a cobertura dos testes de todas as linhas do projeto. Esses arquivos serão salvos no caminho `target/site` dentro do projeto. Abrindo o arquivo `index.html`, será mostrado um resumo de como estão os testes na aplicação inteira. A figura a seguir mostra que a *user-api* está com uma cobertura de testes em 53% do código.

user-api













Element	Missed Instructions	Cov.
 com.santana.java.back.end.service		48%
 com.santana.java.back.end.controller		20%
 com.santana.java.back.end.model		75%
 com.santana.java.back.end.exception.advice		0%
 com.santana.java.back.end		0%
 com.santana.java.back.end.converter		91%
Total	184 of 399	53%

Figura 11.1: Tela com a cobertura de testes da user-api.

É também possível ver a cobertura de código linha a linha. A figura a seguir mostra que, na classe `UserService`, os métodos `getAll` e `save` estão totalmente cobertos por testes; já os métodos `getAllPage` e `findById` não possuem nenhum teste.

```

public List<UserDTO> getAll() {
    List<User> users = userRepository.findAll();
    return users
        .stream()
        .map(DTOConverter::convert)
        .collect(Collectors.toList());
}

public Page<UserDTO> getAllPage(Pageable page) {
    Page<User> users = userRepository.findAll(page);
    return users
        .map(DTOConverter::convert);
}

public UserDTO findById(long userId) {
    User user = userRepository.findById(userId).orElseThrow(() -> new UserNotFoundException());
    return DTOConverter.convert(user);
}

public UserDTO save(UserDTO userDTO) {
    userDTO.setKey(UUID.randomUUID().toString());
    userDTO.setDataCadastro(LocalDateTime.now());
    User user = userRepository.save(User.convert(userDTO));
    return DTOConverter.convert(user);
}

```

Figura 11.2: Cobertura de testes nas linhas da classe UserService.

É difícil saber qual é a cobertura de teste ideal para um projeto. O ideal é que os métodos mais críticos e os que tenham mudanças constantes tenham maior cobertura, para ter uma boa segurança de que essas mudanças não vão quebrar a aplicação.

Agora as aplicações estão prontas e com uma boa cobertura de testes, o que possibilita uma maior confiança em futuras mudanças no código. Porém, temos um problema: temos 3 APIs que devem ser acessadas em três portas diferentes. Se fizermos o deploy de nossas aplicações em máquinas diferentes fica ainda pior, pois a URL das aplicações também seriam diferentes. No próximo capítulo, veremos uma maneira de facilitar esse acesso usando um *api-gateway*.

CAPÍTULO 12

Api-gateway

Os três microsserviços estão prontos, mas um problema é que os três são executados em portas diferentes, e podem ser executados em servidores diferentes. Então um cliente da nossa aplicação precisará conhecer todas as URLs e todas as portas de todos os serviços. Com três microsserviços, não seria um grande problema descobrir todas essas informações, mas imagine isso em um ambiente com dezenas, ou até centenas de microsserviços. Além disso, dependendo do ambiente em que uma aplicação roda, as URLs e as portas dos serviços podem mudar. A figura a seguir mostra como é feito o acesso as aplicações com a versão atual dos microsserviços.

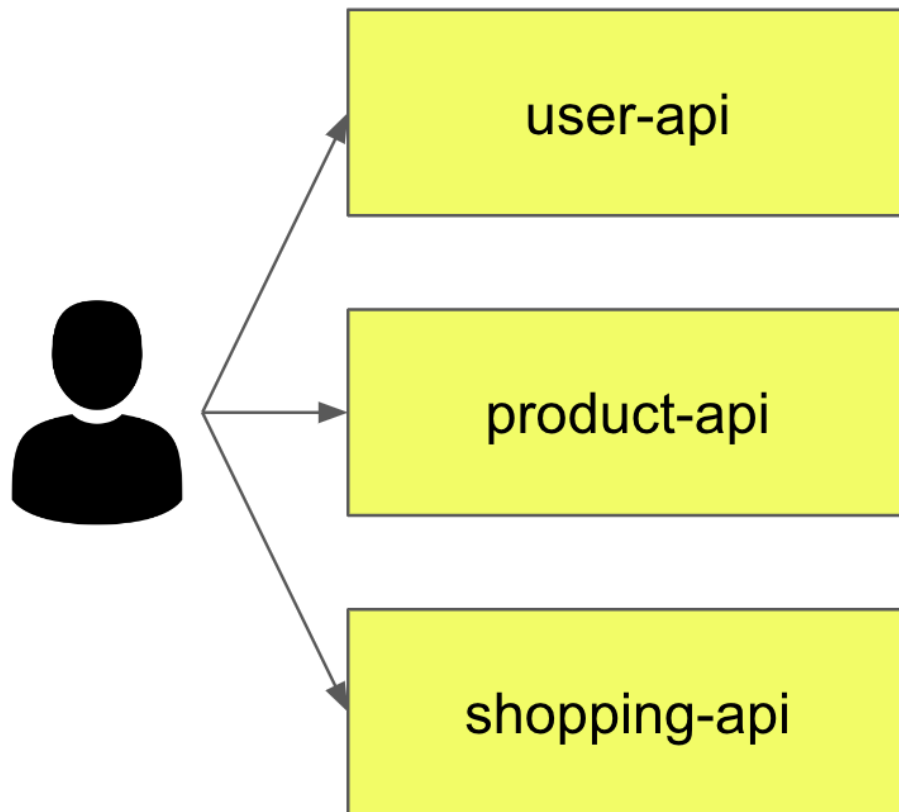


Figura 12.1: Aplicação sem um api-gateway.

É possível resolver esse problema utilizando um *api-gateway*, que é uma aplicação que fica na frente dos serviços, e todas as requisições são feitas para ela. Nela são configuradas algumas condições para que o gateway saiba redirecionar uma requisição que foi feita para ele para o serviço certo. A forma mais comum de fazer essa configuração é utilizar o prefixo das rotas; por exemplo, se a requisição feita para o gateway tiver o caminho iniciado com `/user`, é feito o redirecionamento para a *user-api*; caso ela seja feita para o caminho `/product`, é feito para a *product-api*. A figura a seguir mostra como ficará o acesso à aplicação após a implementação do *api-gateway*.

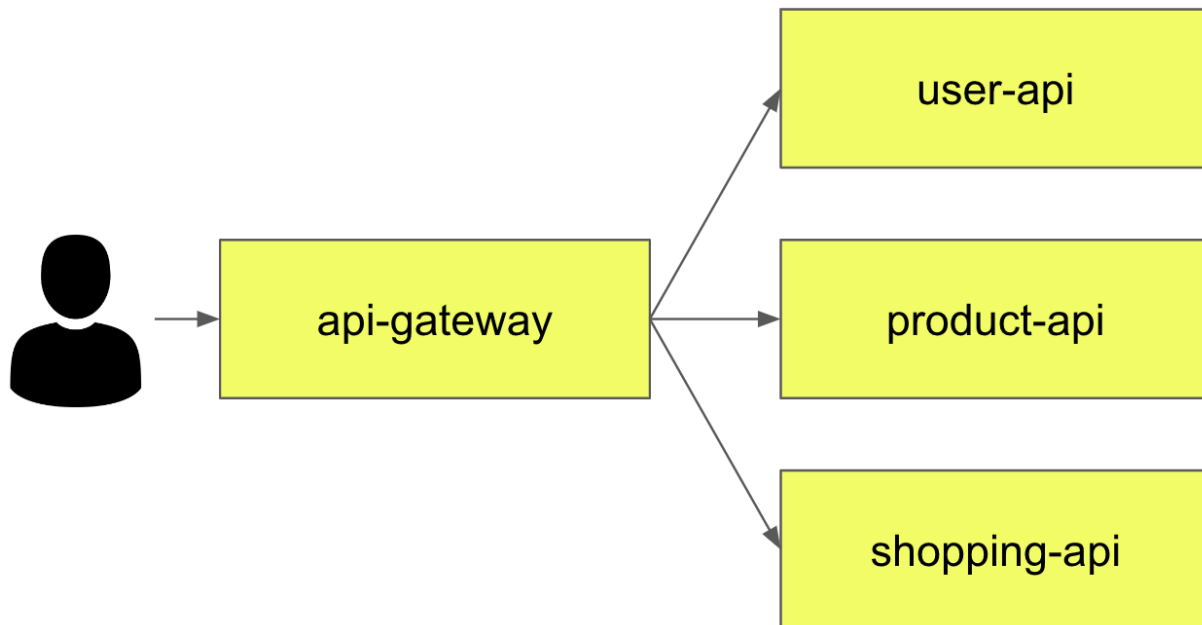


Figura 12.2: Aplicação com um api-gateway.

