

Spring Boot

Acelere o desenvolvimento
de microsserviços



FERNANDO BOAGLIO

Spring Boot

Acelere o desenvolvimento
de microsserviços



FERNANDO BOAGLIO

Sumário

[ISBN](#)

[Agradecimentos](#)

[Quem é Fernando Boaglio?](#)

[Introdução](#)

[1. Tendências do mercado](#)

[2. Conhecendo o Spring Boot](#)

[3. Primeiro sistema](#)

[4. Explorando os dados](#)

[5. Explorando os templates](#)

[6. Desenvolvimento produtivo](#)

[7. Customizando nossa aplicação web](#)

[8. Expondo a API do seu serviço](#)

[9. Testando sua aplicação](#)

[10. Conversando com banco de dados NoSQL](#)

[11. Utilizando filas](#)

[12. Spring Reativo](#)

[13. Segurança](#)

[14. Empacotando e disponibilizando sua aplicação](#)

[15. Subindo na nuvem](#)

[16. Alta disponibilidade em sua aplicação](#)

[17. Ferramentas não oficiais](#)

[18. Indo além](#)

[19. Apêndice A — Starters](#)

[20. Apêndice B — Resumo das propriedades](#)

[21. Apêndice C — Guia de atualização](#)

ISBN

Impresso: 978-85-94120-00-7

EPUB: 978-85-94120-01-4

MOBI: 978-85-94120-02-1

-

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

-

Agradecimentos

Agradeço a você por decidir se aprofundar no Spring Boot, uma revolução no desenvolvimento convencional de aplicações Java.

Agradeço ao Josh Long, por gentilmente fornecer detalhes da história do Spring Boot.

Agradeço ao Henrique Lobo Weissmann que, além da excelente publicação do livro Vire o jogo com Spring Framework, ajudou na revisão do meu livro.

Agradeço também a todas as pessoas que se dedicam ao software livre, pois, sem elas, não teríamos excelentes sistemas operacionais, banco de dados, servidores de aplicação, browsers, ferramentas e tudo mais de ótima qualidade.

Agradeço à minha esposa por sempre estar ao meu lado, aos meus pais e a Deus por tudo.

E segue o jogo!

Quem é Fernando Boaglio?

Uma imagem fala mais do que mil palavras. Veja quem eu sou na figura:

Já foi estudante...



Linux user desde 1996



Já deu aulas...



Developer+commiter!



E começou a escrever...

BOAGLIO.COM



Figura -1.1: Fernando Boaglio

Introdução

Este livro foi feito para profissionais ou entusiastas Java que conhecem um pouco de Spring Framework e precisam aumentar sua produtividade, turbinando suas aplicações com Spring Boot. Você conhecerá os componentes principais dessa arquitetura revolucionária e tirará o máximo proveito dela por meio dos exemplos de acesso a banco de dados, filas, exibição de páginas web através de templates, serviços ReST (incluindo reativos com Spring Webflux) consumidos por front-end em jQuery e AngularJS, testes unitários e de integração, deploy na nuvem com a segurança do Spring Security e com a alta disponibilidade do Spring Cloud Gateway.

Com dois exemplos completos de sistemas, você poderá facilmente adaptá-los ao seu sistema e tirar proveito de suas vantagens o mais rápido possível. Por ser focado no Spring Boot, este livro não vai se aprofundar nos conceitos usados no Spring Framework, como JPA, Inversão de Controle ou Injeção de Dependências e também não contém todo o código necessário para construir as aplicações de exemplo, ele explica apenas as partes mais importantes. Se você já conhece o Spring Boot, além de melhorar os seus conhecimentos com este conteúdo, você pode aproveitar o Apêndice C — Guia de Atualização com todas as dicas e truques para atualizar o seu sistema.

Os códigos dos principais tópicos podem ser encontrados, por branches, no GitHub: <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo>

Capítulo 1

Tendências do mercado

Quem trabalhou com Java 5 ou com versões anteriores provavelmente lembra das aplicações grandes e pesadas que engoliam o hardware do servidor e, por uma pequena falha em uma parte da aplicação, comprometiam todo o sistema. Em uma aplicação nova, não podemos mais pensar dessa maneira, mas é interessante entender como chegamos a esse ponto e aprender com os erros do passado.

A evolução dos serviços

Era uma vez empresas que adotaram em massa um tal de Java. Ele conversava com qualquer banco de dados e até mainframe, fazia transação distribuída e tornava qualquer plataforma confiável para executar os seus sistemas. Depois do décimo sistema feito por diferentes equipes, foi descoberto que todos eles tinham um mesmo cadastro de clientes. O que começou a surgir nessas empresas foi o velho (e como é velho) problema do reaproveitamento de código.

Que tal isolar essa parte comum em um sistema único?

Assim, na virada do século, os serviços (internos e externos) começaram a usar um padrão de comunicação via XML, chamado Simple Object Access Protocol (SOAP) ou, em português, protocolo de acesso a objetos simples. Com ele, sistemas começaram a trocar informações, inclusive de diferentes linguagens e sistemas operacionais. Foi sem dúvida uma revolução.

Começava a era da arquitetura orientada a serviços, conhecida como SOA, que padronizava essa comunicação entre os diferentes serviços. O problema do padrão SOAP é sua complexidade em fazer qualquer coisa, como, por exemplo, realizar um serviço de consulta que retorne um valor simples. Isso tem muita abstração envolvida, com servidor de um lado e obrigatoriamente um cliente do serviço do outro, trafegando XML para ambos os lados. E tudo isso em cima do protocolo usado na internet (HTTP).

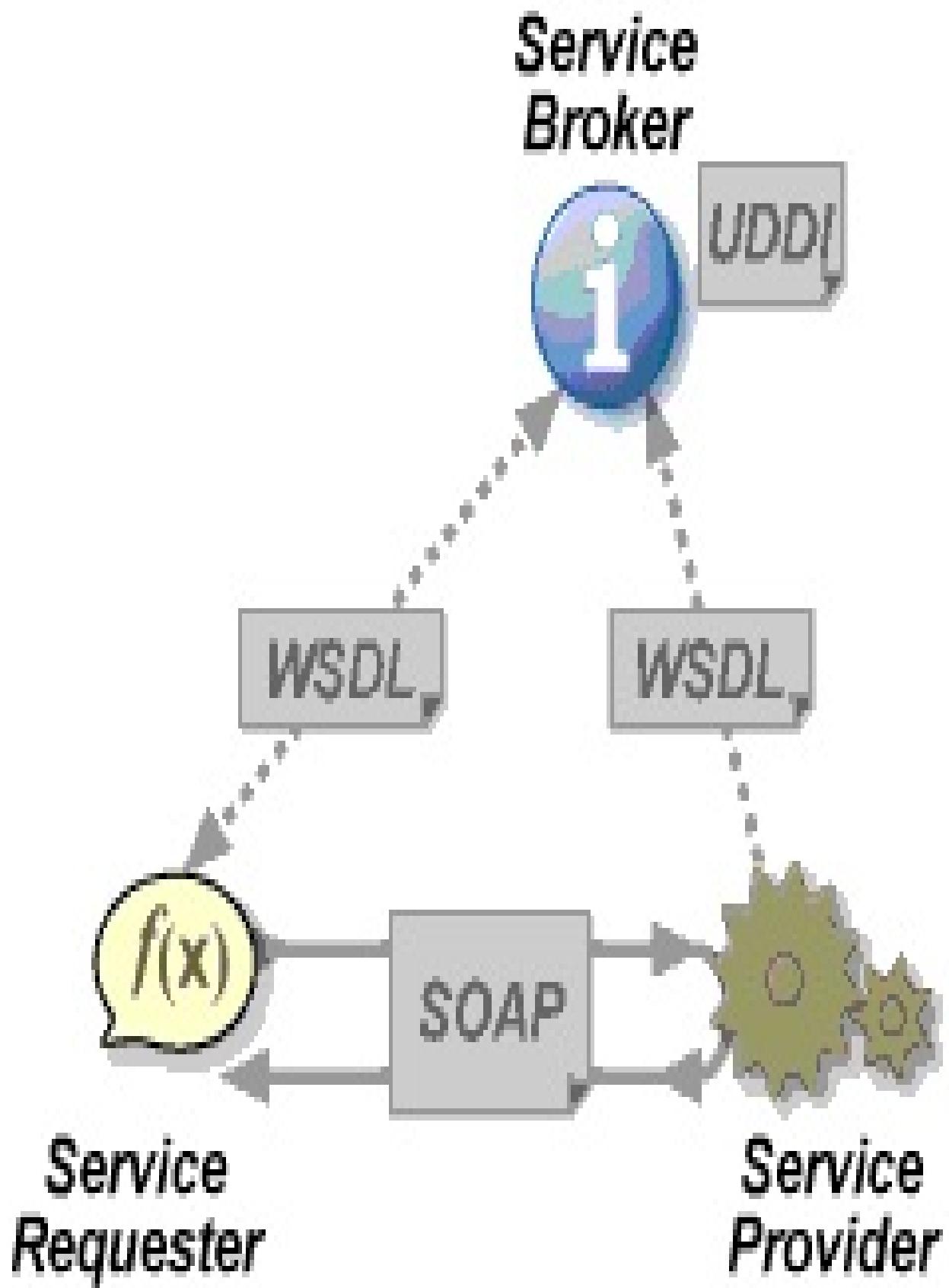


Figura 1.1: Serviços SOAP

No decorrer do tempo, muitos serviços SOAP, usados por vários sistemas, começaram a aparecer rapidamente como a principal causa de lentidão, obrigando os programadores a procurar alternativas, como trocar o serviço SOAP por um acesso direto ao banco de dados ou a um servidor Active Directory. Com esse cenário, Roy Thomas, um dos fundadores do projeto Apache HTTP, defendeu uma tese de doutorado apresentando uma alternativa bem simples: o famoso Representational State Transfer (Transferência de Estado Representacional), ou simplesmente REST (ou ReST). Essa simples alternativa ao SOAP aproveita os métodos existentes no protocolo HTTP para fazer as operações mais comuns existentes nos serviços, como busca e cadastro. Assim, por sua simplicidade e rapidez, esse padrão foi rapidamente adotado pelo mercado.

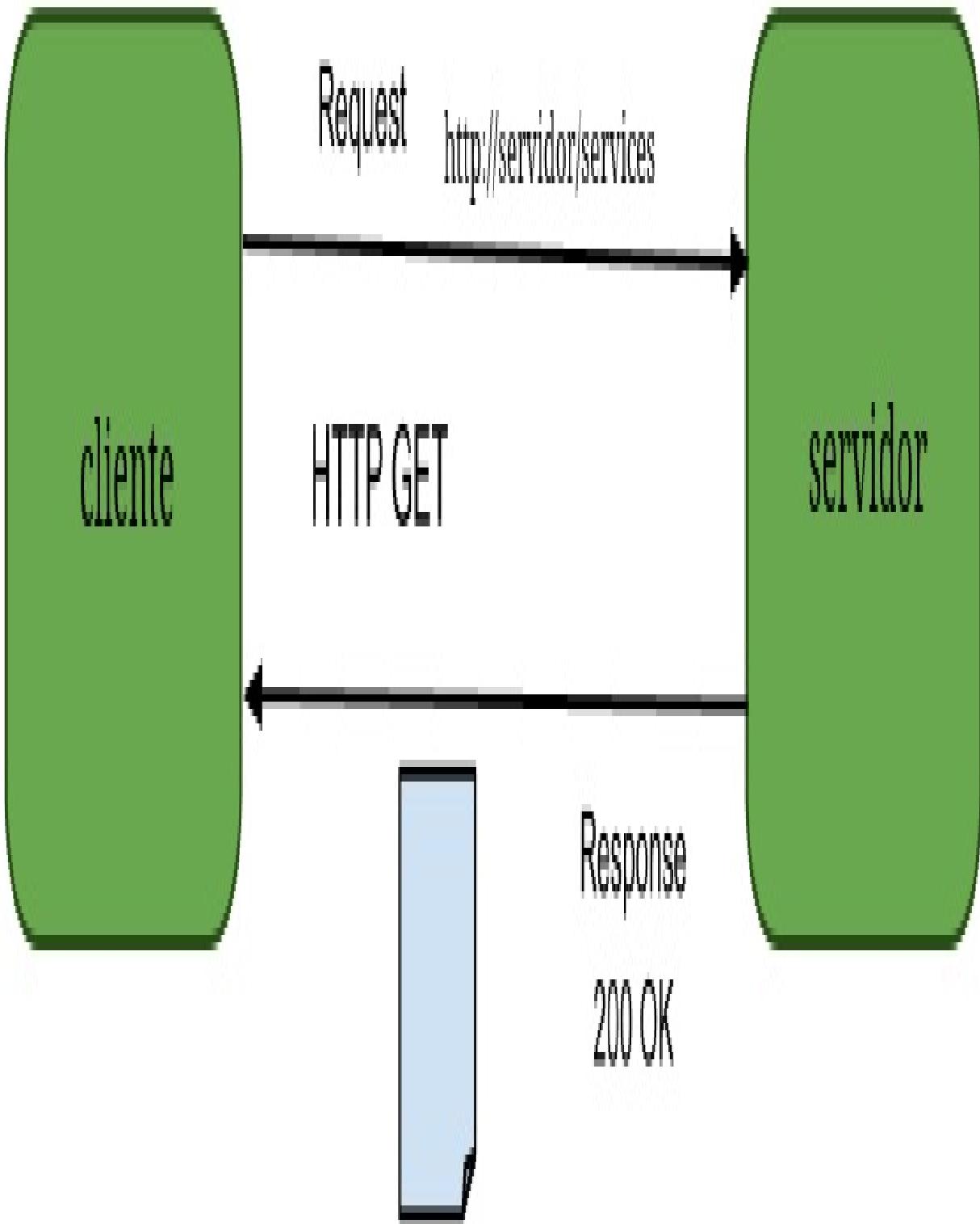


Figura 1.2: Serviços ReST

Os sistemas que usam ReST têm diferentes níveis de implementação não muito bem definidos pelo mercado. Algo mais próximo que existe de um padrão de classificação é o trabalho de Leonard Richardson, que definiu quatro níveis de maturidade de um sistema em ReST.

Glória do REST



Nível 3: controles hipermídia

Nível 2: verbos HTTP

Nível 1: recursos

Nível 0: a lama de XML



Figura 1.3: Glória do ReST

Mesmo com os serviços em ReST, as aplicações continuam a empacotar todas as funcionalidades em um só lugar, sendo classificadas como aplicações monolíticas. Fazendo uma analogia de funcionalidade com um brinquedo, a nossa aplicação sempre oferece a mesma quantidade de brinquedos, independentemente da demanda. Em um cenário com muitas crianças que só brincam com bonecas, sobrarão muitos aviões.