Para implementar o *api-gateway*, vamos utilizar o Spring Data, que é um projeto complementar ao Spring e que disponibiliza diversas ferramentas para o desenvolvimento e o gerenciamento de microsserviços.

O *api-gateway* vai ser um novo projeto, que executará na porta 8084, então todas as requisições para qualquer um dos microsserviços a partir de agora serão feitas para o endereço `http://localhost:8084`.

12.1 Configuração

Vamos utilizar o Maven para configurar o projeto do *api-gateway*. A primeira parte da configuração é parecida com a dos outros projetos. Primeiro, definimos a versão do Spring Boot que será utilizada e adicionamos a dependência `spring-cloud-starter-gateway`, que é a biblioteca do Spring para a implementação do *api-gateway*. A listagem a seguir mostra essa primeira parte da configuração da aplicação.

```
<?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>
  <groupId>com.santana.java.back.end</groupId>
  <artifactId>gateway</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>gateway</name>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.5</version>
    <relativePath/>
  </parent>

  <properties>
    <java.version>17</java.version>
    <spring-cloud.version>2021.0.5</spring-cloud.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>

```

Porém, como o *api-gateway* é parte do projeto Spring Cloud, também é necessário adicionar uma outra seção no arquivo `pom.xml` que indica qual a

versão das dependências desse projeto que será baixada. Isso é feito com a tag `dependencyManagement` do Maven.

Uma observação importante: no projeto do *api-gateway*, ainda estamos usando o Spring Boot versão 2.7.5, pois, na data da escrita do livro, ainda não existe uma versão oficial do Spring Cloud compatível com o Spring Boot 3.0.0. Porém, quando ela for lançada, nenhuma mudança no código será necessária. Apenas mudar as dependências será o suficiente para atualizar essa aplicação para a nova versão do Spring.

```
<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>
  </dependencies>
</dependencyManagement>
```

Além do `pom.xml`, também será necessário criar o arquivo `application.properties` para configurar a aplicação a ser executada na porta 8084.

```
server.port=8084
```

12.2 Implementação

Na implementação do projeto, teremos apenas uma classe, a `GatewayApplication`, que terá o método `main`, obrigatório em qualquer projeto Spring Boot, e o método `customRouteLocator`, que será responsável por fazer os redirecionamentos.

```
package com.santana.java.back.end.gateway;
```

```

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

@SpringBootApplication
public class GatewayApplication {

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

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder
builder) {
        return builder.routes()
            .route("user_route", r -> r.path("/user/**")
                .uri("http://localhost:8080"))
            .route("product_route", r -> r.path("/product/**")
                .uri("http://localhost:8081"))
            .route("shopping_route", r ->
r.path("/shopping/**")
                .uri("http://localhost:8082"))
            .build();

    }
}

```

A implementação desses redirecionamentos é bastante direta. Nela definimos o `path` que será recebido com uma expressão regular — por exemplo, `/user/**`, que indica qualquer path que inicie com `/user` — e depois definimos para qual serviço será feito o redirecionamento. Assim, se o *api-gateway* receber uma requisição para a rota `POST /user` no *api-gateway*, ela será redirecionada para a *user-api*; se a requisição for `GET /product/tv`, ela será redirecionada para a rota de busca de um produto na *product-api*.

A execução do *api-gateway* é exatamente igual à das outras aplicações: basta executar a classe que tem o método `main` e, para chamar uma rota pelo gateway, agora basta trocar a porta da requisição. Por exemplo: antes, a

chamada para a rota que listava os usuários da aplicação era `GET http://localhost:8080/user` ; agora, usando o *api-gateway*, é `GET http://localhost:8084/user` . Todas as rotas da aplicação devem continuar funcionando normalmente usando a porta 8084.

Agora que as aplicações estão todas prontas, devemos nos preocupar agora em como prepará-las para a implantação em um servidor e, também, deixá-las configuradas para a execução local, mas em um ambiente parecido com o de produção. Para isso, nos próximos capítulos vamos configurar o Docker e o Kubernetes e preparar a aplicação para ser executada com essas ferramentas.

CAPÍTULO 13

Executando a aplicação com Docker

Vamos executar nossas aplicações com o Docker agora. Será um passo importante antes de criar o cluster no Kubernetes, pois criaremos as imagens do Docker de todos os microsserviços. Para fazer isso, primeiro teremos que fazer algumas pequenas mudanças nas aplicações para deixar as configurações mais flexíveis. Depois, teremos que adicionar algumas dependências no projeto para gerar as imagens das nossas aplicações no Docker.

13.1 Adaptando as aplicações para o Docker

Na versão atual dos microsserviços, tanto as configurações do banco de dados no arquivo `application.properties` e o caminho para a *user-api* e a *product-api* na *shopping-api* estão *hardcoded*. Obviamente isso não é uma boa prática de programação, já que, dependendo do ambiente, essas configurações podem ser diferentes. Para resolver esse problema utilizaremos variáveis de ambiente. Veremos ainda neste capítulo que definir variáveis de ambiente no Docker (e, mais para a frente, no Kubernetes) e usá-las no Spring Boot é bastante simples.

A primeira mudança será no arquivo `application.properties` de todos os projetos. Utilizaremos as variáveis de ambiente para definir a URL, o nome de usuário e a senha do banco de dados. Como mostra a listagem a seguir, para utilizá-las, temos a seguinte sintaxe: `${ENV_VAR:'valor padrão'}`. Nela, definimos a variável de ambiente `ENV_VAR` e, se a variável existir, o valor utilizado será o dela; caso contrário, será utilizado o valor padrão definido depois dos dois pontos.

```
spring.datasource.url=${POSTGRES_URL:jdbc:postgresql://localhost:5432/dev}
```

```
spring.datasource.username=${POSTGRES_USER:postgres}
spring.datasource.password=${POSTGRES_PASSWORD:postgres}
```

Foram definidas três variáveis: a `POSTGRES_URL`, `POSTGRES_USER` e `POSTGRES_PASSWORD`. Veja que, para os valores padrões, se mantêm os *hardcoded* que estávamos utilizando antes, o que fará com que as aplicações continuem funcionando no ambiente local da mesma forma, sem que tenhamos que criar as variáveis de ambiente em nossas máquinas.

13.2 Shopping-api

A mesma ideia será aplicada para configurar o endereço da *user-api* e da *product-api* na *shopping-api*. Esses endereços estão configurados nas classes `ProductService` e `UserService`, nos métodos que fazem a chamada para as APIs. A mudança será bastante simples: utilizaremos a anotação `@Value` para carregar o valor de uma variável de ambiente na classe. A listagem a seguir mostra essa mudança na classe `ProductService`. Se você analisar o código desenvolvido no capítulo 8, verá que a ideia é a mesma, com a única diferença de que, em vez de utilizar o endereço da *product-api* diretamente no código, agora utilizamos uma variável de ambiente. Note que, depois da variável de ambiente, existe um valor padrão que será utilizado caso a variável de ambiente não seja encontrada.

```
@Service
public class ProductService {

    @Value("${PRODUCT_API_URL:http://localhost:8081}")
    private String productApiURL;

    public ProductDTO getProductByIdentifier(String
productIdentifier) {

        try {
            WebClient webClient = WebClient.builder()
                .baseUrl(productApiURL)
```

```

        .build();

        Mono<ProductDTO> product = webClient.get()
            .uri("/product/" + productIdentifier)
            .retrieve()
            .bodyToMono(ProductDTO.class);

        return product.block();
    } catch (Exception e) {
        e.printStackTrace();
        throw new ProductNotFoundException();
    }
}
}
}

```

Na classe `UserService`, faremos exatamente a mesma mudança, criando uma nova variável que recebe o valor da variável de ambiente

`USER_API_URL`.

```

@Service
public class UserService {

    @Value("${USER_API_URL:http://localhost:8080}")
    private String userApiURL;

    public UserDTO getUserByCpf(String cpf, String key) {
        try {
            WebClient webClient = WebClient.builder()
                .baseUrl(userApiURL)
                .build();

            Mono<UserDTO> user = webClient.get()
                .uri("/user/" + cpf + "/cpf?key="+key)
                .retrieve()
                .bodyToMono(UserDTO.class);

            return user.block();
        } catch (Exception e) {
            throw new UserNotFoundException();
        }
    }
}

```

```

    }
}
}

```

13.3 Api-gateway

No *api-gateway* teremos que fazer a mesma mudança para evitar que os endereços dos microsserviços fiquem *hardcoded* diretamente no código. Para isso, vamos criar três variáveis de ambiente com o endereço de cada uma das aplicações. Essas variáveis serão acessadas com a anotação `@Value` — note que aqui também deixamos os valores padrão, caso as variáveis de ambiente não sejam encontradas.

```

@Value("${USER_API_URL:http://localhost:8080}")
private String userApiURL;

@Value("${PRODUCT_API_URL:http://localhost:8081}")
private String productApiURL;

@Value("${SHOPPING_API_URL:http://localhost:8082}")
private String shoppingApiURL;

@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder)
{
    return builder.routes()
        .route("user_route", r -> r.path("/user/**")
            .uri(userApiURL))
        .route("product_route", r -> r.path("/product/**")
            .uri(productApiURL))
        .route("shopping_route", r -> r.path("/shopping/**")
            .uri(shoppingApiURL))
        .build();
}

```

Mas onde as variáveis de ambiente serão configuradas? Chegaremos lá. Até aqui, já acertamos as aplicações para utilizá-las; agora, quando formos

executar a aplicação com o Docker, indicaremos o valor para todas as variáveis de ambiente.

13.4 Configurando nossas aplicações para utilizar o Docker

Agora vamos criar as imagens do Docker com os microsserviços de nossa aplicação. Para isso, precisamos primeiro criar o `Dockerfile`, que é como uma receita de como a imagem de nosso projeto deve ser criada.

Cada microsserviço terá um `Dockerfile` específico, que deverá ser colocado na raiz do projeto. Além disso, precisamos adicionar no arquivo `pom.xml` de cada projeto um plugin do Docker para o Spring Boot.

Configurando o `settings.xml`

O Maven possui um arquivo para a definição de configurações gerais. Ele normalmente fica na pasta `.m2`, uma pasta na qual o Maven cria um repositório local com todas as dependências utilizadas nos projetos.

Normalmente, esse diretório fica na pasta do usuário — por exemplo, no Linux, seria na pasta `/home/usuario/.m2`; no MacOS,

`/Users/usuario/.m2` e, no Windows, `c:/Users/usuario/.m2`. Até agora não precisamos fazer nenhuma alteração nas configurações básicas do arquivo e, mesmo que ele não exista, o Maven utiliza as configurações padrões. Porém, para a utilização do *plugin* que gera a imagem do Docker, precisaremos mudar uma configuração.

O plugin que utilizaremos foi desenvolvido pela empresa *Spotify* e, para possibilitar sua utilização nos projetos, precisamos indicar que os plugins do *Spotify* estejam habilitados. Na listagem a seguir, é mostrado o XML completo do arquivo `settings.xml`. Caso você tenha alguma configuração específica em seu arquivo `settings.xml`, copie apenas o que está dentro da tag `<settings>`:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings
  xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <pluginGroups>
    <pluginGroup>com.spotify</pluginGroup>
  </pluginGroups>

</settings>
```

Configurando o pom.xml

No `pom.xml` de cada projeto, devem ser adicionadas duas novas configurações. A primeira é definir um prefixo para as imagens que serão criadas — o que não é obrigatório, mas é bom para facilitar agrupar as imagens de um mesmo projeto. O prefixo é definido com a tag `docker.image.prefix`. A segunda configuração é adicionar o plugin do Maven para gerar as imagens do Docker, o `dockerfile-maven-plugin`. Esse plugin é bastante utilizado em projetos Spring Boot.

O `groupId`, `artifactId` e `version` são valores padrões para o Maven utilizar o plugin corretamente. A única configuração com que devemos nos preocupar é o `repository`, que define o nome da nossa imagem no Docker. Nesse caso, concatenaremos o prefixo configurado inicialmente e o nome do projeto. Assim, geraremos as seguintes imagens Docker: `loja/product-api`, `loja/user-api` e `loja/shopping-api`.

```
<properties>
  <docker.image.prefix>loja</docker.image.prefix>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
```

```

        </plugin>
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>dockerfile-maven-plugin</artifactId>
            <version>1.4.9</version>
            <configuration>

<repository>${docker.image.prefix}/${project.artifactId}
</repository>
            </configuration>
        </plugin>
    </plugins>
</build>

```

Escrevendo o Dockerfile

O Dockerfile é praticamente o mesmo para todas as aplicações e a configuração é bastante simples. Cada linha de um Dockerfile é uma instrução de como gerar a imagem. No nosso arquivo, as seguintes instruções foram utilizadas:

- FROM : indica qual é a imagem base que usaremos — no caso, a `openjdk:17-alpine`, que é uma imagem simples com o Linux Alpine e com a Open JDK, versão 17, já instalada;
- VOLUME : cria a pasta `/tmp` no contêiner com os arquivos do projeto dentro dela;
- ARG : cria uma variável com o caminho para o JAR gerado do projeto;
- COPY : faz uma cópia do arquivo JAR com o nome `app.jar` ;
- ENTRYPOINT : define o comando que será executado dentro do contêiner — no caso, `java -jar /app.jar` .

```

FROM openjdk:17-alpine
VOLUME /tmp
ARG JAR_FILE=target/user-api-0.0.1.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]

```

A única mudança que deve ser feita no Dockerfile dos três microsserviços é o nome do arquivo. No exemplo anterior está *user-api*, ele deve ser trocado

para *shopping-api* e *product-api* nos outros microsserviços. Para construir a imagem do Docker com o nosso projeto, basta executar os seguintes comandos do Maven na raiz:

```
mvn clean install
mvn dockerfile:build
```

O primeiro comando gera o JAR e o segundo gera a imagem do Docker. Para verificar se a imagem foi gerada corretamente, execute o comando `docker images`. Se tudo estiver funcionando, deve ser exibida uma lista com todas as imagens Docker disponíveis na máquina, inclusive as nossas que acabamos de criar. Note também que a imagem do Postgres também deve estar na lista, já que quando criamos o contêiner do Postgres, o Docker fez uma cópia da imagem que está disponível no DockerHub para o registro local.

```
eduardo@eduardo:~/dev/analise_od/src$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
loja/user-api	latest	021a9dad8fd4	3 weeks ago	105MB
loja/product-api	latest	39416896a3c0	3 weeks ago	140MB
loja/shopping-api	latest	6aedfc92bace	3 weeks ago	122MB
loja/gateway	latest	fc78f19f51a1	13 seconds ago	135MB
postgres	latest	e2d75d1c1264	9 months ago	313MB

13.5 Rodando as aplicações com docker-compose

Podemos rodar nossa aplicação agora usando o comando `docker run`, mas, por termos três serviços mais o banco de dados, isso seria bastante trabalhoso. É mais fácil usar o **docker-compose**. Essa ferramenta permite a configuração de vários serviços em apenas um arquivo e, ao rodar apenas o comando `docker-compose up`, é possível inicializar todos os serviços de uma vez só. Da mesma forma, é possível parar todos os serviços com o comando `docker-compose down`. Vamos escrever esse arquivo parte a parte.