**Todas funcionalidades em
um único processo
(aplicação monolítica)**

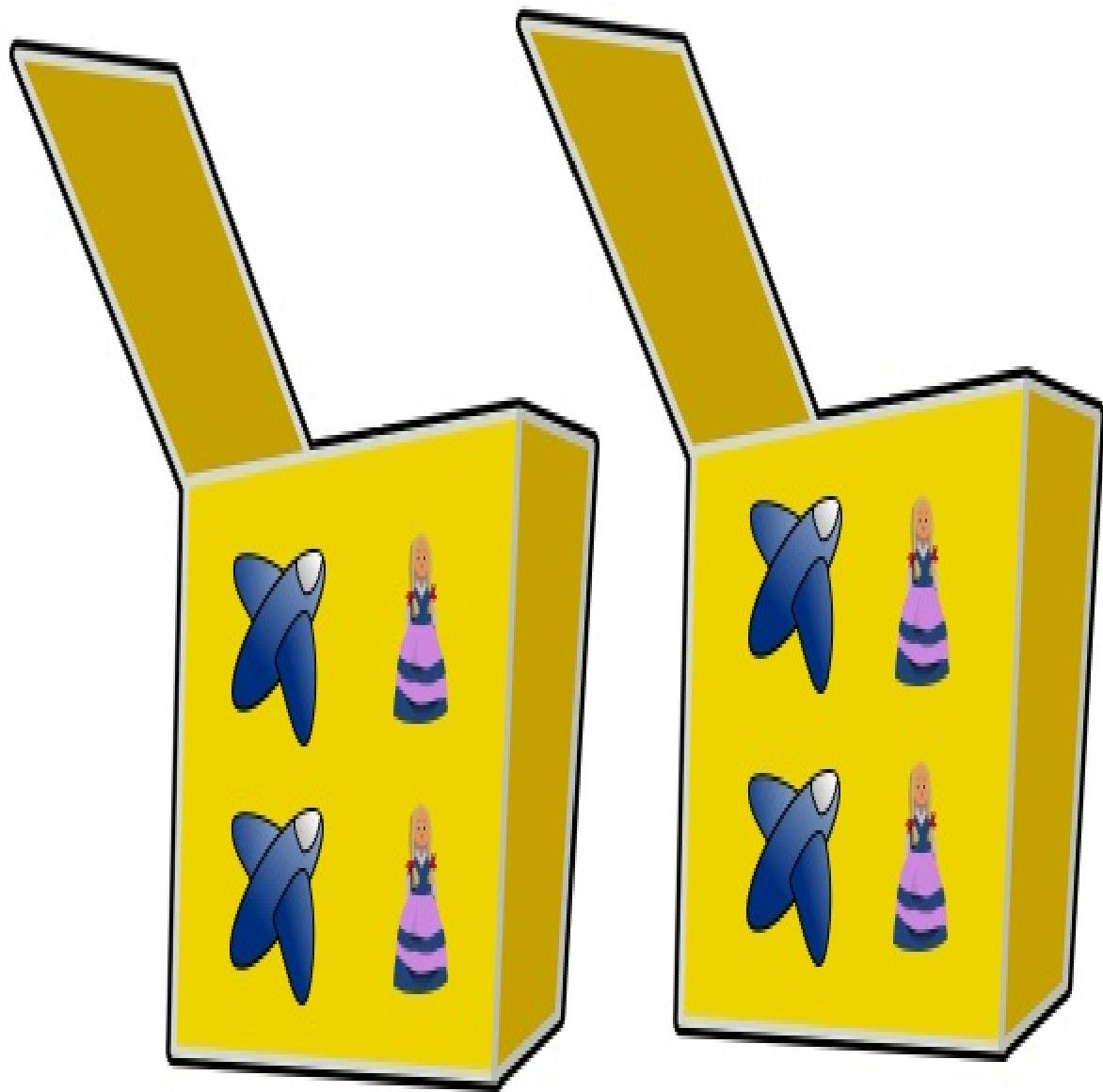
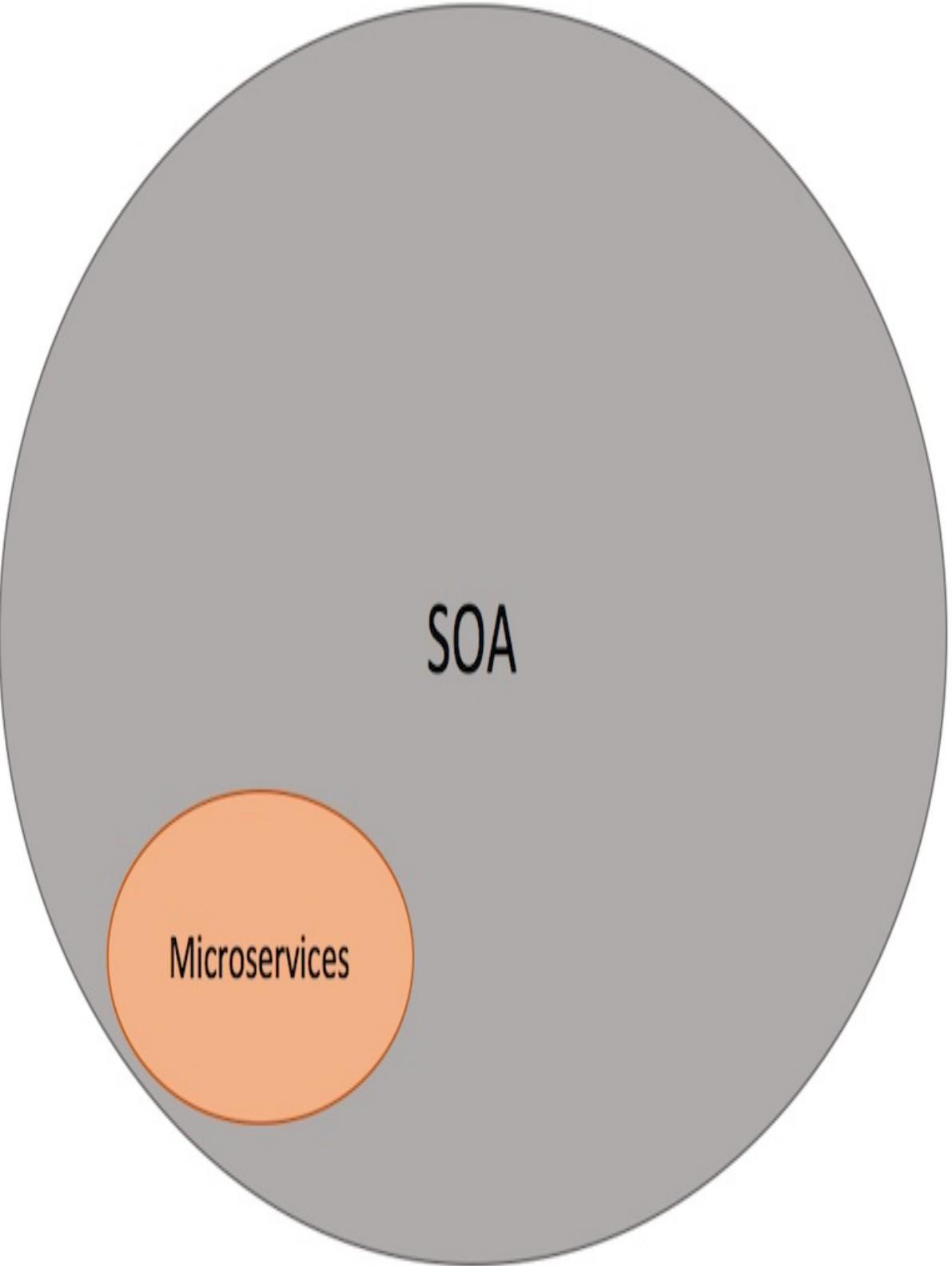
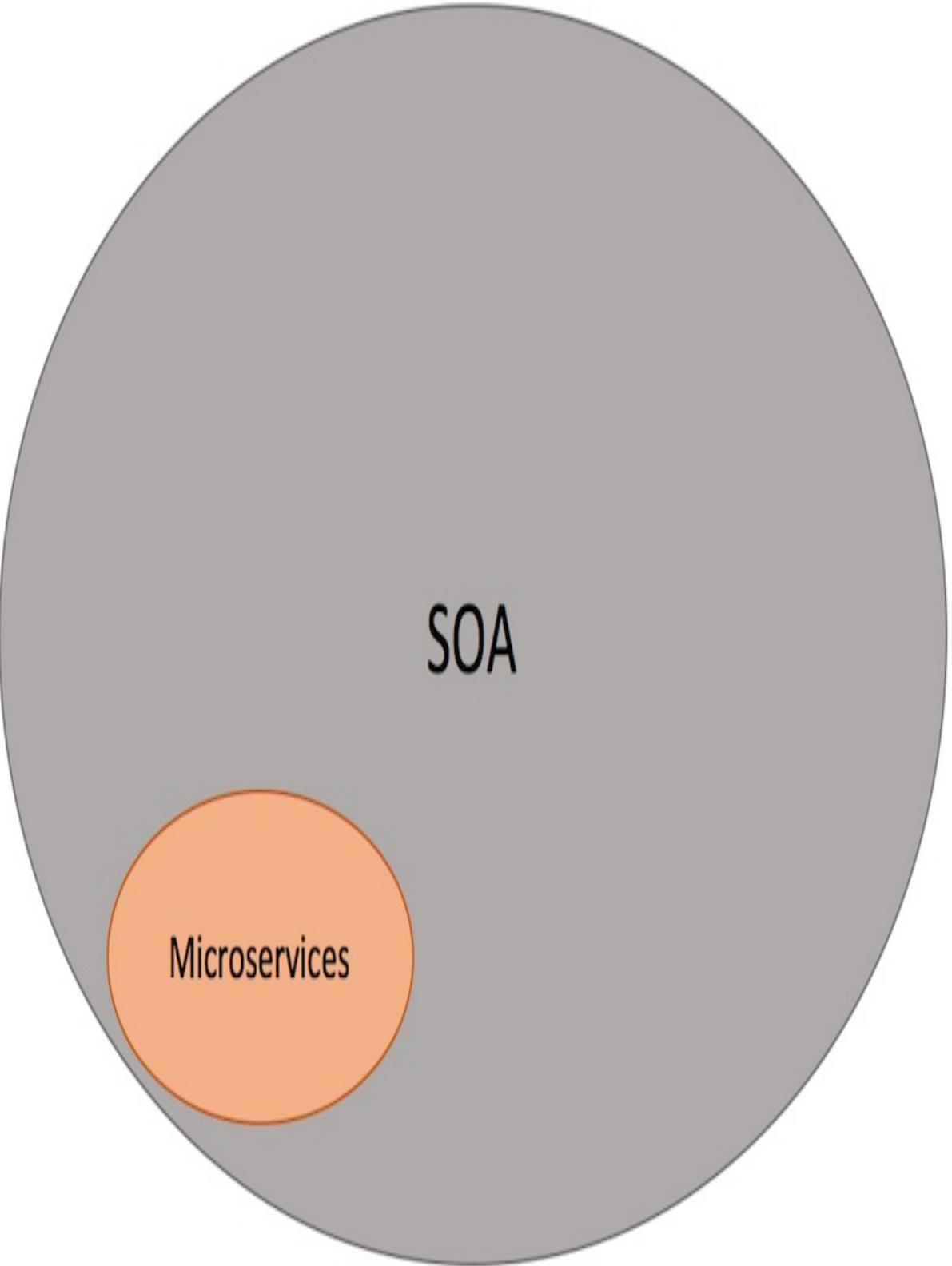


Figura 1.4: Aplicação monolítica

Para resolver esse problema, destaca-se um subconjunto da arquitetura SOA chamado microsserviços (microservices), que abraça a solução ReST com o objetivo de fornecer uma solução separada e independente para um problema.



SOA



Microservices

Figura 1.5: SOA

O ideal de uma aplicação é separar suas funcionalidades e se adequar ao cenário. Com o conceito de microsserviços, cada funcionalidade é independente e podemos crescer a sua quantidade conforme a demanda de nosso cenário.

Agora, conforme a demanda das crianças, podemos oferecer brinquedos sem que nada sobre.

Funcionalidades diferentes em serviços separados (microsserviços)

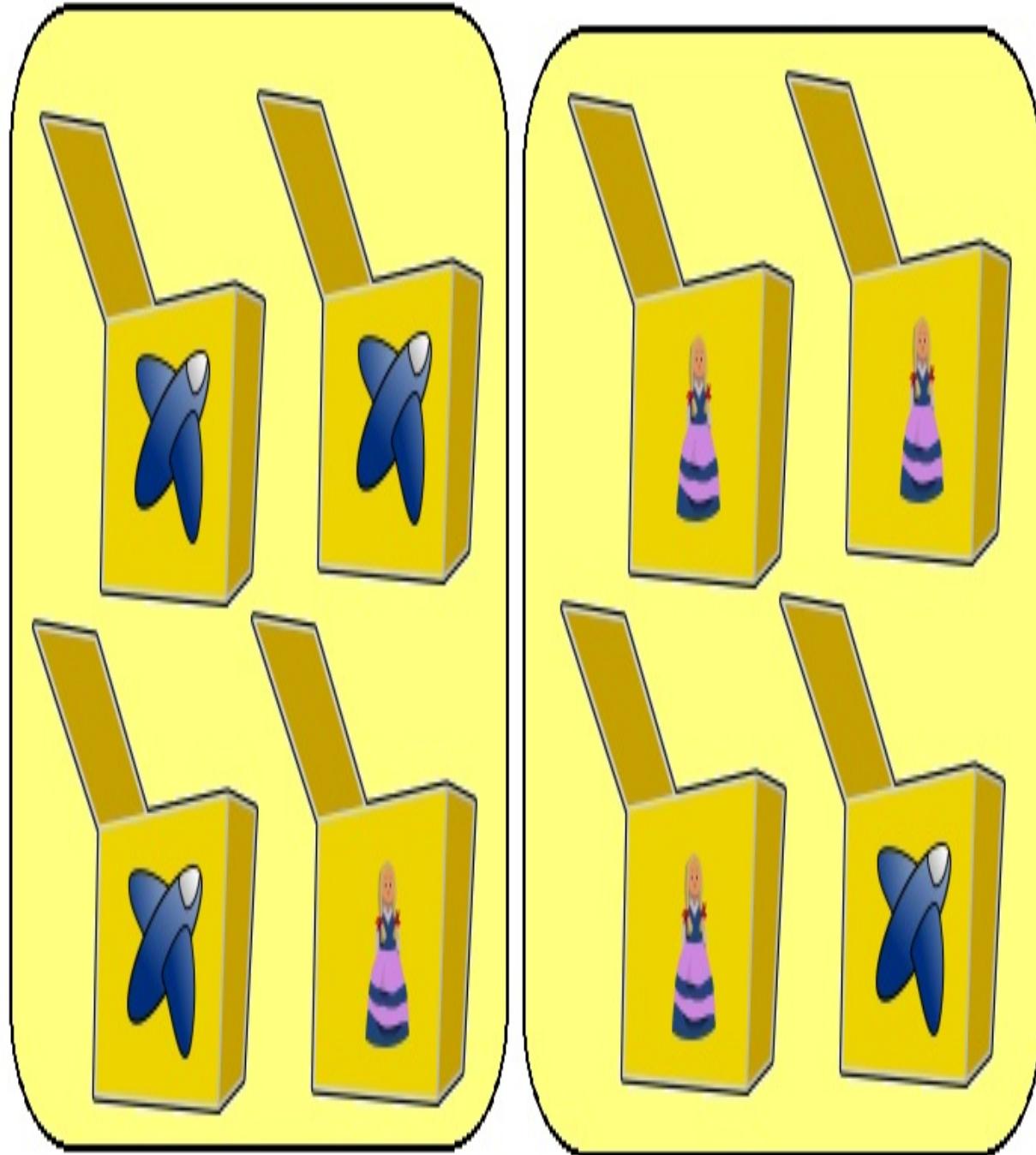


Figura 1.6: Microsserviços

Para nos ajudar nessa nova arquitetura, surgiu o Spring Boot.