A primeira listagem mostra como configurar o banco de dados. No início do arquivo, sempre teremos a tag `version`, que indica qual versão da

especificação do docker-compose estamos utilizando — no caso, a 3.5. Depois vamos configurar uma lista de serviços, o primeiro sendo o `postgres`. A primeira informação é o nome e a versão da imagem (`postgres:latest`), depois, o mapeamento da porta do contêiner para a porta da máquina local (`5432:5432`). Por último, são definidas as três variáveis de ambientes que são necessárias para executar o Postgres: o usuário, a senha e o banco de dados padrão.

```
version: "3.5"

services:
  postgres:
    image: postgres:latest
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_DB: dev
      POSTGRES_PASSWORD: postgres
```

Agora, vamos configurar as nossas aplicações. Veremos que será bastante parecido configurar os três microsserviços. Iniciando com `user-api`, temos que configurar os mesmos valores, a imagem, a porta e as variáveis de ambiente. Vejam que essas variáveis de ambiente são exatamente as mesmas adicionadas quando fizemos as adaptações nas aplicações. Outra configuração importante é o `depends_on` para indicar que os nossos serviços dependem da imagem do Postgres para executar.

```
user:
  image: loja/user-api
  ports:
    - "8080:8080"
  environment:
    POSTGRES_URL: jdbc:postgresql://postgres:5432/dev
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  depends_on:
    - postgres
```

Note que na URL do Postgres, em vez de `localhost`, como estávamos definindo antes, agora estamos usando `postgres`; isso porque o `docker-compose` cria uma rede interna que permite que os contêineres se comuniquem diretamente pelos seus nomes. Então, para acessar o banco de dados, podemos usar o nome `postgres`, que é o identificador do contêiner do banco de dados.

A definição do `product-api` é bastante parecida com a da `user-api`, com diferenças apenas no nome da imagem e na porta utilizada.

```
product:
  image: loja/product-api
  ports:
    - "8081:8081"
  environment:
    POSTGRES_URL: jdbc:postgresql://postgres:5432/dev
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  depends_on:
    - postgres
```

Assim como os outros dois serviços, a definição da `shopping-api` também é bastante simples, com a diferença de que esse serviço tem duas variáveis de ambiente adicionais, que são as URLs para os outros dois serviços. Aqui, note também que os endereços dos serviços estão com a URL `product` e `user`, que são os nomes dos contêineres criados.

```
shopping:
  image: loja/shopping-api
  ports:
    - "8082:8082"
  environment:
    POSTGRES_URL: jdbc:postgresql://postgres:5432/dev
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    PRODUCT_API_URL: http://product:8081
    USER_API_URL: http://user:8080
  depends_on:
    - postgres
```

O último componente que deve ser configurado é o *api-gateway*, que tem as mesmas propriedades dos serviços anteriores. A única diferença é que ele não precisa das variáveis de ambiente do postgres, já que esse serviço não acessa o banco de dados.

```
gateway:
  image: loja/gateway
  ports:
    - "8084:8084"
  environment:
    PRODUCT_API_URL: http://product:8081
    USER_API_URL: http://user:8080
    SHOPPING_API_URL: http://shopping:8082
  depends_on:
    - postgres
```

Depois de definido esse script, basta rodar o comando `docker-compose up` no diretório onde o arquivo foi criado. Também é possível subir apenas um serviço específico; por exemplo, se quisermos subir apenas a `user-api`, podemos executar o comando `docker-compose up user`.

Agora que temos todas as aplicações configuradas e executando em um contêiner Docker, podemos iniciar a configuração do nosso *cluster*. Primeiro entenderemos alguns dos conceitos mais importantes do Kubernetes, depois veremos como instalar e configurar essa ferramenta e, por fim, faremos o deploy da aplicação no cluster.

CAPÍTULO 14

Kubernetes

Antes de começar a usar o Kubernetes, é importante entender bem os principais conceitos que utilizaremos para implantar nossas aplicações em um cluster local. O Kubernetes é uma plataforma de código aberto para gerenciar contêineres, facilitando a criação de um cluster com diversas máquinas virtuais executadas simultaneamente. Além disso, a ferramenta também facilita a automação da configuração e o gerenciamento dos serviços.

É possível testar aplicações baseada em microsserviços Spring Boot sem o Kubernetes. Porém, nesse caso, o ambiente de desenvolvimento fica muito mais simples que o de produção, o que pode acarretar erros inesperados e aumento da complexidade para se reproduzir um erro que acontece em produção. Por isso, é recomendado que o ambiente de quem programa seja o mais próximo possível do ambiente real.

Neste capítulo, veremos os principais conceitos que serão utilizados para a criação de nosso cluster local. Também já serão apresentados exemplos de como criar arquivos YAML, que é o formato principal utilizado para criar os objetos do Kubernetes. Neste livro, descreveremos os conceitos que são importantes para desenvolvedores conhecerem, como *Deployments*, *Pods* e *Services*. Obviamente, o Kubernetes disponibiliza muitas outras funcionalidades, como controle de segurança, por exemplo, porém focaremos no que é suficiente para a implantação de um cluster no ambiente de desenvolvimento que é utilizado basicamente para testes.

14.1 Deployments e Pods

O primeiro conceito importante do Kubernetes é o *Deployment*. Ele é uma especificação de como uma máquina virtual deve ser criada. Nele, definimos qual imagem do Docker será utilizada, quais os recursos de que a

máquina precisará para funcionar (CPU, memória, GPU), qual a porta em que ela deve executar, entre outras propriedades. A listagem a seguir mostra um YAML que cria um Deployment com a imagem Docker do Postgres. Isso será importante porque o nosso cluster precisará do Postgres executando para a nossa aplicação funcionar.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:latest
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_USER
          value: postgres
        - name: POSTGRES_DB
          value: dev
        - name: POSTGRES_PASSWORD
          value: postgres
```

As duas primeiras linhas são definições do Kubernetes, indicando que utilizaremos a versão `apps/v1`, e o tipo definido neste arquivo é um `Deployment`. O `metadata` define o nome do `Deployment` e um `label` que poderá ser usado para referenciá-lo.

O `spec` é a parte mais importante da definição. Ela define um contêiner ou uma lista de contêineres que executarão na máquina virtual. O campo `name` define o nome do contêiner; o `image`, qual a imagem do Docker que será executada; o `ports` define qual porta do contêiner estará aberta para requisições e o `env` define um conjunto de variáveis de ambiente. Por exemplo, na listagem anterior, temos o YAML que utilizaremos para fazer o deploy do Postgres no cluster (não se preocupe ainda com a forma como isso será feito).

A partir de um *Deployment*, o Kubernetes cria um ou mais `Pods`, que são como instâncias de um `Deployment`. O `Pod` é a máquina rodando no cluster e contém um ou mais contêineres sendo executados, um IP real dentro do cluster, espaço para armazenamento e utiliza recursos da máquina. Normalmente, quando criamos um `Deployment`, automaticamente o Kubernetes já cria também um respectivo `Pod`.

14.2 Services

Todos os `Pods` no Kubernetes recebem um IP durante a sua criação, e se um `Pod` de um `Deployment` for reiniciado, o IP pode mudar. Então, se um serviço dentro de um cluster do Kubernetes depende de outro, ele não pode confiar apenas no endereço IP, já que ele pode ser mudado a qualquer momento.

Para resolver esse problema, o Kubernetes possui o conceito de `Services`, que é uma forma de expor `Pods` de um determinado serviço em um endereço que não será alterado. Por exemplo, a listagem a seguir cria um `Service` para o `Deployment` do PostgreSQL que criamos anteriormente.

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    run: postgres
```

```
spec:
  ports:
    - port: 5432
      targetPort: 5432
      protocol: TCP
  selector:
    app: postgres
```

14.3 ConfigMaps

Os *ConfigMaps* são formas de armazenar configurações necessárias para as aplicações de maneira simples dentro do cluster. Em nossa aplicação, utilizaremos ConfigMaps para armazenar os dados para a conexão com o Postgres. Essas configurações podem ser facilmente acessadas pelas aplicações utilizando as variáveis de ambiente que são criadas nos Deployment . A listagem a seguir mostra um exemplo de um ConfigMap com os dados para a configuração de conexão com o Postgres.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: postgres-configmap
data:
  database_url: jdbc:postgresql://postgres:5432/dev
  database_user: postgres
  database_password: postgres
```

A criação de um ConfigMap é simples. As duas primeiras linhas definem o tipo do YAML e o metadata define o nome do ConfigMap que será utilizado depois para referenciá-lo. A parte mais importante é o data , que define os valores que poderão ser utilizados depois pelas aplicações. No exemplo, temos três valores (URL, usuário e senha) para a conexão com o Postgres. Os dados descritos nesse ConfigMap serão utilizados mais à frente, quando definirmos os arquivos de Deployment de nossos microsserviços.

14.4 Autenticação

O Kubernetes tem um sistema complexo de segurança. Precisaremos criar um usuário com acesso total ao cluster, o que será usado no próximo capítulo para acessar o *dashboard* de administração do cluster. Obviamente, essa não é uma boa prática para um cluster em produção, mas em um ambiente de desenvolvimento isso facilitará bastante o gerenciamento de nosso cluster. A listagem a seguir mostra como criar uma conta de usuário no Kubernetes. O usuário criado será o `loja-admin`.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: loja-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: loja-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: loja-admin
  namespace: kube-system
```

O YAML descrito possui duas partes, que são divididas pelos caracteres `--`. É possível fazer isso em qualquer arquivo YAML; por exemplo, poderíamos criar um `Deployment` e um `Service` em um mesmo arquivo, apenas separando as duas definições com os `---`. A primeira definição é de um `ServiceAccount`, que cria uma conta, ainda sem nenhuma permissão dentro do cluster, com o nome `loja-admin`. A segunda definição é um `ClusterRoleBinding`, que é onde vamos fazer associação com uma conta com um papel (*role*) dentro do cluster. O Kubernetes já possui um papel definido, que é o `ClusterRole`; o usuário que tem esse papel tem acesso

total ao cluster. Então, na parte `roleRef`, definimos que o associaremos à conta definida no item `subject`, no caso, `ClusterRole` e `loja-admin`.

O Kubernetes permite ainda a criação de papéis customizados, que possibilitam definirmos um subconjunto de ações e elementos do cluster para o usuário manipular. Por exemplo, na listagem a seguir, criamos um papel que permite que o usuário apenas veja informações sobre os Pods do cluster.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pods-list
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

Existem muitas opções para a autenticação no Kubernetes. Uma boa referência para verificar essas possibilidades é a documentação oficial da ferramenta, disponível no link

<https://kubernetes.io/docs/reference/access-authn-authz/rbac/>.

Dos conceitos principais que utilizaremos neste livro, ainda falta o `Ingress`, que é uma forma de acessar o cluster diretamente da máquina local, mas faremos essa configuração no final, depois que todas as aplicações já estiverem configuradas.

Com isso, temos os principais conceitos que precisaremos para a criação de nosso cluster. No próximo capítulo, vamos instalar e configurar o Kubernetes localmente. Utilizando os YAMLs apresentados neste capítulo, vamos já implantar e acessar o Postgres no cluster. Todos os arquivos YAML apresentados estão disponíveis no GitHub do projeto, na pasta *postgres-configuration*.

CAPÍTULO 15

Instalando o Kubernetes

Neste capítulo, instalaremos o Kubernetes e faremos a configuração básica do cluster, incluindo a instalação do Postgres, para depois apenas implantar os nossos microserviços. Todas as definições do Kubernetes podem ser feitas por arquivos nos formatos JSON ou YAML; neste livro, vamos usar YAML, que é o padrão mais utilizado no Kubernetes.

Windows e MAC

No Windows e no MAC, a instalação do Kubernetes é bem fácil: se você utilizou o *Docker for Desktop* que eu mencionei no capítulo 2, basta alguns cliques para ativar o Kubernetes. A imagem a seguir mostra a tela de configuração do Docker for Desktop. Para instalar o Kubernetes, basta selecionar a opção *Enable Kubernetes*. Esse processo deve demorar alguns minutos e, depois disso, o Kubernetes estará instalado e funcionando na máquina.

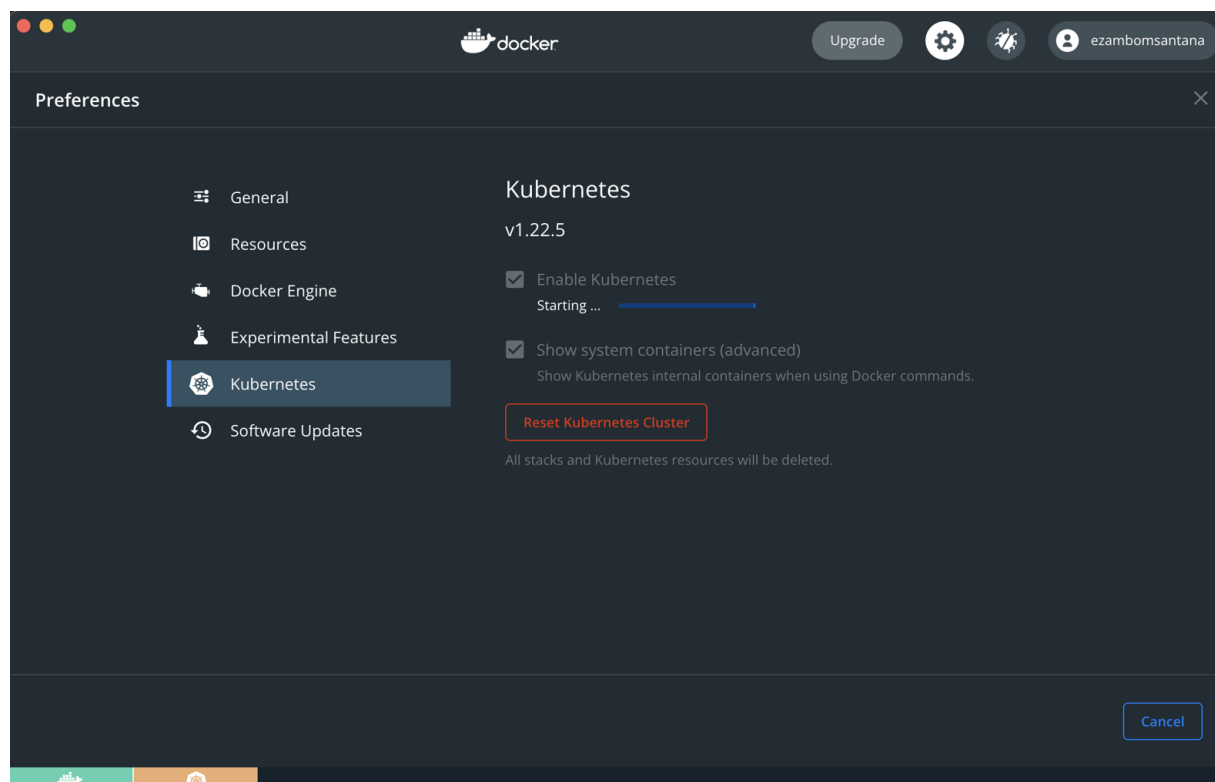


Figura 15.1: Instalando o Kubernetes.

Linux

No Linux, a instalação é um pouco mais complexa e existem algumas versões diferentes do Kubernetes que podem ser instaladas. No Linux, apenas para testes no ambiente de desenvolvimento, eu recomendo a instalação do *minikube* (<https://github.com/kubernetes/minikube>), que é uma versão simplificada do Kubernetes, mas que contém todas as funcionalidades importantes para o desenvolvedor. O primeiro passo para a instalação do minikube é instalar o *VirtualBox*, que é o servidor de virtualização que o minikube utilizará.


```
sudo apt install virtualbox virtualbox-ext-pack
```

Depois, podemos baixar o minikube com o comando `wget`, que baixa sua última versão estável. Temos que dar a permissão de execução ao arquivo e, por fim, copiá-lo para a pasta `/usr/local/bin/` para que possamos acessar esse programa pela linha de comando.

```
wget
https://storage.googleapis.com/minikube/releases/latest/minikube-
linux-amd64
chmod +x minikube-linux-amd64
sudo mv minikube-linux-amd64 /usr/local/bin/minikube
```

Depois desses três comandos, vamos verificar se o minikube foi instalado corretamente executando o comando `minikube version`. Provavelmente foi mostrada a versão do minikube e o hash do último *commit* feito nessa versão do programa. Finalmente, agora podemos iniciar o nosso cluster com o comando `minikube start`.

Se a instalação funcionar, a execução do comando deve exibir um *log* como o da figura abaixo.