1.1 Como surgiu o Spring Boot

Depois de 18 meses e milhares de commits, saiu a primeira versão do Spring Boot em abril de 2014. Josh Long, desenvolvedor Spring na Pivotal, afirma que a ideia inicial do Spring Boot veio da necessidade de o Spring Framework ter suporte a servidores web embutidos.

Depois, a equipe do Spring percebeu que existiam também outras pendências, como fazer aplicações prontas para nuvem (cloud-ready applications). Mais tarde, a anotação `@Conditional` foi criada no Spring Framework 4, o que foi a base para que o Spring Boot fosse criado.

Portanto, o Spring Boot é uma maneira eficiente e eficaz de criar uma aplicação em Spring e facilmente colocá-la no ar, funcionando sem depender de um servidor de aplicação. O Spring Boot criou um conceito novo, não existente até o momento na especificação JEE, que acelera o desenvolvimento e simplifica bastante a vida de quem trabalha com aplicações do Spring Framework, mas não ajuda em nada quem desenvolve com a especificação oficial (com EJB, CDI ou JSF).

No próximo capítulo, vamos conhecer mais detalhes sobre essa ferramenta revolucionária.

Capítulo 2

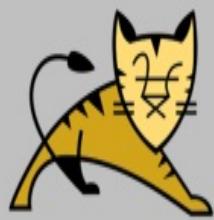
Conhecendo o Spring Boot

Desde 2003, o ecossistema Spring cresceu muito, o que do ponto de vista do desenvolvedor é bom, pois aumenta a gama de opções para usar, e ele mesmo não precisa implementar. Por exemplo: para adicionar uma autenticação na aplicação, podemos usar o Spring Security; para autenticar no Facebook ou Google, podemos usar o Spring Social; já se existir uma necessidade de criar muitos processos com horário agendado, temos o Spring Batch. E essa lista é enorme.

Entretanto, esse crescimento do Spring trouxe alguns problemas: com muitos módulos, vieram muitas dependências, e a configuração já não é tão simples como antes.

O Spring Boot, além de impulsionar o desenvolvimento para microserviços, também ajuda na configuração importando e configurando automaticamente todas as dependências, como veremos nos próximos capítulos. Algumas vezes, ele é confundido com um simples framework, mas na verdade ele é um conceito totalmente novo de criar aplicações web.

No conceito de Java Web Container, temos o framework Spring controlando as suas regras de negócio empacotadas em um JAR, que deverá obedecer aos padrões (servlet, filter, diretório WEB-INF etc.).



Apache
Tomcat

jetty://

undertow

The logo for Undertow, which features a stylized blue 'U' shape with a wavy line underneath it.

spring

The logo for Spring, which includes a green leaf icon followed by the word "spring" in lowercase.

regras
de negócio

Figura 2.1: Arquitetura Java Web tradicional

No conceito novo, temos o Spring Boot no controle total providenciando o servidor web e controlando as suas regras de negócio.



jetty://



regras
de negócio

Figura 2.2: Arquitetura Java Web com Spring Boot

2.1 Java 11

Java 11 (lançado em 2018) não é a versão mais recente, mas é a atual LTS (Long Term Support). Isso significa que o fabricante garante suporte com patches por um tempo maior do que as outras versões (nesse caso, por oito anos, ou seja, até 2026).

No novo modelo de lançamento de versões (releases), a cada três anos teremos uma LTS.

New JDK Release Model - LTS Every 3 years

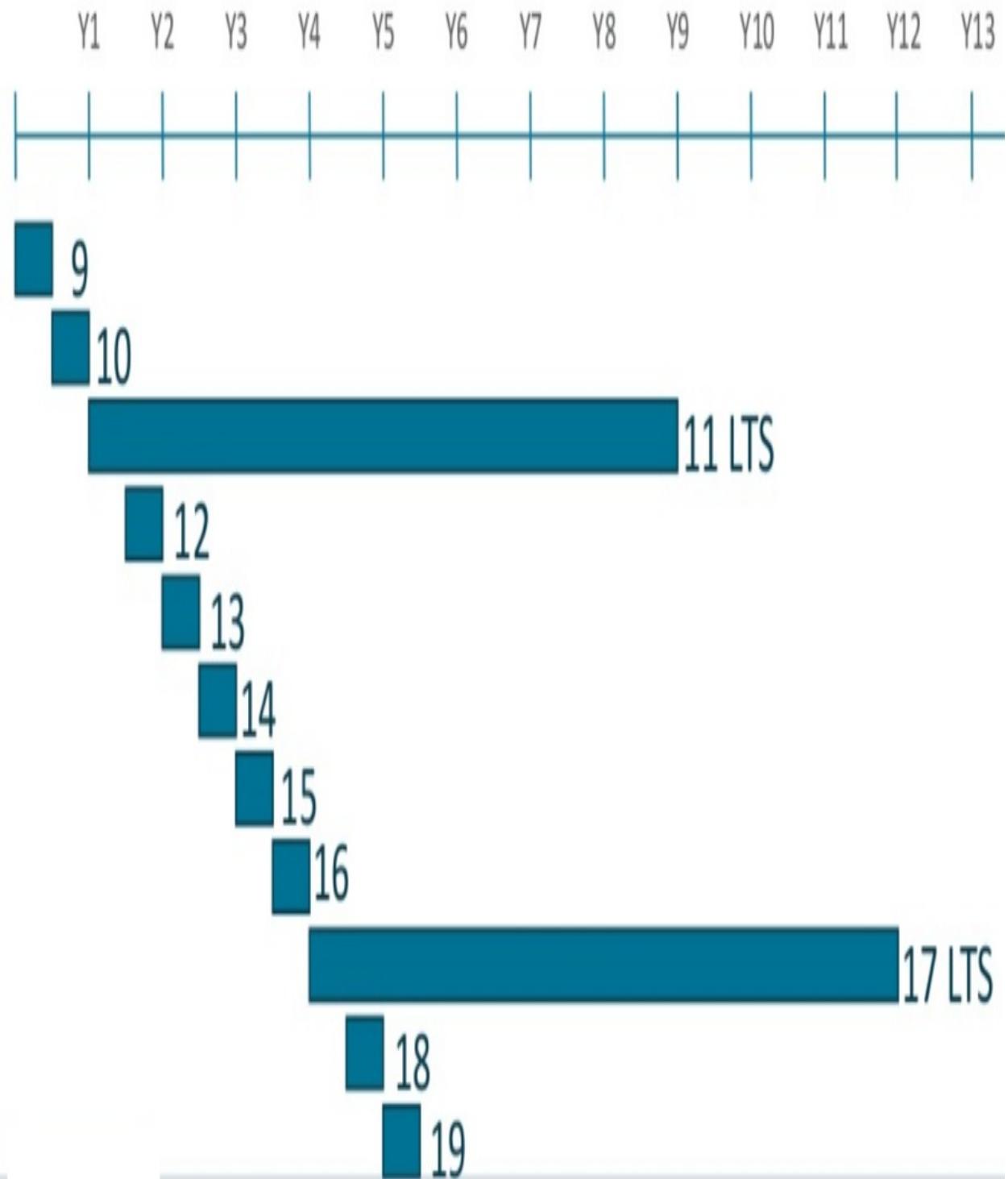


Figura 2.3: Versões do Java

O uso das versões LTS em produção é sugerido para oferecer mais segurança e estabilidade aos sistemas.

Desde 2018, o Spring Framework 5.1 suporta oficialmente o Java 11. O Java 11 é compatível com Spring Boot 2.1 ou superior, portanto ele foi adotado como versão oficial neste livro.

O Java 11 também oferece suporte ao TLS (Transport Layer Security) 1.3, que fornece melhorias significativas de segurança e desempenho.

2.2 Sua arquitetura

O logotipo do Spring Boot vem do ícone de iniciar a máquina (boot), cuja ideia é iniciar a aplicação:

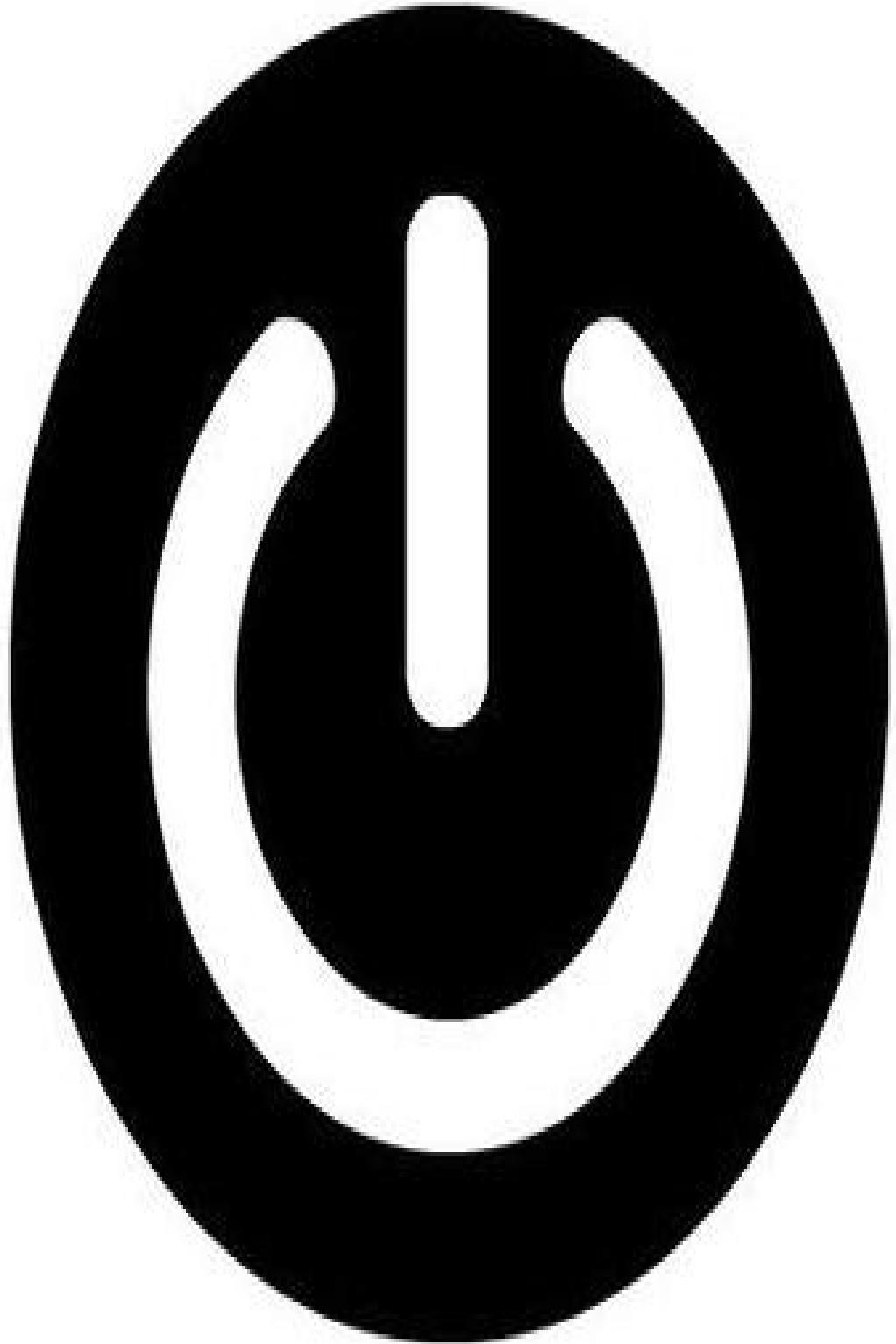


Figura 2.4: Boot

A arquitetura do Spring Boot é formada pelos componentes:

CLI — O Spring Boot CLI é uma ferramenta de linha de comando que facilita a criação de protótipos através de scripts em Groovy.

Starters — Conjunto de componentes de dependências que podem ser adicionados aos nossos sistemas.

Autoconfigure — Configura automaticamente os componentes carregados.

Actuator — Ajuda a monitorar e gerenciar as aplicações publicadas em produção.

Tools — Uma IDE customizada para o desenvolvimento com Spring Boot.

Samples — Dezenas de exemplos de implementações disponíveis para uso.

Veremos os componentes com detalhes no decorrer do livro. A seguir, veremos mais sobre o componente Tools.