A terminal window with a dark background and light-colored text. The prompt is 'eduardo@eduardo:~\$'. The command 'minikube start' has been entered. The output shows 'minikube v1.5.2 on Ubuntu 18.04', a tip to use 'minikube start -p <name>' to create a new cluster or 'minikube delete' to delete the current one, and progress messages: 'Using the running virtualbox "minikube" VM ...', 'Waiting for the host to be provisioned ...', 'Preparing Kubernetes v1.16.2 on Docker "18.09.9" ...', 'Relaunching Kubernetes using kubeadm ...', 'Waiting for: apiserver', and finally 'Done! kubectl is now configured to use "minikube"'.

```
eduardo@eduardo:~$ minikube start
minikube v1.5.2 on Ubuntu 18.04
Tip: Use 'minikube start -p <name>' to create a new cluster, or 'minikube delete' to delete this one.
Using the running virtualbox "minikube" VM ...
Waiting for the host to be provisioned ...
Preparing Kubernetes v1.16.2 on Docker "18.09.9" ...
Relaunching Kubernetes using kubeadm ...
Waiting for: apiserver
Done! kubectl is now configured to use "minikube"
```

Figura 15.2: Inicialização do minikube.

Por padrão, o minikube é iniciado usando apenas uma CPU e com 1gb de memória. Se você possui uma máquina com mais recursos, eu recomendo aumentar a quantidade de recursos utilizados. Na minha máquina, eu utilizo 2 CPUs e 4gb de memória. Para fazer isso, use o comando `minikube start` com os parâmetros `--cpus` e `--memory`, como no comando a seguir:

```
minikube start --cpus=2 --memory=4000mb
```

15.1 Instalando o kubectl

Existem duas formas de interagir com o Kubernetes agora: ou pela linha de comando, usando a ferramenta `kubectl`, ou via um dashboard.

Inicialmente, vamos instalar o `kubectl`. Isso é bastante simples nos três sistemas. No Ubuntu, o pacote de instalação do `kubectl` não está por padrão no `apt`, por isso são necessários os quatro comandos listados abaixo.

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |  
sudo apt-key add -  
sudo touch /etc/apt/sources.list.d/kubernetes.list  
echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" | sudo  
tee -a /etc/apt/sources.list.d/kubernetes.list  
sudo apt update  
sudo apt install kubectl
```

Basicamente, as quatro primeiras linhas configuram no `apt` local o repositório do Kubernetes e a última linha instala o `kubectl`. Para verificar se o `kubectl` foi instalado corretamente, rode o comando listado abaixo. Ele lista todos os objetos que existem no cluster. Como o cluster ainda está vazio, ele não deve exibir nada neste momento.

```
kubectl get all
```

No MAC, a instalação é mais simples: utilizando o HomeBrew, basta executar o comando a seguir que o `kubectl` será instalado na máquina.

```
brew install kubectl
```

No Windows, a instalação também é bem simples: basta baixar o executável `kubectl` e colocar o caminho para esse executável na variável de ambiente `PATH` do sistema operacional. Esse tutorial oficial do Kubernetes pode ajudar em caso de dúvidas nessa instalação:

<https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-linux>

Comandos importantes do `kubectl`

Agora que o Kubernetes e o `kubectl` estão instalados na máquina, podemos executar alguns comandos para verificar se o Kubernetes está

funcionando. Eles são iguais para todos os sistemas operacionais. Um primeiro comando para verificar a versão do Kubernetes que está instalada é o `kubectl version`, que, se tudo estiver corretamente configurado, tem como resposta algo parecido com:

```
Client Version: version.Info{Major:"1", Minor:"22",
GitVersion:"v1.22.5",
GitCommit:"5c99e2ac2ff9a3c549d9ca665e7bc05a3e18f07e",
GitTreeState:"clean", BuildDate:"2021-12-16T08:38:33Z",
GoVersion:"go1.16.12", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"22",
GitVersion:"v1.22.5",
GitCommit:"5c99e2ac2ff9a3c549d9ca665e7bc05a3e18f07e",
GitTreeState:"clean", BuildDate:"2021-12-16T08:32:32Z",
GoVersion:"go1.16.12", Compiler:"gc", Platform:"linux/amd64"}
```

Com o `kubectl` é possível verificar todos os elementos que estão instalados no cluster. Por exemplo, o comando `kubectl get pods` listará todos os *Pods* e o comando `kubectl get services` listará todos os *Services* que estão executando no cluster. Se você acabou de instalar o cluster, a lista ainda estará vazia, mas depois que instalarmos as nossas aplicações no cluster, o comando retornará:

NAME	READY	STATUS	RESTARTS	AGE
postgres-586c77c748-l4xkn	1/1	Running	2	1d
product-api-58bc98966c-7x6tj	1/1	Running	6	1d
shopping-api-57b775d45c-mtzwc	1/1	Running	4	1d
user-api-dc65df948-ggvnj	1/1	Running	3	1d

Outro comando que utilizaremos bastante é o que cria elementos no cluster. Ele tem o formato `kubectl create -f arquivo.yaml`, onde `arquivo.yaml` é o caminho para um arquivo que contém um YAML que criará um novo elemento no cluster. Esse processo é igual para criar qualquer coisa no cluster, como *Deployments*, *Pods* e *Services*. Esses são apenas os comandos básicos para começarmos a operar o cluster. Veremos mais alguns comandos nos próximos capítulos.

15.2 Instalando o Kubernetes Dashboard

O `kubectl` é o caminho mais direto para executar comandos no cluster. Porém, algumas atividades podem ser facilitadas utilizando o dashboard do Kubernetes. O dashboard não é incluído na instalação padrão do Kubernetes, então temos que instalá-lo separadamente.

O primeiro passo é rodar um YAML que está disponível na internet utilizando o `kubectl`. A opção `create` executa um YAML no cluster. Esse YAML contém a definição de diversas coisas, como um usuário padrão, um novo Deployment com o dashboard e um Service.

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.6.1/aio/deploy/recommended.yaml
```

Agora, para acessar o dashboard, precisamos criar um usuário no nosso cluster. Para isso, crie um arquivo YAML com a definição do `ServiceAccount` e `ClusterRoleBinding` mostrados no capítulo anterior. Basta executar o comando `kubectl create` como definido a seguir. O parâmetro `-f` indica o arquivo onde está o YAML.

```
kubectl create -f create-user.yaml
```

Para acessar o dashboard agora, é necessário executar o comando `kubectl proxy`, que cria um proxy para acessar a API do Kubernetes, incluindo o dashboard. Importante: esse comando abre um caminho apenas para acessar os serviços do próprio Kubernetes, mas não as aplicações. O dashboard agora está disponível para ser acessado na URL <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/login>.

Ao acessar essa página, será mostrada uma tela de login, como a da figura a seguir.

Kubernetes Dashboard

☐ Kubeconfig

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the [Configure Access to Multiple Clusters](#) section.

☒ Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the [Authentication](#) section.

Enter token *

Internal error (500): Not enough data to create auth info structure.

Sign in

Figura 15.3: Tela de login do dashboard.

O acesso ao dashboard pode ser feito de duas maneiras: ou com a criação de um arquivo de configuração, ou com o uso de um token para o usuário. O acesso com o token é mais simples, já que o acesso com o arquivo de configuração também precisa do token. Conseguir o token é bastante simples, basta executar o comando a seguir:

```
kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep loja-admin | awk '{print $1}')
```

O comando `-n kube-system describe secret` retorna uma lista com todos os tokens que existem no cluster. Como queremos apenas para o usuário `loja-admin`, fazemos o `grep` apenas para ele. Esse comando responderá algo assim:

```
Name:          loja-admin-token-qhnbw
Namespace:     kube-system
Labels:        <none>
Annotations:   kubernetes.io/service-account.name: loja-admin
                kubernetes.io/service-account.uid: 777caae2-e903-43df-b289-49b3410278eb
```

```
Type:  kubernetes.io/service-account-token
```

```
Data
```

```
====
```

```
token:          SEU_TOKEN_AQUI
```

```
ca.crt:      1066 bytes
namespace:   11 bytes
```

15.3 Subindo o Postgres no cluster

Agora, com o cluster e o `kubectl` instalados e utilizando os YAML apresentados no capítulo anterior para a definição do `Deployment` e do `Service` do Postgres, podemos já subir o banco de dados no minikube. Para isso, crie um arquivo com aqueles YAML, um chamado `postgres-deployment.yaml`, com a definição do `Deployment`, e outro chamado `postgres-service.yaml`, com a definição do `Service`. Para fazer a implantação agora, utilizamos o comando `create` do `kubectl`, como mostrado na listagem a seguir. A flag `-f` indica o nome do arquivo que será executado.

```
kubectl create -f postgres-deployment.yaml
kubectl create -f postgres-service.yaml
```

Se tudo funcionar corretamente, o Postgres deve estar rodando no cluster. Uma boa maneira de conferir isso é acessar o dashboard e verificar se está tudo lá. A imagem a seguir mostra um `Pod` do Postgres rodando no cluster.