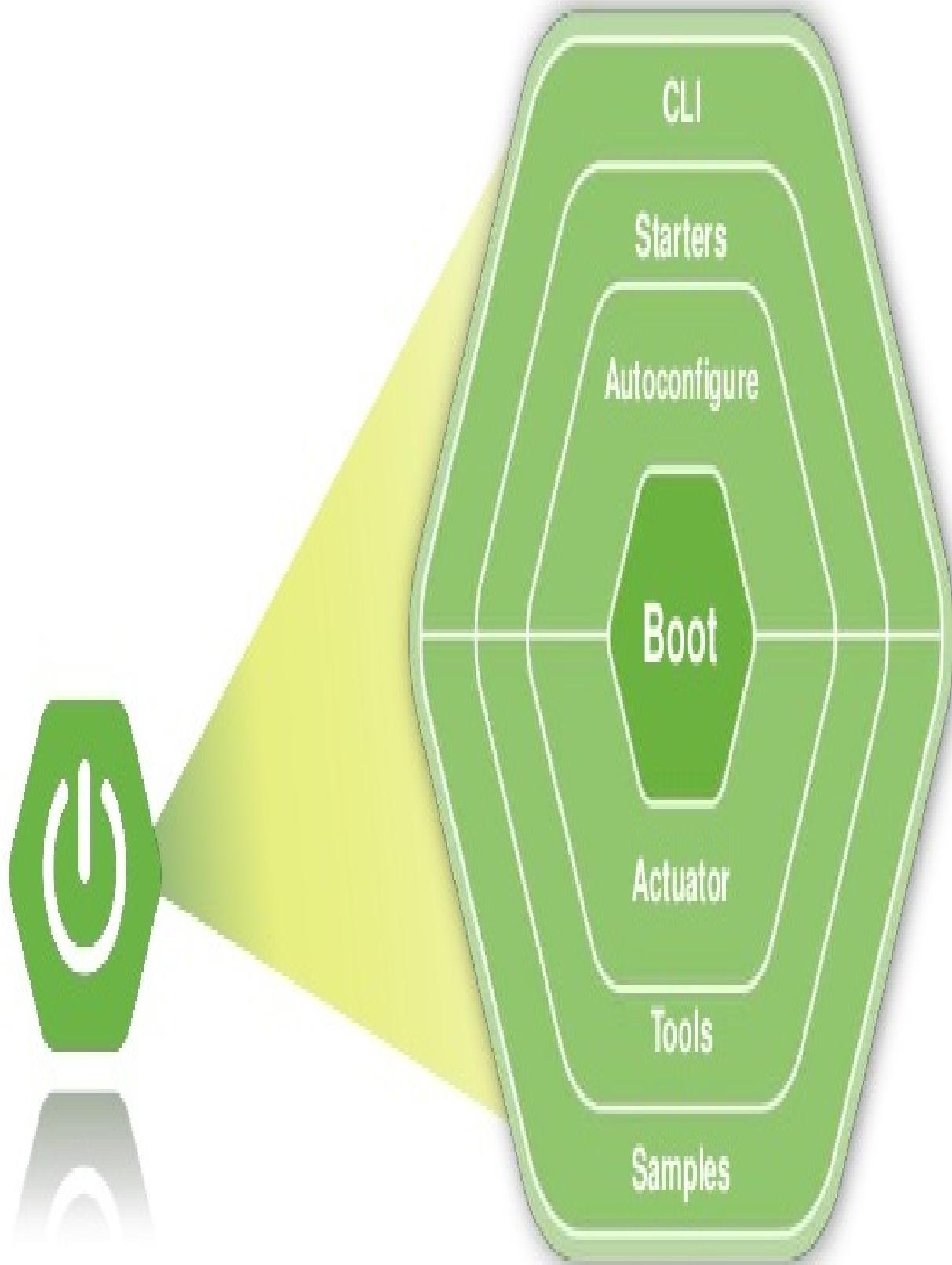


Figura 2.5: Arquitetura do Spring Boot

2.3 Nossa ferramenta

Neste livro, vamos usar a IDE oficial, gratuita e mantida pela Pivotal, para trabalhar com Spring Boot: o Spring Tools Suite ou simplesmente STS. O Spring Tools Suite é um conjunto de ferramentas baseadas no Eclipse para o desenvolvimento de aplicações Spring, disponível em: <https://spring.io/tools>.

Existem duas opções: a ferramenta completa pronta para uso e um plugin para um Eclipse já instalado. Neste livro, vamos usar a primeira por ser a mais simples. Se você já possui o Eclipse instalado, pode usar a segunda opção, o resultado final será o mesmo.

A instalação não tem mistério, é só descompactar o arquivo (não tem instalador) e usar. Dentro do diretório descompactado, execute o executável chamado SpringToolSuite4. Um usuário do Eclipse sente-se em casa e nota algumas diferenças da versão oficial. Temos um botão Boot Dashboard e um espaço novo para gerenciar as aplicações criadas com Spring Boot. Além de ter suporte integrado a todo ecossistema do Spring, ele já está integrado ao Maven, Gradle e Git. Veja mais detalhes em <https://spring.io/tools>.

 Spring Tool Suite | 4

2007-2020 Pivotal Software Inc. All Rights Reserved. Java and all Java-related trademarks and logos are trademarks or Oracle Corporation. In all U.S. other countries, or both. Eclipse is a trademark of the Eclipse Foundation, Inc.

by Pivotal.

Figura 2.6: Iniciando o Spring Tools Suite



workspace-sts - Spring Tool Suite 4

File Edit Navigate Search Project Run Window Help



Package Explorer X

There are no projects in your workspace.

To add a project:

- [Create a Java project](#)
- [Create new Spring Starter Project](#)
- [Import Spring Getting Started Content](#)
- [Create a project...](#)
- [Import projects...](#)



Boot Dashboard X



Type tags, projects, or working set names to match

local

Problems X

0 items

Description

Figura 2.7: Boot Dashboard

-

O

Apache Maven

é uma ferramenta usada para gerenciar as dependências e automatizar seus builds. Veja mais informações em <http://maven.apache.org>.

-

2.4 Nosso primeiro programa

Vamos criar uma aplicação web mais simples possível e analisar o seu resultado. Para criar um novo projeto, acessamos a opção File, New e Spring Starter Project.



Figura 2.8: Novo projeto

Inicialmente, temos algumas opções padrão de um projeto novo no Maven, como nome, versão do Java, entre outras coisas. Até aqui, sem novidades.

New Spring Starter Project

Service URL:

Name:

Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

Working sets

Add project to working sets

Working sets:

Figura 2.9: Informações padrão do novo projeto

Nesta parte, temos o diferencial: escolhemos as dependências de que o nosso projeto precisa para funcionar. Ao digitarmos no campo destacado web, são exibidas todas as opções relacionadas. Como o nosso exemplo é bem simples, vamos selecionar apenas a opção Spring Web; em outros exemplos usaremos outras opções.

New Spring Starter Project Dependencies

Spring Boot Version: 2.4.1

Available: Selected:

web

- ▼ Messaging
 - WebSocket
- ▼ Template Engines
 - Thymeleaf
 - Apache Freemarker
- ▼ Testing
 - Testcontainers
- ▼ Web
 - Spring Web
 - Spring Reactive Web
 - Spring Web Services
 - Jersey
 - Vaadin

X Spring Web

Make Default Clear Selection

?

< Back Next >

Finish

Cancel

Figura 2.10: Escolhendo dependências do projeto

Clicando em Finish, o projeto será criado. Mas existe a opção Next, que exibe o link do site Spring Initializr (<http://start.spring.io>) com alguns parâmetros.

Na verdade, o que o Eclipse faz é chamar esse site passando os parâmetros e baixar o projeto compactado. É possível fazer a mesma coisa acessando o link via web browser.



New Spring Starter Project

Site Info

Base Url

<https://start.spring.io/starter.zip>

Full Url

<https://start.spring.io/starter.zip?name=demo&groupId=com.example&artifactId=demo&version=0.0.1-SNAPSHOT&description=Demo+project+for+Spring+Boot&packageName=com.example.demo&type=maven-project&packaging=jar&javaVersion=11&language=java&bootVersion=2.4.1&dependencies=web>



< Back

Next >

Finish

Cancel

Figura 2.11: Parâmetros de URL da criação do projeto

Vamos subir o projeto criado. Para tal, use a opção Run As no projeto e depois Spring Boot App. Outra alternativa mais simples é selecionar a aplicação no dashboard e clicar no botão Start.

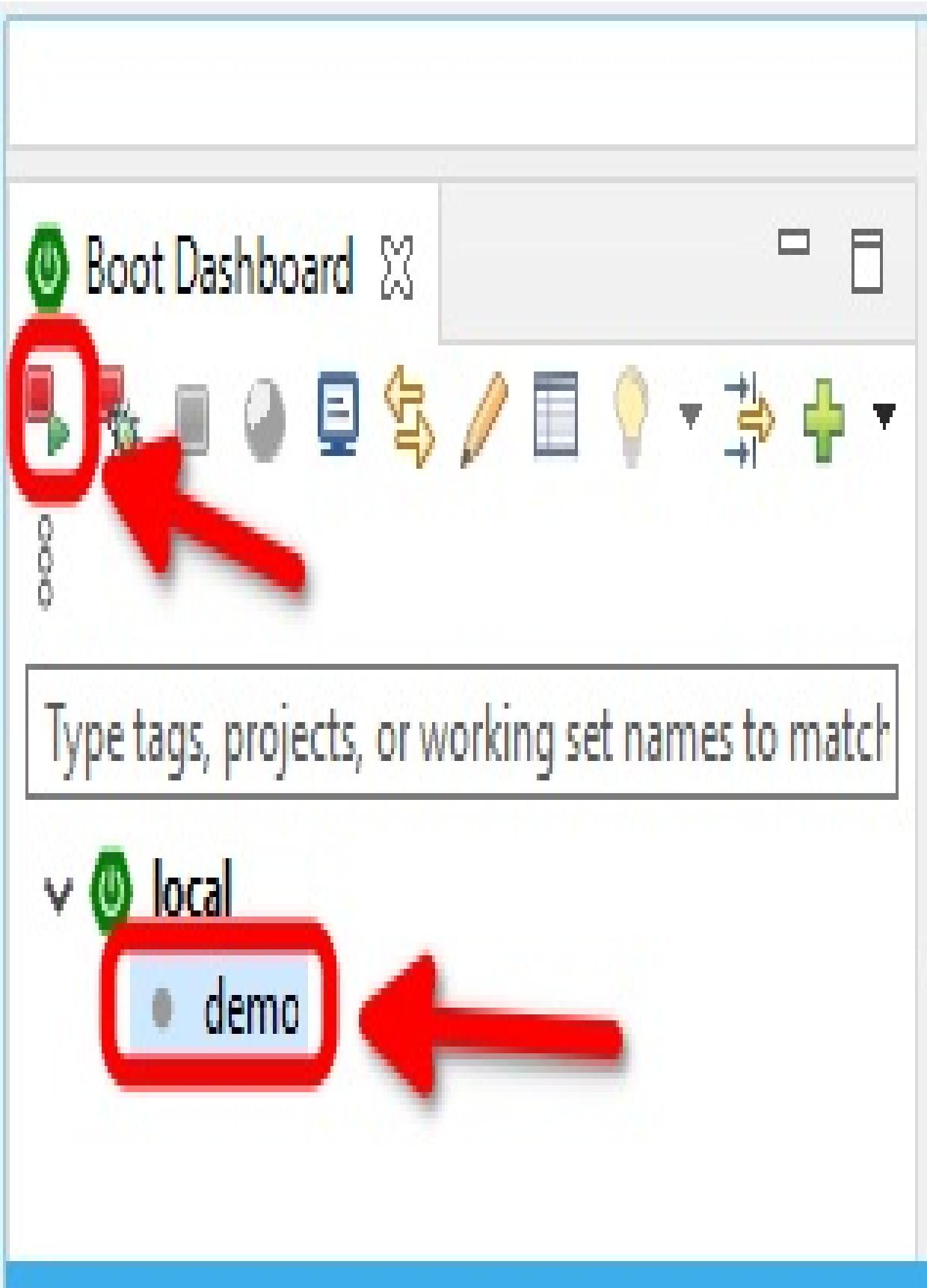
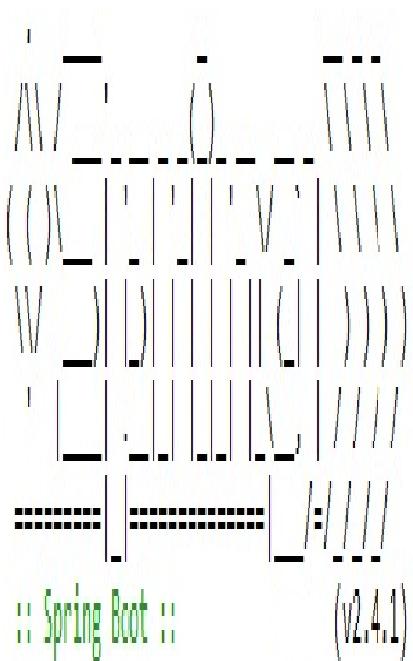


Figura 2.12: Subindo o projeto

Ao subir o projeto, aparece no log que uma instância do Apache Tomcat está no ar na porta 8080.



```
INFO 4884 --- [           main] com.example.demo.DemoApplication      : Starting DemoApplication using Java 15.0.1 on MSEDEGW
INFO 4884 --- [           main] com.example.demo.DemoApplication      : No active profile set, falling back to default profiles: [Tomcat]
INFO 4884 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
INFO 4884 --- [           main] o.apache.catalina.core.StandardService : Starting service [tomcat]
INFO 4884 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.41]
INFO 4884 --- [           main] o.a.c.r.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
INFO 4884 --- [           main] w.s.c.ServletWebApplicationContext        : Root WebApplicationContext: initialization completed:
INFO 4884 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
INFO 4884 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context p
INFO 4884 --- [           main] com.example.demo.DemoApplication      : Started DemoApplication in 4.273 seconds (JVM running
```

Figura 2.13: Spring Boot no console do Eclipse

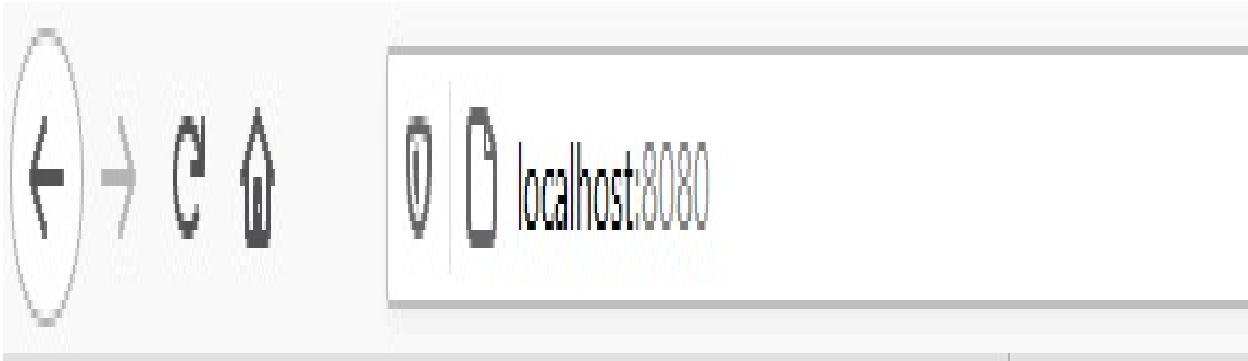
Outra alternativa para subir o projeto fora do Eclipse é usar o Maven, com o comando mvn spring-boot:run.

cmd.exe - mvn spring-boot:run

```
C:\workspace-sts\demo>mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example:demo >-----
[INFO] Building demo 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----[INFO]
[INFO] >>> spring-boot-maven-plugin:2.4.1:run (default-cli) >
[INFO]
```

Figura 2.14: Spring Boot no console do Windows

Abrindo o endereço `http://localhost:8080` no web browser, recebemos uma mensagem de página não encontrada (o que é esperado, já que não definimos nenhuma).



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Dec 27 16:37:30 PST 2020

There was an unexpected error (type=NotFound, status=404).

Figura 2.15: Página inicial no web browser

Vamos agora criar nossa página inicial adicionando ao projeto a seguinte classe:

```
package com.example;

import org.springframework.web.bind.annotation.*;

@RestController

public class PaginaInicial
```

```
@GetMapping("/")
)

String home()
{
    return "Olá Spring Boot!!"
};

}
}
```

Em seguida, para o projeto pegar essa classe nova, clicamos novamente no botão Start. Como resultado, temos a mensagem exibida na página inicial.



Ola Spring Boot!!

Figura 2.16: Nova página inicial

Vamos avaliar o que aconteceu aqui: escolhemos apenas que queríamos uma aplicação com dependência web e conseguimos ter rodando rapidamente uma página web, que responde ao conteúdo escolhido sem a necessidade de instalar nada a mais.

Qual servidor web foi usado? Qual versão do Spring foi escolhida? Quais as dependências de que o projeto precisa para funcionar? Tudo isso foi gerenciado automaticamente pelo Spring Boot.

2.5 Ajustando os parafusos

Até mesmo uma aplicação simples precisa de uma customização. No Spring Boot existe o esquema de convenção sobre configuração. Ou seja, sem nenhum ajuste, a aplicação funciona com valores predefinidos, e que, se você quiser, pode mudar via configuração.

A simplicidade do Spring Boot existe até em sua customização. Ela pode ser feita via Java ou via arquivo de propriedades `application.properties`. Veremos ambos os exemplos nos próximos capítulos.

Nesse arquivo, existem centenas de configurações nas quais é possível fazer ajustes. A documentação oficial explica cada uma delas (<http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>).

Uma simples customização

Para um simples teste de customização, vamos alterar a porta padrão do servidor de 8080 para 9000. Isso é feito adicionando o parâmetro `server.port=9000` no arquivo `application.properties`, localizado no diretório `src/main/resources` do projeto.

workspace-sts - demo/src/main/resources/application.properties - Spring Tool Suite 4

File Edit Navigate Search Project Run Window Help

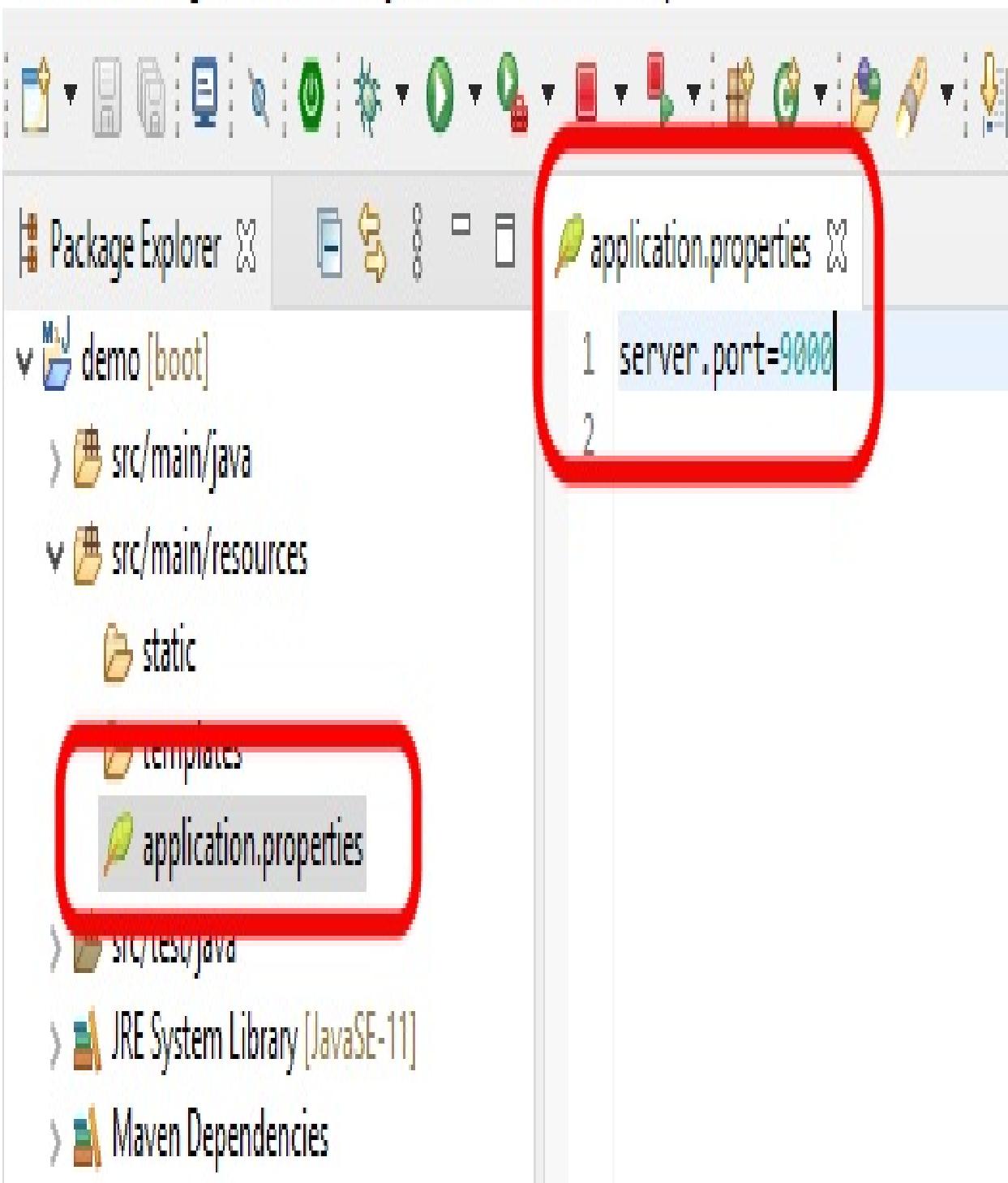


Figura 2.17: Arquivo application.properties

Em seguida, apenas com um restart na aplicação, verificamos a nova porta 9000 sendo usada.

```
[main] com.example.demo.DemoApplication : Starting DemoApplication using Java 11 on MSEdge/100.0.1185.152
[main] com.example.demo.DemoApplication : No active profile set, falling back to default profile
[main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9000 (http)
[main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
[main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.41]
[main] o.a.c.c.C[Tomcat][localhost][/] : Initializing Spring embedded WebApplicationContext
[main] w.s.c.ServletWebApplicationContext : Root WebApplicationContext: initialization completed in 1 ms
[main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
[main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9000 (http) with context pa
[main] com.example.demo.DemoApplication : Started DemoApplication in 4.857 seconds (JVM running:
```

Figura 2.18: Porta 9000 em execução

Outras IDEs

O IntelliJ IDEA oferece a geração de aplicações Spring Boot apenas na versão paga (Ultimate), portanto uma opção para funcionar na versão gratuita (Community) é acessar o site Spring Initializr com o web browser, escolher as opções desejadas e depois importar o projeto Maven.

O Visual Studio Code, entre opções independentes de Spring Boot, também tem um plugin open source oficial da Pivotal chamado Spring Boot Tools.

2.6 O que aprendemos

Certifique-se de que você aprendeu:

- a instalar o Spring Suite Tools;
- sobre a arquitetura geral do Spring Boot;
- a customizar os parâmetros do Spring;
- a fazer um programa simples.

No próximo capítulo, vamos criar um sistema mais complexo e mostrar as facilidades que o Spring Boot oferece aos desenvolvedores.

Capítulo 3

Primeiro sistema

O nosso contexto

Aprender uma nova tecnologia é sempre um desafio. Entretanto, ao aplicarmos exemplos do dia a dia, o aprendizado fica mais fácil. Portanto, vamos apresentar o nosso contexto, que certamente terá algumas semelhanças com o seu.

Temos aqui o empresário Rodrigo Haole que criou a startup Green Dog, uma empresa especializada em fast-food de cachorro-quente vegetariano. Ele tem um sistema comprado (sem os códigos-fontes) que faz o controle de pedidos de delivery e armazena-os em um banco de dados. Este precisa ser substituído por uma aplicação Java o mais rápido possível.

POC (aplicação de prova de conceito)

Antes de tomar a decisão de qual framework Java utilizar, Rodrigo decidiu fazer uma prova de conceito criando uma aplicação simples, envolvendo um acesso ao banco de dados e uma tela web simples de consulta.

Percebendo a grande quantidade de configurações do Spring Boot, Rodrigo teve a ideia de fazer uma aplicação para consultar as propriedades existentes em uma tela web simples. Dessa maneira, seria possível fazer uma aplicação para validar a facilidade e a produtividade do Spring Boot.

Temos, portanto, um escopo de banco de dados (das propriedades) exposto, via serviços ReST, e sendo consumido na tela web, via JavaScript.

Para cada tecnologia que Rodrigo precisa para sua aplicação, o Spring Boot tem uma configuração predefinida chamada starter.

Os starters do Spring Boot

Os starters são configurações predefinidas da tecnologia desejada para usar em seu projeto. A utilização de um starter facilita muito o desenvolvimento, pois ajusta automaticamente todas as bibliotecas e versões, livrando o desenvolvedor dessa árdua tarefa.

Não devemos nos preocupar com o funcionamento deles, nós apenas usamos e é garantido que não ocorrerá nenhum problema de bibliotecas.

No exemplo inicial, usamos o starter web (veja a lista completa no Apêndice A — Starters), que nos proporciona apenas uma aplicação web simples. Para mais opções, é necessário usar mais starters.

3.1 Tecnologia Java da POC

Com o nosso escopo definido em uma aplicação Java web tradicional (sem nenhum framework), a implementação poderia ser feita com:

Java Servlets para os serviços ReST.

JDBC para acessar o banco de dados.

Manualmente converter os dados em JSON.

Manualmente carregar os dados para o banco de dados.

JSP com Bootstrap para layout.

JSP com JQuery para chamar os serviços ReST.

Servlet Container para rodar a aplicação.

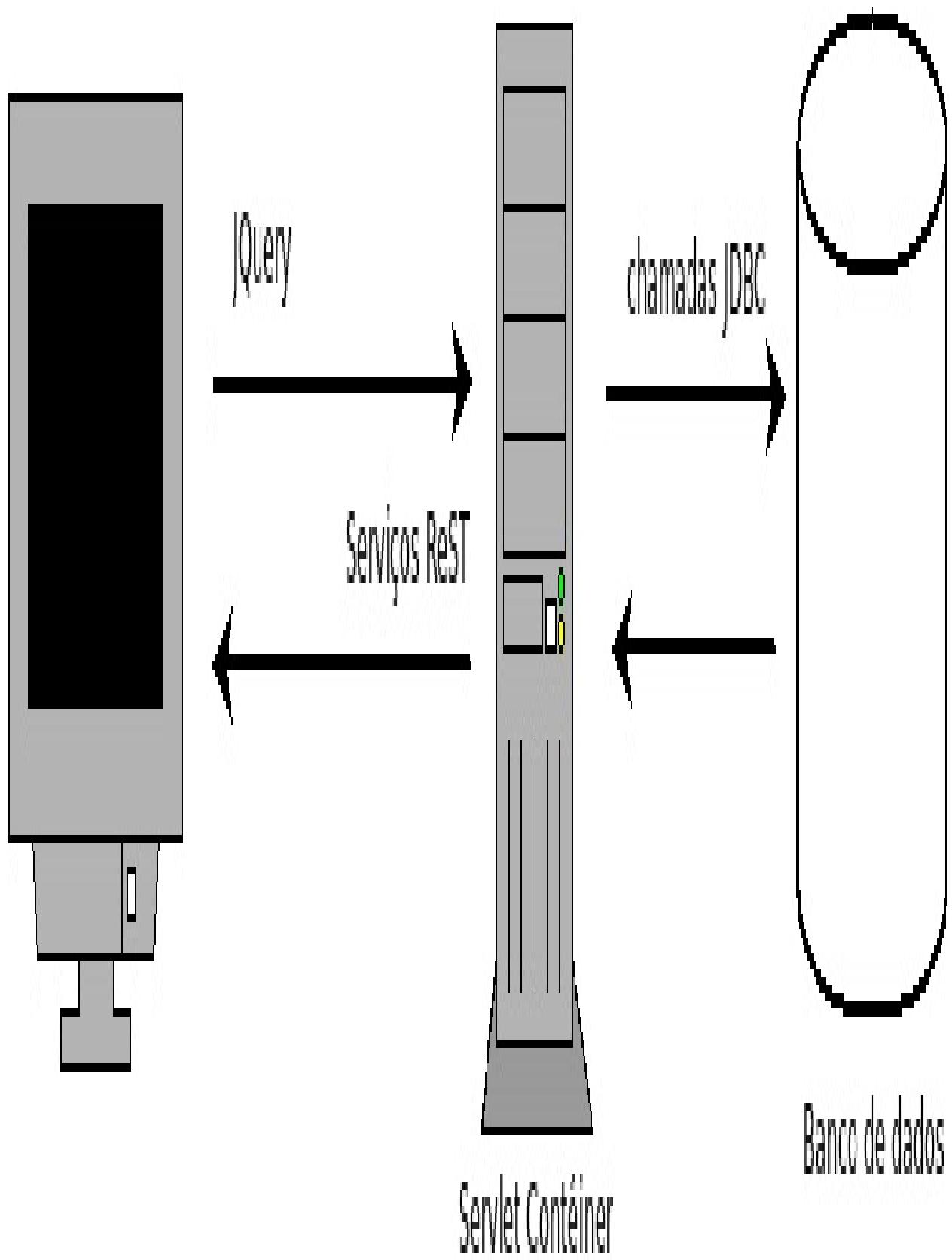


Figura 3.1: Aplicação simples sem framework Java

Utilizando Spring Framework, podemos atualizar essa lista para:

Spring MVC para os serviços ReST.

Spring MVC para converter os dados em JSON.

Spring Data para acessar o banco de dados.

Spring Data para carregar os dados para o banco de dados.

JSP com Bootstrap para layout.

JSP com JQuery para chamar os serviços ReST.

Servlet Container para rodar a aplicação.

-

Spring MVC

é o projeto que implementa o padrão MVC, além de fornecer suporte à injeção de dependência, JPA, JDBC, entre outras opções. Veja mais informações em <https://spring.io/projects/spring-framework>.

Já o Spring Data fornece um modelo de programação consistente e familiar, baseado no Spring, para acesso aos dados. Veja mais informações em <https://spring.io/projects/spring-data>.

Para fazermos as operações da lista anterior com o Spring Boot vamos usar os seguintes starters:

spring-boot-starter-web;

spring-boot-starter-data-jpa;

spring-boot-starter-data-rest;

spring-boot-starter-tomcat.

Iniciaremos criando um novo projeto como o exemplo no capítulo anterior. Então, acessamos a opção File, New e Spring Starter Project. Em seguida, informamos valores para o projeto.

Type:	Maven	Packaging:	Jar
Java Version:	11	Language:	Java
Group	com.boaglio.casadocodigo		
Artifact	springbootproperties		
Version	0.0.1-SNAPSHOT		
Description	lista de propriedades do Spring Boot		
Package	com.boaglio.casadocodigo		

Figura 3.2: Valores iniciais do projeto

E escolhemos as opções que utilizaremos:

Available:

rest

▼ NoSQL

Spring Data Elasticsearch (Access+Driver)

▼ Spring Cloud Discovery

Eureka Discovery Client

▼ Spring Cloud Routing

OpenFeign

▼ Testing

Spring REST Docs

Contract Stub Runner

▼ Web

Spring Web

Rest Repositories

Rest Repositories HAL Explorer

Rest Repositories HAL Browser

Spring HATEOAS

Jersey

Selected:

X Spring Data JPA

X Spring Web

X Rest Repositories

Make Default

Clear Selection

Figura 3.3: Escolhendo as dependências do projeto

3.2 Análise dos dados da POC

Temos o projeto criado, agora analisaremos os dados. Vamos trabalhar com as propriedades no seguinte formato: uma categoria tem várias subcategorias, que têm várias propriedades. Cada propriedade tem um nome, um valor e uma descrição. A nossa tabela de banco de dados receberá a carga dessa maneira, como a propriedade server.port exibida anteriormente:

```
insert into
propriedade
(categoria, nome, valor, descricao)
values ('Server properties', 'server.port'
,
'8080', 'Server HTTP port.');
```

Portanto, podemos criar nossa classe de domínio, que representará essa informação desta maneira:

```
@Entity public class Propriedade
{
```

```
@Id private String nome; private String valor; private String descricao; private  
String categoria;  
  
/* getters e setters */
```

A carga inicial será feita automaticamente pelo Spring Boot. Precisaremos apenas colocar o arquivo SQL de carga chamado data.sql no diretório de resources.