Workloads

Deployments

Name	Namespace	Labels	Pods	Age ↑	Images	
postgres	default	app: postgres	1 / 1	8 hours	postgres:latest	
1 - 1 of 1 < < > >						

Pods

Name	Namespace	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Age ↑	
postgres-586c77c748-l4xkn	default	app: postgres pod-template-hash: 586c77c748	minikube	Running	1	-	-	8 hours	
1 - 1 of 1 < < > >									

Replica Sets

Name	Namespace	Labels	Pods	Age ↑	Images	
postgres-586c77c748	default	app: postgres pod-template-hash: 586c77c748	1 / 1	8 hours	postgres:latest	
1 - 1 of 1 < < > >						

Figura 15.4: Dashboard com instalação do Postgres.

Se o Postgres estiver rodando corretamente, agora é possível acessá-lo dentro do cluster. A forma mais simples de se fazer isso é utilizar o comando `port-forward` do `kubectl`, que mapeia uma porta do SO para uma porta do *Pod* que está em execução no cluster.

```
kubectl port-forward svc/postgres 5000:5432
```

No exemplo, a porta 5000 da máquina é mapeada para a porta 5432 do Service `svc/postgres`. O `svc` indica que o `port-forward` será feito para um Service do cluster. Sem isso, o `kubectl` tentará conectar em um Pod, o que também é possível, mas não recomendável, já que o nome do Pod pode ser alterado sempre que ele é reiniciado.

Agora, vamos criar também o `ConfigMap` com os dados de acesso ao Postgres. Esses dados serão utilizados pelas aplicações para que as informações de acesso ao banco de dados não fiquem *hardcoded*, isso é, definidas diretamente no código.

```
kubectl create -f config-map.yaml
```

O nosso cluster está pronto para receber as aplicações. Vamos agora, então, criar os YAMLS para todas as nossas aplicações — no código disponível no GitHub, esses arquivos estão na pasta `deploy` de todos os projetos.

CAPÍTULO 16

Implantando as aplicações no Kubernetes

Todos os projetos terão dois arquivos YAML, que serão o `deployment.yaml` e o `service.yaml`. O primeiro criará o *Deployment* de cada microsserviço e o segundo, o *Service* do Kubernetes para permitir o acesso ao serviço. Além disso, a *shopping-api* terá um *ConfigMap* que terá a URL da *user-api* e da *product-api*. Os arquivos *Deployment* e *Service* de todos os projetos são bem parecidos, mudando só as informações básicas de cada API, como o nome, a porta e o nome da imagem do Docker.

16.1 User-api e product-api

A definição do YAML para as APIs se parece com o YAML do Postgres apresentado no capítulo 12. As primeiras linhas basicamente dizem que o YAML definirá um deployment e qual o seu nome. Depois, a parte mais importante é a que define o contêiner, onde temos que definir o nome do contêiner, a imagem do Docker que será utilizada (a `loja/user-api:latest`), a porta (a 8080) e as variáveis de ambiente.

Note que foram utilizadas três variáveis, todas utilizando as informações do *ConfigMap* `postgres-configmap` que foi definido no capítulo 12. Uma propriedade nova nesse YAML é a `imagePullPolicy`, que indica para o Kubernetes procurar a imagem no registro local, e não no DockerHub. No YAML do Postgres isso não era necessário, pois a imagem do Postgres está no DockerHub.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-api
  labels:
    app: user-api
spec:
```

```

replicas: 1
selector:
  matchLabels:
    app: user-api
template:
  metadata:
    labels:
      app: user-api
spec:
  containers:
    - name: user-api
      image: loja/user-api:latest
      imagePullPolicy: Never
      ports:
        - containerPort: 8080
      env:
        - name: POSTGRES_URL
          valueFrom:
            configMapKeyRef:
              name: postgres-configmap
              key: database_url
        - name: POSTGRES_USER
          valueFrom:
            configMapKeyRef:
              name: postgres-configmap
              key: database_user
        - name: POSTGRES_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: postgres-configmap
              key: database_password

```

O arquivo `service` é exatamente igual ao do Postgres, apenas mudando os nomes e a porta que deverá ser acessada.

```

apiVersion: v1
kind: Service
metadata:
  name: user-api
  labels:
    run: user-api

```



```
spec:
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
  selector:
    app: user-api
```

Para a *product-api*, os arquivos possuem exatamente a mesma estrutura, porém devem ser trocados alguns valores — em todas as referências para a *user-api*, trocar para a *product-api*. A imagem deve ser trocada de `loja/user-api:latest` para `loja/product-api:latest`. Por último, devem ser trocadas as propriedades `port` e `containerPort` de `8080` para `8081`.

16.2 Shopping-api

Para a *shopping-api* também devem ser feitas as mesmas trocas nos arquivos do *Deployment* e *Service*: em todas as referências para a *user-api*, trocar para a *shopping-api*. A imagem deve ser trocada de `loja/user-api:latest` para `loja/shopping-api:latest`. Também devem ser trocadas as propriedades `port` e `containerPort` de `8080` para `8082`. Por último, deve-se adicionar no fim do arquivo YAML duas variáveis de ambiente, a `PRODUCT_API_URL` e a `USER_API_URL`. Essas variáveis terão a URL para acessar os outros dois microsserviços.

```
- name: PRODUCT_API_URL
  valueFrom:
    configMapKeyRef:
      name: shopping-api-configmap
      key: product_api_url
- name: USER_API_URL
  valueFrom:
    configMapKeyRef:
      name: shopping-api-configmap
      key: user_api_url
```

Além disso, a *shopping-api* também necessitará de um `ConfigMap` novo. Esse arquivo terá o endereço dos outros dois microsserviços. Note que esses valores serão injetados nas duas variáveis de ambiente que definimos no YAML anterior.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: shopping-api-configmap
data:
  user_api_url: http://user-api:8080/
  product_api_url: http://product-api:8081/
```

Agora, as aplicações estão totalmente configuradas para rodar no Kubernetes. Para isso, fizemos duas coisas importantes: primeiro, criamos a imagem do Docker e, depois, criamos os arquivos YAML para o Postgres e para os nossos microsserviços. Se você não estiver usando o minikube, basta rodar os comandos `kubectl create -f deploy/deployment.yaml` e `kubectl create -f deploy/service.yaml` em todos os projetos, criando-os no Kubernetes.

Se você estiver usando o minikube, será necessária mais uma pequena configuração: o minikube cria um registro do Docker próprio e não usa o da máquina hospedeira; por isso, é necessário configurar o terminal para utilizar o registro do minikube com o comando `eval $(minikube docker-env)`. Isso vale apenas para o terminal corrente, isto é, se for aberto outro terminal, ele voltará a acessar o registro da máquina hospedeira. Depois desse comando, é possível criar as imagens dos microsserviços exatamente como fizemos no capítulo 9. Se você utilizou o Docker for Desktop, ou a versão completa do Kubernetes, isso não é necessário, pois essas versões utilizam o registro do Docker da máquina hospedeira.

Se tudo estiver configurado corretamente, teremos o Dashboard com todos os *Pods* de nossos microsserviços rodando, como mostra a figura a seguir.