Para facilitar o nosso trabalho, vamos usar o banco de dados em memória H2 e colocar as propriedades a seguir no arquivo de propriedades application.properties.

```
spring.h2.console.enabled=true  
spring.datasource.url=jdbc:h2:mem:meuBancoDeDados
```

Para o serviço que será chamado pela tela web, será criada uma rotina de busca pelo nome ou categoria com Spring Data:

```
public interface PropriedadeRepository  
extends PagingAndSortingRepository<Propriedade, String> {
```

```
@Query("Select c from Propriedade c  
where c.nome like %:filtro%  
order by categoria,nome")  
public List<Propriedade> findByFiltro  
(@Param("filtro") String filtro);  
  
}
```

E para chamar esse serviço, criamos um controller ReST do Spring MVC para repassar a chamada:

```
@RestController @RequestMapping("/api") public class PropriedadeController  
{  
  
    @Autowired private  
    PropriedadeRepository repository;  
  
    @GetMapping("/find") List<Propriedade> findByFiltro  
    (@RequestParam("filtro") String filtro)  
    {
```

```
return repository.findByFiltro(filtro);  
}  
  
}
```

A parte do back-end da aplicação em Java terminou. Agora precisamos fazer o front-end em HTML, que chamará os serviços usando JavaScript.

O arquivo index.html deverá ficar dentro do diretório resources/static, pois o Spring Boot por convenção espera que esses arquivos estejam lá. Dentro do diretório, teremos uma rotina JavaScript que chama o serviço de busca de propriedades e monta o HTML dinamicamente na tela com o resultado:

```
$(document).ready(function ()  
{  
  
    var  
        options = {  
            url:  
                function (filtro)
```

```
{  
  
return '/api/find/'  
;  
  
},  
  
getValue:  
  
function (element)  
{  
  
    valorPadrao=  
    ""  
  
    if  
        (element.valor)  
            valorPadrao =  
                "( default: "+element.valor+") "  
  
    return "["+element.nome + "] <i>"  
        + element.categoria  
        +
```

```
" </i> - <b>" + element.nome +</b> "
```

```
+ valorPadrao +
```

```
" - "
```

```
+ element.descricao ;
```

```
},
```

```
ajaxSettings: {
```

```
    dataType:
```

```
'json'
```

```
,
```

```
    method:
```

```
'GET'
```

```
,
```

```
    data: {
```

```
    }
```

```
},
```

```
    preparePostData:
```

```
function (data)
```

```
{
```

```
    data.filtro = $(
```

```
'#search'
```

```
).val();
```

```
return  
data;  
,  
theme:  
"square"  
,  
list: {  
maxNumberOfElements:  
15  
  
}  
requestDelay:  
400  
  
};  
$(  
'#search'  
).easyAutocomplete(options);  
});
```

E finalmente, para informar ao Spring Boot que usaremos uma aplicação web baseada em Servlets, na nossa classe SpringbootpropertiesApplication usamos a herança da classe SpringBootServletInitializer e adicionamos uma implementação para identificar os arquivos da pasta resources.

```
@SpringBootApplication public class SpringbootpropertiesApplication
```

```
extends SpringBootServletInitializer
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
    SpringApplication.run(
```

```
        SpringbootpropertiesApplication.class, args);
```

```
}
```

```
@Override protected
```

```
    SpringApplicationBuilder
```

```
    configure(SpringApplicationBuilder application)
```

```
{
```

```
    return
```

```
        application.sources(
```

```
SpringbootpropertiesApplication.class);  
}
```

```
}
```

E adicionaremos as novas dependências ao arquivo pom.xml:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-tomcat</artifactId> <scope>provided</scope>  
</dependency> <dependency>
```

```
<groupId>com.h2database</groupId>
```

```
<artifactId>h2</artifactId> </dependency> <dependency>
```

```
<groupId>org.webjars</groupId>
```

```
<artifactId>bootstrap</artifactId>
```

```
<version>4.5.2</version> </dependency> <dependency>
```

```
<groupId>org.webjars</groupId>
```

```
<artifactId>jquery</artifactId>
```

```
<version>3.5.1</version> </dependency> <dependency>
```

```
<groupId>org.webjars.bower</groupId>
```

```
<artifactId>EasyAutocomplete</artifactId>
```

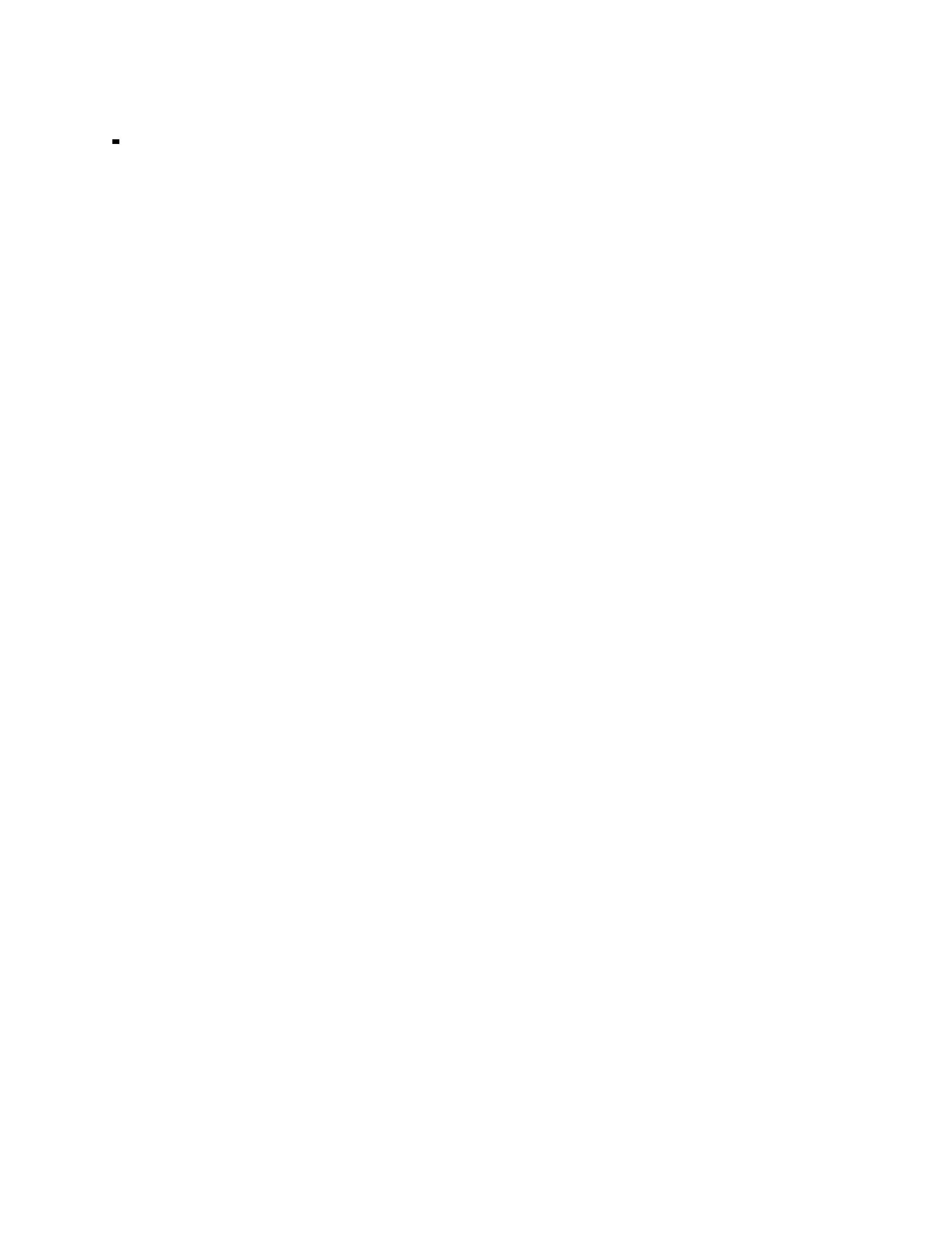
```
<version>1.3.3</version> </dependency>
```

```
-
```

As

WebJars

são bibliotecas web (como jQuery ou Bootstrap) empacotadas em arquivos JAR. Elas são uma opção interessante para aplicações sem acesso à internet, pois não é necessário nenhum download. Usamos algumas no nosso projeto. Veja mais opções em <http://www.webjars.org>.



3.3 Usando a aplicação

Depois de subir a aplicação pelo Eclipse (ou manualmente com mvn spring-boot:run), observamos no console as rotinas do Hibernate subindo e executando o script de carga.

```
o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:datasource1'; h2ConsoleAvailable=true
```

```
o.hibernate.jpa.internal.util.LogHelper : HHH000304: Processing PersistenceUnitInfo {name: default}
```

```
org.hibernate.Version : HHH000412: Hibernate Core version 5.4.25.Final
```

```
o.hibernate.annotations.common.Version : HCA000001: Hibernate Commons Annotations {5.1.2.Final}
```

```
org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
```

```
o.h.e.t.j.p.i.PlatformInitiator : HHH000000: Using JtaPlatform implementation: [org.hibernate.jta.platform.internal.JBoss Seam Platform]
```

```
j.LocalContainerEntityManagerFactoryBean : JBoss Seam Platform initialized EntityManagerFactory for persistence unit 'datasource1'.
```

```
JpaBaseConfiguration$JpaBaseConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly call openSession() or persist(...)
```

Figura 3.4: Log das operações do Hibernate

Em seguida, podemos fazer a busca no campo. Após pressionar Enter, o resultado aparece na mesma tela, sem reload da página.



spring® Spring Boot Properties

Buscar:

- [spring.webservices.path] *Integration properties* - **spring.webservices.path** (default: /services) - Path that serves as the base URI for the services.
- [spring.webservices.servlet.init.*] *Integration properties* - **spring.webservices.servlet.init.*** - Servlet init parameters to pass to Spring Web Services.
- [spring.webservices.servlet.load-on-startup] *Integration properties* - **spring.webservices.servlet.load-on-startup** (default: -1) - Load on startup priority of the Spring Web Services servlet.
- [spring.webservices.wsdl-locations] *Integration properties* - **spring.webservices.wsdl-locations** - Comma-separated list of locations of WSDLs and accompanying XSDs to be exposed as beans.

Figura 3.5: Buscando por propriedades de webservices

3.4 Usando o console H2

Ao buscarmos por h2, notamos que existe a propriedade `spring.h2.console.enabled`, que tem como valor padrão `false`. Mas no nosso arquivo de propriedades `application.properties`, já ajustamos essa propriedade para `true`.

Vamos acessar o console H2 para conseguirmos executar comandos SQL no banco de dados em memória através da URL `http://localhost:8080/h2-console/`. Esse caminho pode ser alterado com a propriedade `spring.h2.console.path`.

Preenchendo o campo de JDBC URL com o mesmo valor que colocarmos nas propriedades (`jdbc:h2:mem:meuBancoDeDados`), conseguimos acessar os dados:



① localhost:8080/h2-console/login.jsp?jsessionid=d9bc1a5f6f6dd54762

Português (Brasil) ▾

Preferências Tools Ajuda

Login

Configuração ativa:

Nome da configuração:

Classe com o driver:

JDBC URL:

Usuário:

Senha:

Figura 3.6: Entrar no console do H2

Dentro do console, podemos fazer consultas em SQL, como listar as propriedades que começam com spring.h2.

Auto commit | Max rows: 1000 | Auto complete Off | Auto select On

jdbc:h2:mem:meuBancoDeDados: Run | Run Selected | Auto complete | Clear SQL statement:

PROPRIADES

- ID
- CATEGORIA
- NOME
- VALOR
- DESCRICAO
- Indexes

INFORMATION_SCHEMA

Sequences

Users

H2 1.4.200 (2019-10-14)