Pods								
Name	Namespace	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Age ↑
✓ product-api-58bc98966c-2fh7n	default	app: product-api pod-template-hash: 58bc98966c	minikube	Running	0	-	-	13 seconds
✓ user-api-dc65df948-xssvp	default	app: user-api pod-template-hash: dc65df948	minikube	Running	0	-	-	19 seconds
✓ shopping-api-57b775d45c-98zhr	default	app: shopping-api pod-template-hash: 57b775d45c	minikube	Running	0	-	-	22 seconds
✓ postgres-5ffbc67c7f-gpl98	default	app: postgres pod-template-hash: 5ffbc67c7f	minikube	Running	0	-	-	38 seconds
1 - 4 of 4 < < > >								

Figura 16.1: Dashboard com todos os microsserviços configurados.

O último passo agora é configurar o acesso externo ao cluster sem que seja necessário fazer um `port-forward`. Para isso, veremos, no próximo capítulo, mais um conceito importante do Kubernetes: o *Ingress*.

CAPÍTULO 17

Acesso externo ao cluster

Já conseguimos acessar o cluster usando o comando `kubectl port-forward`, porém, essa não é a melhor forma de fazer isso, pois toda vez que quisermos testar alguma coisa, teremos que executar esse comando. É possível configurar o acesso externo direto ao cluster, assim poderemos executar os serviços no cluster como se eles estivessem rodando normalmente em nossa máquina. Para isso, utilizaremos dois conceitos novos, o *Nginx* e o *Ingress*.

17.1 Nginx

O *Nginx* é um servidor web de código aberto que pode ser usado no Kubernetes. Com ele, é possível acessar os serviços no Kubernetes diretamente, sem ter que abrir uma porta da máquina local para o contêiner. Trata-se de um serviço independente que pode ser instalado no cluster, assim como fizemos com o Postgres e as nossas aplicações. Para instalar o *Nginx* no servidor, execute o seguinte comando:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-
nginx/controller-v1.4.0/deploy/static/provider/cloud/deploy.yaml
```

Esse comando acessa um arquivo YAML da última versão do *Nginx* e o instala em nosso cluster. A instalação fica disponível em um novo *Namespace* do Kubernetes, o `ingress-nginx`. Após executar esse comando, espere alguns segundos, pois o *Nginx* cria alguns *Pods* e *Services* e isso pode demorar um pouco. Quando finalizar, você deverá ver algo parecido com a imagem a seguir:

Namespace	Jobs				
ingress-nginx					
Overview					
Workloads					
Cron Jobs					
Daemon Sets					
Deployments					
Jobs					
Pods					
Replica Sets					
Replication Controllers					

Name	Labels	Pods	Age ↑	Images
✓ ingress-nginx-admission-create	app.kubernetes.io/component: admission-webhook app.kubernetes.io/instance: ingress-nginx Show all	0 / 1	2 minutes	jettech/kube-webhook-certgen:v1.2.0
✓ ingress-nginx-admission-patch	app.kubernetes.io/component: admission-webhook app.kubernetes.io/instance: ingress-nginx Show all	0 / 1	2 minutes	jettech/kube-webhook-certgen:v1.2.0

1 - 2 of 2 |< < > >|

Figura 17.1: Dashboard com o Nginx instalado.

Note o *Namespace* no canto superior esquerdo da figura. Estamos verificando o `ingress-nginx` e as nossas aplicações, que estão no *Namespace* default . Veja também que todos os *Jobs* do *Nginx* estão inicializados e executando corretamente.

17.2 Ingress

O último passo agora é criar um *Ingress*, que é um elemento do Kubernetes para permitir o acesso externo ao cluster sem a necessidade de fazer `port-forward` . Basicamente, o *Ingress* redireciona um acesso ao cluster para um *Service* de uma aplicação. A listagem a seguir mostra a criação do *Ingress* para a nossa aplicação.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: gateway-ingress
spec:
  rules:
    - host: localhost
      http:
        paths:
          - path: /user
            pathType: Prefix
            backend:
```

```

      service:
        name: user-api
        port:
          number: 8080
- path: /product
  pathType: Prefix
  backend:
    service:
      name: product-api
      port:
        number: 8081
- path: /shopping
  pathType: Prefix
  backend:
    service:
      name: shopping-api
      port:
        number: 8082

```

As informações importantes nesse YAML são o *name* do *Ingress*, que eu chamei de `gateway-ingress`, mas que pode ter qualquer valor, e os dados dentro das `rules`. O primeiro campo é o `host`, onde definiremos a URL de acesso ao nosso cluster, nesse caso, a `localhost` e os `paths`.

Note que existe um `path` para cada um dos nossos microsserviços mapeando o caminho para o *Service* do Kubernetes que criamos. Não é coincidência que os `paths`, como `/user/`, são iguais ao início dos nomes das rotas que definimos nos capítulos 4, 5 e 6. Isso é essencial, pois será com esses nomes que o Kubernetes conseguirá fazer o redirecionamento correto para as rotas.

Você pode verificar se o *Ingress* foi corretamente criado com o comando `kubectl get ingress`. Se tudo funcionou corretamente, você receberá uma resposta como a da listagem a seguir. Se você tentar acessar a rota pelo host `shopping.com`, isso ainda não funcionará. Precisaremos mapear o IP apresentado em *ADDRESS* para o host definido, mas o IP já está funcionando.

NAME	HOSTS	ADDRESS	PORTS	AGE
gateway-ingress	shopping.com	localhost	80	31s

Agora, basta acessar as aplicações normalmente, como fizemos nos capítulos 4, 5 e 6. Por exemplo, se quisermos criar um novo usuário na aplicação, podemos chamar a rota POST `http://localhost/user` e, depois, se quisermos listar os usuários, é só chamar a rota GET `http://localhost/user`.

17.3 Outros comandos do Kubectl

Agora que temos o cluster completo, podemos usar o `kubectl` para diversas coisas. No Dashboard, é possível fazer praticamente todas as ações, mas usar o `kubectl` normalmente é bem mais rápido. Já vimos alguns comandos importantes, como o `create`, para criar objetos no cluster, e o `get`, para recuperar uma lista de objetos.

Um outro comando interessante é o para verificar o log das aplicações. Para isso, basta executar o comando `kubectl logs -f nome-pod` que será acessado o log do `Pod` específico. Ele ficará sendo atualizado enquanto o console estiver executando.

Podemos também excluir elementos do cluster com o comando `kubectl delete nome-objeto`. Qualquer tipo de objeto pode ser excluído com esse comando, porém, é necessário sempre definir qual tipo de objeto está sendo excluído antes do nome do objeto — por exemplo, para deletar um `pod`, o comando é `kubectl delete pod/nome-objeto`.

Caso um microserviço esteja ficando lento, podemos aumentar o número de réplicas desse serviço, o que fará com que mais `Pods` de um mesmo `Deployment` sejam criados. Por exemplo, se quisermos executar três `Pods` da `user-api`, podemos executar o comando `kubectl scale --replicas=3 deployment/user-api`.

Outro fator importante é o `Namespace` em que os comandos estão sendo executados. Se nada for definido, tudo será executado no `Namespace default`. Para mudar isso, podemos passar um `-n nome-namespace`. Isso funciona para qualquer comando. Por exemplo, se quisermos listar os `pods`

do Nginx, podemos executar o comando `kubectl get pods -n ingress-nginx`.

NAME	READY	STATUS
RESTARTS AGE		
ingress-nginx-admission-create-zrqqc 0 25d	0/1	Completed
ingress-nginx-admission-patch-dkdrn 0 25d	0/1	Completed
ingress-nginx-controller-5cc4589cc8-cl24c 3 25d	1/1	Running

Conclusões

Com o acesso ao cluster via *Ingress* completo, a configuração do cluster está finalizada. Além das aplicações, configuramos o Postgres e o Nginx no cluster. Essa estrutura básica pode ser utilizada para o desenvolvimento de qualquer tipo de aplicação. No nosso exemplo, desenvolvemos uma miniloja virtual, porém essa arquitetura permite o desenvolvimento de aplicações para qualquer domínio.

Existem outros tópicos importantes que não foram tratados neste livro, mas que também valem a atenção, como o desenvolvimento de mecanismos de autenticação e autorização, a utilização de filas como o RabbitMQ e o Kafka, e a utilização de banco de dados NoSQL. O Kubernetes facilita a utilização de todos esses mecanismos — inclusive, nele podemos desenvolver microsserviços em outras linguagens e que podem se comunicar entre si.

Espero que você tenha gostado do livro e que tenha conseguido desenvolver a aplicação completa até aqui.