```
select *
from propriedade
where nome like 'spring.h2%'
```

```
select *
from propriedade
where nome like 'spring.h2%';
```

ID	CATEGORIA	NOME	VALOR	DESCRICAO
376	Data properties	spring.h2.console.enabled	false	Whether to enable the console.
377	Data properties	spring.h2.console.path	/h2-console	Path at which the console is available.
378	Data properties	spring.h2.console.settings.trace	false	Whether to enable trace output.
379	Data properties	spring.h2.console.settings.web-allow-others	false	Whether to enable remote access.

(4 rows, 16 ms)

Figura 3.7: Consultando propriedades via SQL no console do H2

3.5 O que aprendemos

Os códigos-fontes desse projeto estão no GitHub, em
<https://github.com/boaglio/spring-boot-propriedades-casadocodigo>.

Certifique-se de que você aprendeu:

- a utilizar Spring Data e Spring MVC com Spring Boot;
- a consultar as propriedades do arquivo de configuração application.properties;
- a fazer carga inicial em uma base H2;
- a usar o H2 console.

No próximo capítulo, vamos criar um sistema mais robusto e com uma arquitetura mais complexa.

Capítulo 4

Explorando os dados

Depois que Rodrigo validou que, com o Spring Boot, é possível fazer muita coisa com pouco código, ele precisa começar a planejar sua aplicação principal. Os requisitos de seu negócio são:

Tela de cadastro de clientes.

Tela de cadastro de itens.

Tela de cadastro de pedidos.

Tela para fazer pedido com opção de oferta.

Notificar o cliente do novo pedido recebido.

Os requisitos técnicos são:

Ser um projeto 100% web.

Usar um banco de dados MySQL.

Expor serviços via ReST.

Ter alta disponibilidade para fazer novos pedidos.

Ter uma tela para validar se o ambiente está ok.

Usar AngularJS na tela de novo pedido.

Com esses requisitos, Rodrigo montou o seguinte diagrama de classes:

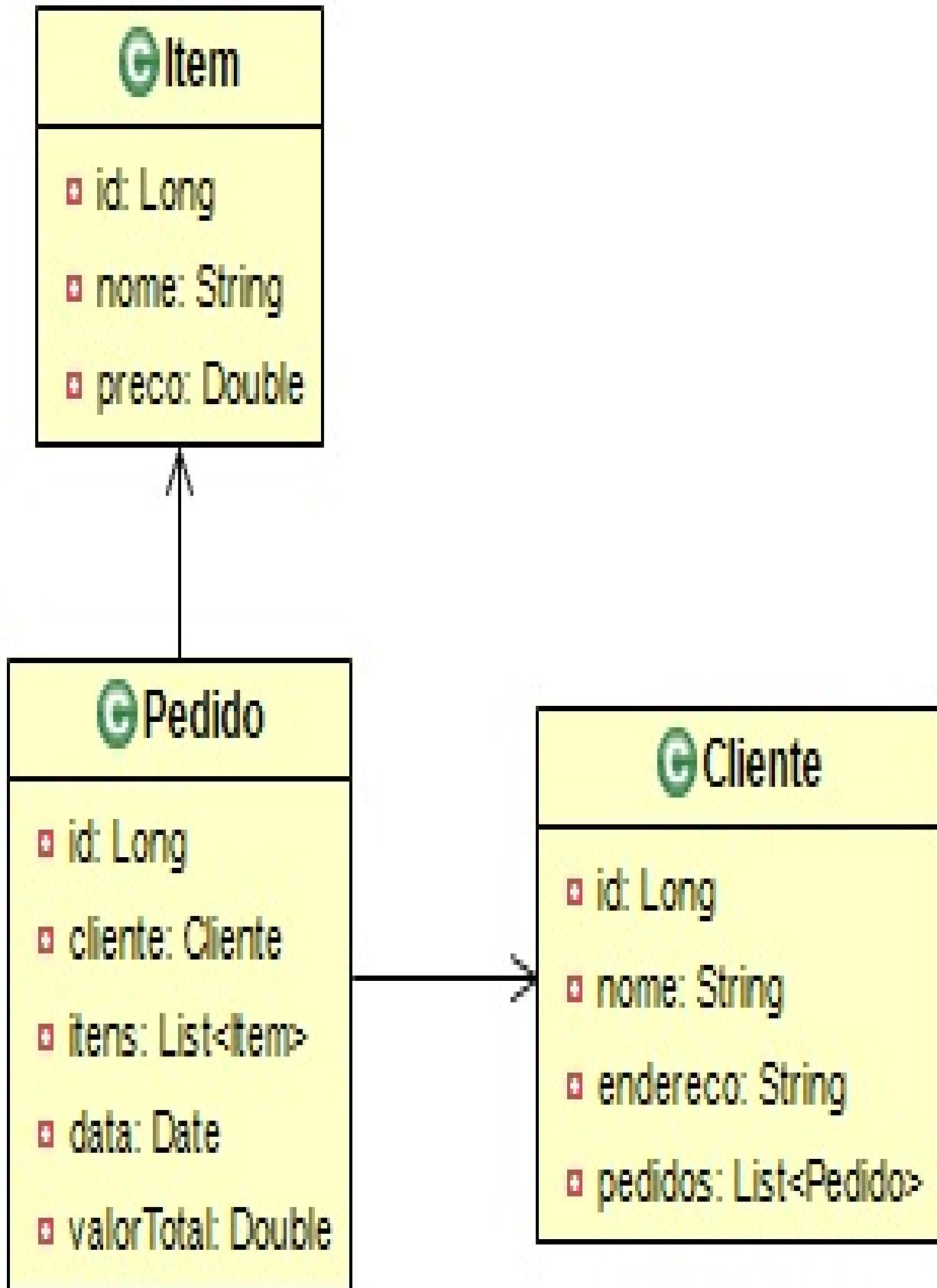


Figura 4.1: Diagrama de classes

Esse diagrama mostra a clássica regra de um conjunto de clientes, em que cada um deles pode fazer um pedido, e cada pedido pode conter um ou mais itens.

O novo projeto

Sabemos que precisamos de banco de dados, então vamos usar a opção de JPA. Precisaremos também de templates para gerar as páginas dinâmicas. Para tal, ficamos com a engine recomendada pela equipe do Spring, o Thymeleaf, que veremos com mais detalhes no próximo capítulo. Agora, detalharemos a implementação do back-end.

O Java Persistence API (ou simplesmente JPA) é uma API padrão da linguagem Java que descreve uma interface comum para frameworks de persistência de dados. Veja mais informações em
<https://www.oracle.com/java/technologies/persistence-jsp.html>.

No nosso projeto do Eclipse springboot-greendogdelivery, vamos escolher essas duas opções e também a opção de Rest Repositories. Se tudo der certo, o projeto terá dentro do seu pom.xml os seguintes starters:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-jpa</artifactId> </dependency>
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-rest</artifactId> </dependency>
<dependency>
```

```
<groupId>org.springframework.data</groupId>
```

```
<artifactId>spring-data-rest-webmvc</artifactId> </dependency> <dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-thymeleaf</artifactId> </dependency>
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-web</artifactId> </dependency> <dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-test</artifactId>
```

```
<scope>test</scope> </dependency>
```

4.1 As classes de domínio

Conforme o diagrama de classes, criamos três classes de domínio já com as anotações de validação. A classe pai de todo o sistema é a de clientes, que contém nome, endereço e uma lista de pedidos.

```
@Entity public class Cliente
```

```
{
```

```
    @Id @GeneratedValue
```

```
(strategy=GenerationType.IDENTITY)
```

```
    private
```

```
        Long id;
```

```
        @NotNull @Length(min=2, max=30
```

```
, message=
```

```
"O tamanho do nome deve ser entre {min} e {max} caracteres")
```

```
        private
```

```
        String nome;
```

```
@NotNull @Length(min=2, max=300  
,message=  
"O tamanho do endereço deve ser entre {min} e {max} caracteres") private  
String endereco;  
  
@OneToMany(mappedBy = "cliente",fetch = FetchType.EAGER)  
@Cascade(CascadeType.ALL) private List<Pedido> pedidos;
```

A classe de pedidos está ligada ao cliente e tem, além da data e o valor, uma lista de itens:

```
@Entity public class Pedido  
{  
  
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private  
    Long id;  
  
    @ManyToOne(optional = true) private  
    Cliente cliente;
```

```
@ManyToMany @Cascade(CascadeType.MERGE) private  
List<Item> itens;
```

```
@DateTimeFormat(pattern = "dd-MM-yyyy") private
```

```
Date data;
```

```
@Min(1) private Double valorTotal;
```

Finalmente, a classe de item, com o nome e o preço:

```
@Entity  
public class Item {
```

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id;
```

```
@NotNull  
@Length(min=2, max=30,  
message="O tamanho do nome deve ser entre {min} e  
{max} caracteres")
```

```
private String nome;
```

```
@NotNull
```

```
@Min(value=20,message="O valor mínimo deve ser {value} reais")
```

```
private Double preco;
```

4.2 Repositórios

Com as classes de domínio, criaremos as classes de repositório.

Com a facilidade do Spring Data, gerenciar conexão, chamar PreparedStatement e fazer um loop para atribuir o ResultSet a um objeto são coisas do passado; tudo isso é implementado e usamos apenas interfaces.

```
@Repository public interface ClienteRepository
```

```
extends JpaRepository<Cliente, Long>
```

```
{
```

```
}
```

```
@Repository public interface PedidoRepository
```

```
extends JpaRepository<Pedido, Long>
```

```
{
```

```
}
```

```
@Repository public interface ItemRepository
```

```
extends JpaRepository<Item, Long>
```

```
{
```

```
}
```

- A anotação @Repository foi colocada por motivos didáticos, pois o Spring Boot já reconhece que as interfaces estendem JpaRepository e carrega todos os repositórios automaticamente.

-

4.3 Interfaces do tipo Runner

O Spring Boot oferece algumas interfaces para executar instruções logo após a aplicação subir, como a `CommandLineRunner` ou a `ApplicationRunner`. Ambas fazem a mesma coisa.

Quem implementa essa interface precisa declarar o método `run` e especificar as instruções que serão executadas. A diferença entre elas está nos argumentos do método `run`, nos quais o `CommandLineRunner` usa o clássico array de `Strings` para ler os argumentos de linha de comando e o `ApplicationRunner` usa o tipo `ApplicationArguments`, que tem outras informações internas do Spring Boot além do array de `Strings`.

Em um mesmo sistema podemos ter diversas classes desse tipo.

Neste exemplo, `CommandLineRunner` está fazendo a chamada:

```
$ java -jar app.jar --debug teste1 teste2
```

Veja também outras maneiras de iniciar uma aplicação Spring Boot, além da lista resumida das propriedades em Apêndice B — Propriedades.

Conseguimos listar os argumentos de uma maneira mais simples:

```
@Component public class ExemploCommandRunner implements  
CommandLineRunner
```

```
{
```

```
@Override
```

```
public void run(String ... args) throws Exception
```

```
{
```

```
    System.out.println(
```

```
        "===== ExemploCommandRunner ====="
```

```
);
```

```
for
```

```
(String arg : args)
```

```
    System.out.println(
```

```
        "[" + arg + "]")
```

```
);
```

```
    System.out.println(
```

```
        "===== ExemploCommandRunner ====="
```

```
);
```

}

No meio do log do Spring Boot reparamos na saída os parâmetros teste1 e teste2 passados ao 4:

===== ExemploCommandRunner =====

[--debug]

[teste1]

[teste2]

===== ExemploCommandRunner =====

Já usando ApplicationRunner , conseguimos trabalhar com os argumentos e com mais opções:

```
@Component public class ExemploAppRunner implements ApplicationRunner
```

{

@Override

```
public void run(ApplicationArguments args) throws Exception
{
    System.out.println(
"===== ExemploAppRunner ====="
);

boolean debug = args.containsOption("debug"
);
List<String> files = args.getNonOptionArgs();
System.out.println(
"Chamou com debug? "
+ debug);
System.out.println(files);
System.out.println(
"===== ExemploAppRunner ====="
);
}
```

No log do Spring Boot podemos observar a saída:

===== ExemploAppRunner =====

Chamou com debug?

true

[teste1, teste2]

===== ExemploAppRunner =====

Para o nosso exemplo de carga inicial, podemos implementar qualquer uma das duas interfaces.

4.4 Carga inicial

Como vimos no projeto anterior das propriedades do Spring Boot, ele executa facilmente scripts SQL. Mas nesse projeto faremos a carga via Java para aprendermos a usar essa opção no Spring Boot.

Criaremos uma classe RepositoryTest para fazer a mesma coisa em Java.

Inicialmente, para organizarmos os IDs do sistema, vamos definir algumas constantes para cliente, item e pedido:

```
private static final long ID_CLIENTE_FERNANDO = 11; private static final  
long ID_CLIENTE_ZE_PEQUENO = 22  
l;
```

```
private static final long ID_ITEM1 = 100l; private static final long ID_ITEM2 =  
101l; private static final long ID_ITEM3 = 102  
l;
```

```
private static final long ID_PEDIDO1 = 1000l; private static final long  
ID_PEDIDO2 = 1001l; private static final long ID_PEDIDO3 = 1002l;
```

Depois, declaramos o repositório de cliente e o método run, que o Spring Boot chamará para ser executado:

```
@Autowired private  
ClienteRepository clienteRepository;  
  
@Override public void run(ApplicationArguments applicationArguments)  
throws Exception {
```

Começamos declarando os clientes:

```
System.out.println(  
    ">>> Iniciando carga de dados..."  
);  
  
Cliente fernando =  
  
    new  
    Cliente(ID_CLIENTE_FERNANDO,
```

"Fernando Boaglio","Sampa"

);

Cliente zePequeno =

new

Cliente(ID_CLIENTE_ZE_PEQUENO,

"Zé Pequeno","Cidade de Deus");

Depois os três itens disponíveis para venda:

Item dog1=**new Item(ID_ITEM1,"Green Dog tradicional",25**

d);

Item dog2=

new Item(ID_ITEM2,"Green Dog tradicional picante",27

d);

Item dog3=

new Item(ID_ITEM3,"Green Dog max salada",30d);

Em seguida, a lista de pedidos:

```
List<Item> listaPedidoFernando1 = new  
ArrayList<Item>();  
listaPedidoFernando1.add(dog1);
```

```
List<Item> listaPedidoZePequeno1 =  
new  
ArrayList<Item>();  
listaPedidoZePequeno1.add(dog2);  
listaPedidoZePequeno1.add(dog3);
```

Depois, montamos as listas nos objetos de pedido:

```
Pedido pedidoDoFernando = new  
Pedido(ID_PEDIDO1,ffernando,  
listaPedidoFernando1,dog1.getPreco());  
ffernando.novoPedido(pedidoDoFernando);
```

```
Pedido pedidoDoZepequeno =  
new  
Pedido(ID_PEDIDO2,zePequeno,  
listaPedidoZePequeno1, dog2.getPreco()+dog3.getPreco()));
```

```
zePequeno.novoPedido(pedidoDoZepequeno);
```

```
System.out.println(  
    ">>> Pedido 1 - Fernando : "  
    +  
    pedidoDoFernando);  
  
System.out.println(  
    ">>> Pedido 2 - Zé Pequeno: "  
    +  
    pedidoDoZepequeno);
```

E, finalmente, persistimos no banco de dados:

```
clienteRepository.saveAndFlush(zePequeno);  
  
System.out.println(  
    ">>> Gravado cliente 2: "  
    +zePequeno);
```

```
List<Item> listaPedidoFernando2 =  
new
```

```
ArrayList<Item>();  
listaPedidoFernando2.add(dog2);  
Pedido pedido2DoFernando =  
new  
Pedido(ID_PEDIDO3,fernando,  
listaPedidoFernando2,dog2.getPreco());  
fernando.novoPedido(pedido2DoFernando);  
clienteRepository.saveAndFlush(fernando);  
System.out.println(  
">>> Pedido 2-Fernando:"  
+pedido2DoFernando);  
System.out.println(  
">>> Gravado cliente 1: "  
+fernando);  
}
```

É interessante notar que a classe ClienteRepository não tem nenhum método. Ela herdou o método saveAndFlush da classe JpaRepository, que faz parte do Spring Data.

Nesse projeto, usaremos inicialmente a base de dados em memória H2, e depois o MySQL. Mas para facilitar o desenvolvimento adicionaremos ao arquivo pom.xml dependências dos dois:

```
<dependency>
```

```
<groupId>mysql</groupId>
```

```
<artifactId>mysql-connector-java</artifactId> </dependency> <dependency>
```

```
<groupId>com.h2database</groupId>
```

```
<artifactId>h2</artifactId> </dependency>
```

Para verificar se a carga foi feita, adicionamos ao arquivo application.properties os valores do H2 console:

```
# h2
```

```
spring.h2.console.enabled=
```

```
true
```

```
spring.h2.console.path=/h2
```

```
# jpa
```

```
spring.jpa.show-sql=
```

```
true
```

```
spring.datasource.url=jdbc:h2:mem:greendog
```

Executando o projeto, verificamos no console o Hibernate executando os comandos SQL. Acessando o console no endereço <http://localhost:8080/h2/>, informamos o novo banco no JDBC URL:

← → C

① localhost:8080/h2/login.jsp?jsessionid=6bd9e1317c

English ▾

Preferences Tools Help

Login

Saved Settings:

Setting Name:

Driver Class:

JDBC URL:

User Name:

Password:

Figura 4.2: Entrar no console H2

E podemos verificar que a carga foi feita com sucesso:

The screenshot shows the H2 Database Browser interface. On the left, a tree view displays database objects: 'jdbc:h2:mem:greendog' (selected), 'CLIENTE', 'ITEM', 'PEDIDO', 'PEDIDO_ITENS', 'INFORMATION_SCHEMA', 'Sequences', 'Users', and 'H2 1.4.193'. The 'PEDIDO_ITENS' node is highlighted with a red box. At the top right, there are several icons: a red heart, a yellow star, a checkmark for 'Auto commit', a blue circle with a zero, a magnifying glass, a 'Max rows: 1000' dropdown, and a green arrow. Below these are buttons for 'Run', 'Run Selected', 'Auto complete', 'Clear', and 'SQL statement'. A red box highlights the 'Run' button and the SQL input field containing 'SELECT * FROM ITEM;'. The results pane at the bottom shows the query results in a table format. A red box highlights the entire results table. The table has columns 'ID', 'NOME', and 'PRECO'. The data is as follows:

ID	NOME	PRECO
1	Green Dog tradicional picante	27.0
2	Green Dog max salada	30.0
3	Green Dog tradicional	25.0
4	Green Dog tradicional picante	27.0

(4 rows, 3 ms)

At the bottom right of the results pane is an 'Edit' button.

Figura 4.3: Carga inicial no H2

4.5 O que aprendemos

Os fontes do projeto estão em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/dados>.

Certifique-se de que você aprendeu:

a criar classes de repositórios;

a fazer carga inicial em Java.

No próximo capítulo, vamos conhecer a engine de templates do Spring Boot para fazer páginas web, o Thymeleaf.

Capítulo 5

Explorando os templates

Rodrigo já conseguiu fazer a carga de seu sistema com o Spring Boot. Agora ele precisa criar uma página web para fazer os cadastros de clientes, itens e pedidos.

Só de lembrar de Servlet e JSP, Rodrigo já ficou muito preocupado, pois além da complexidade de controlar a navegação entre as páginas e trabalhar com parâmetros, para os designers, o JSP é uma engine complicada de trabalhar, pois sua sintaxe complexa quebra todo o layout.

O problema da navegação entre as páginas e parâmetros resolvemos com Spring MVC, mas o problema do layout conseguimos resolver apenas com o Thymeleaf.

5.1 Templates naturais com Thymeleaf

Thymeleaf é uma biblioteca Java que trabalha com templates XHTML/HTML5 para exibir dados sem quebrar a estrutura/ layout do documento. Essa capacidade de manter o layout é o que caracteriza o Thymeleaf como template natural, algo que as engines JSP, Velocity e FreeMarker não têm.

Veja este exemplo de uma página JSP renderizada dinamicamente, tudo ok:

This is a screenshot of a web browser displaying a JSP page. The URL in the address bar is `http://localhost:8080/thvsjsp/subscribejsp`. The page content is as follows:

This is a JSP

Email:

Type:

All emails
 Daily Digest

SUBSCRIBE ME!

Figura 5.1: Página JSP renderizada

Mas o layout dessa mesma página fica comprometido ao abrir a página JSP fonte no web browser:



```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form" %><%@ taglib prefix="s"  
uri="http://www.springframework.org/tags" %><%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"  
%><%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

This is a JSP



Figura 5.2: Página JSP fonte

A página renderizada no Thymeleaf se comporta igual ao JSP:

This screenshot shows a web browser window displaying a Thymeleaf template for a subscription form. The URL in the address bar is `http://localhost:8080/thvsjsp/subscribeth`. The page content is as follows:

This is a Thymeleaf template

Email

Type

All emails

Daily Digest

SUBSCRIBE ME!

Figura 5.3: Página Thymeleaf renderizada

Entretanto, a página de template fonte mantém o layout e é ideal para os designers trabalharem.

The screenshot shows a web browser window with the following details:

- Address Bar:** file:///C:/Documents and Settings/Daniel/... (partially visible)
- Toolbar:** Includes standard browser icons for back, forward, search, and navigation.
- Content Area:** Displays a Thymeleaf template with the following structure:
 - Email:** A text input field.
 - Type:** A label followed by two radio button options:
 - First type
 - Second Type
 - SUBSCRIBE ME!**: A large green rectangular button containing the text "SUBSCRIBE ME!".

Figura 5.4: Página Thymeleaf fonte

Essa "mágica" é possível porque o Thymeleaf trabalha com os seus argumentos dentro de atributos th, algo que o navegador ignora, mas que mantém o layout intacto.

Atributos

Os principais atributos do Thymeleaf são:

th:text — Exibe o conteúdo de uma variável ou expressão, como
th:text="\${cliente.id}".

th:href — Monta um link relativo, como th:href="@{/clientes/}".

th:src — Monta link relativo para atributo src, como
th:src="@{/img/oferta.png}".

th:action — Monta link relativo para tag atributo action de formulário dessa forma: th:action="@{/pedidos/(form)}".

th:field e th:object — Relaciona um objeto a um campo do formulário, como
th:object="\${cliente}" e th:field="*{id}".

th:value — Mostra o valor do atributo value do formulário, por exemplo:
th:value="\${cliente.id}".

th:if — Mostra conteúdo conforme resultado da expressão, como
th:if="\${clientes.empty}".

`th:each` — Mostra valores de uma lista, por exemplo: `th:each="item : ${itens}"`.

`th:class` — Aplica o estilo se a expressão for válida dessa forma:
`th:class="#{#fields.hasErrors(id)} ? 'field-error'"`.

Convenções para aplicações web

Como o Spring Boot gerencia o sistema e o servidor de aplicação, não podemos colocar nossas páginas do Thymeleaf, ou melhor, nossos templates em qualquer lugar. O Spring Boot oferece algumas convenções para as aplicações web, algo bem diferente do conhecido padrão JEE.

Dentro do diretório `src/main/resources`, temos três opções:

`public` — Contém páginas de erro padrão.

`static` — Contém os arquivos estáticos da aplicação (arquivos de estilo, JavaScript e imagens).

`templates` — Arquivos de template do Thymeleaf.

- ▼  **springboot-greendogdelivery [boot]**
- ▼  **src/main/java**
 - ▶  **com.boaglio.casadocodigo.greendogdelivery**
 - ▼  **src/main/resources**
 -  **public**
 -  **static**
 -  **templates**
 -  **application.properties**
 - ▶  **src/test/java**
- ▶  **JRE System Library [JavaSE-1.8]**
- ▶  **Maven Dependencies**

Figura 5.5: Convenções web do Spring Boot

5.2 Layout padrão do Thymeleaf

O Thymeleaf oferece a opção de layout padrão, que é chamado em cada página e pode ser usado para colocar os scripts comuns de layout.

No nosso projeto precisamos adicionar duas dependências:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-thymeleaf</artifactId> </dependency>
<dependency>
```

```
<groupId>nz.net.ultraq.thymeleaf</groupId>
```

```
<artifactId>thymeleaf-layout-dialect</artifactId> </dependency>
```

Seguindo a convenção do Spring Boot, dentro de src/main/resources/templates, criaremos o arquivo layout.html, iniciando com o cabeçalho do layout:

```
<!DOCTYPE html> <html xmlns:th="http://www.thymeleaf.org"
```

```
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"> <head>
```

```
        <title>Green Dog Delivery</title>
```

Depois, fazemos os scripts do Bootstrap e do JQuery:

```
<!-- Bootstrap -->
```

```
<link ... bootstrap.min.css">
```

```
<link ... bootstrap-theme.min.css">
```

```
<script ... bootstrap.min.js"></script>
```

```
<!-- JQuery -->
```

```
<link ... jquery-ui.css">
```

```
<script ... jquery-1.12.4.js"></script>
```

```
<script ... jquery-ui.js"></script>
```

```
<style>
```

```
...
```

```
</style> </head>
```

Em seguida, o cabeçalho com o logo:

```
<body>
```

```
<div class="container">
```

```
<nav class="navbar navbar-default navbar-static-top">
```

```
<div class="navbar-header">
```

```
<a class="navbar-brand" th:href="@{/}">
```

```

```

```
</a>
```

```
</div>
```

E, então, fazemos os links de menu:

```
<ul class="nav navbar-nav">
```

```
<li>
```

```
<a class="brand" href="https://www.casadocodigo.com.br">
```

Casa do Código

```
</a></li>
```

```
<li><a class="brand" href="https://www.boaglio.com">
```

boaglio.com

```
</a></li>
```

...

```
</ul> </nav>
```

Por último, os fragmentos do layout, nos quais serão adicionadas as páginas dinâmicas:

```
<h1 layout:fragment="header">cabeçalho falso</h1>
```

```
<div layout:fragment="content">content falso</div> </div> </body> </html>
```

Ao verificarmos o layout no browser, confirmamos que, apesar das tags, o formato fica alinhado corretamente:

```
← → ⌂ file:///home/fb/workspace/sts-livro/spring-boot-green-dog-delivery-casa-do-codigo/src/r
```



Casa do Código

boaglio.com

Spring Boot

Thymeleaf Layout

cabeçalho falso

content falso

Figura 5.6: Fonte do layout no browser

Para usarmos esse layout, usamos no cabeçalho do arquivo index.html a opção `decorate`. É ela que define o nome do template que a página está usando.

```
<!DOCTYPE html> <html xmlns:th="http://www.thymeleaf.org"  
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"  
layout:decorate="layout"> <head>
```

Em seguida, especificamos o primeiro fragment (pedaço de template HTML):

```
<meta http-equiv="Content-Type"  
content="text/html;charset=UTF-8" /> </head> <body>
```

```
<h1 layout:fragment="header">
```

Green Dog Delivery

```
<small>Painel de controle</small></h1>
```

Depois, fazemos o segundo fragment:

```
<div layout:fragment="content" class="container"> <div class="jumbotron">  
  
<a th:href="@{/ambiente/}"  
class="btn btn-lg btn-info">Ambiente</a>  
  
<a th:href="@{/h2}"  
class="btn btn-lg btn-info">H2 Console</a> </div> </div> </body> </html>
```

Depois de um restart no projeto, teremos o layout aplicado corretamente:



C | ① localhost:8080



Casa do Código

boaglio.com

Spring Boot

Thymeleaf Layout

Green Dog Delivery Painel de controle

Ambiente

H2 Console

Figura 5.7: Fonte do layout correto no browser

5.3 CRUD

As rotinas de CRUD (Create, Read, Update, Delete) ou, em português, criar, ler, atualizar e excluir, existem na maioria dos sistemas, e no sistema do Rodrigo não será diferente: teremos o cadastro de clientes, itens e pedidos. Como o procedimento é bem semelhante, detalharemos apenas o cadastro de clientes.

O CRUD de cliente tem três componentes básicos: a classe de domínio Cliente, a classe controller ClienteController e três arquivos de template do Thymeleaf: form, view e list.

Como roteiro geral, teremos:

Listar clientes: o método ClienteController.list chama o template list.html.

Cadastrar cliente: o método ClienteController.createForm chama o template form.html, que chama o método ClienteController.create.

Exibir cliente: o método ClienteController.view chama o template view.html.

Alterar cliente: o template view.html chama o método ClienteController.alterarForm, que chama o template form.html. Este, por sua vez, chama o método ClienteController.create.

Remover cliente: o template view.html chama o método ClienteController.remover.

Vamos detalhar todos os componentes envolvidos.

Na classe de domínio, temos os campos de ID, nome, endereço e pedidos, com suas validações:

```
@Id
```

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private Long id;
```

```
@NotNull
```

```
@Length(min=2, max=30,
```

```
message="O tamanho do nome deve ser entre {min}
```

```
e {max} caracteres")
```

```
private String nome;
```

```
@NotNull
```

```
@Length(min=2, max=300,
```

```
message="O tamanho do endereço deve ser entre {min}
```

```
e {max} caracteres")
```

```
private String endereco;
```

```
@OneToOne(mappedBy = "cliente", fetch = FetchType.EAGER)  
@Cascade(CascadeType.ALL)  
private List<Pedido> pedidos;
```

A classe ClienteController usa o clienteRepository para suas operações de banco de dados.

```
@Controller @RequestMapping("/clientes") public class ClienteController  
{
```

```
    private final ClienteRepository clienteRepository;
```

A tela inicial retorna uma lista dos clientes cadastrados:

```
@GetMapping("/") public ModelAndView list()  
{  
    Iterable<Cliente> clientes =  
    this  
.clienteRepository.findAll();  
    return new ModelAndView("clientes/list","clientes",clientes)
```

```
;
```

```
}
```

A lista é renderizada pelo template list.html, iniciada com o cabeçalho:

```
<!DOCTYPE html> <html xmlns:th="http://www.thymeleaf.org"
```

```
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
```

```
layout:decorate="layout"> <head> <title>Clientes</title> </head> <body>
```

```
<h1 layout:fragment="header">Lista de clientes</h1>
```

Em seguida, fazemos um link para um novo cadastro e um th:if para alguma mensagem do controller:

```
<div layout:fragment="content" class="container"> <a href="form.html"  
th:href="@{/clientes/novo}">Novo cliente</a> <div class="alert alert-success"  
th:if="${globalMessage}">
```

```
th:text="${globalMessage}">mensagem</div>
```

Depois criamos a tabela HTML para exibir a lista de clientes, iniciando com o cabeçalho e o tratamento de th:if para uma lista vazia:

```
<table class="table table-bordered table-striped"> <thead>
```

```
<tr>
```

```
<td>ID</td>
```

```
<td>Nome</td>
```

```
</tr> </thead> <tbody> <tr th:if="${clientes.empty}">
```

```
<td colspan="3">Sem clientes</td> </tr>
```

Com o atributo th:each, listamos o conteúdo de um array de lista de clientes exibindo os valores com th:text e gerando um link de cada um deles para alteração com th:each.

```
<tr th:each="cliente : ${clientes}">
```

```
<td th:text="${cliente.id}">1</td>

<td><a href="view.html" th:href="@{'/clientes/'+${cliente.id}}">
th:text="${cliente.nome}"> nome </a></td> </tr> </tbody></table></div>
</body> </html>
```

A tela de detalhe de cada cliente usa este método:

```
@GetMapping("{id}") public ModelAndView view(@PathVariable("id")
Cliente cliente)
{
    return new ModelAndView("clientes/view","cliente",cliente);
}
```

O detalhe de cliente é renderizado pelo template view.html, iniciada com o cabeçalho:

```
<html xmlns:th="http://www.thymeleaf.org"
```

```
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
```

```
layout:decorate="layout"> <head>
```

```
<title>Cliente</title> </head>
```

Em seguida, adicionamos uma condição th:if para alguma mensagem do controller:

```
<body>
```

```
<h1 layout:fragment="header">Cliente</h1>
```

```
<div layout:fragment="content" class="well">
```

```
<div class="alert alert-success" th:if="${globalMessage}"
```

```
th:text="${globalMessage}">cliente gravado com sucesso</div>
```

Depois, adicionamos a tabela HTML para exibir o detalhe do cliente, iniciando com o cabeçalho:

```
<table class="table table-striped"> <thead><tr>

<th>ID</th>

<th>Nome</th>

<th>Endereço</th> </tr></thead>
```

Com o tx:text, exibimos o conteúdo do objeto cliente:

```
<tbody><tr>

<td id="id" th:text="${cliente.id}">123</td>

<td id="nome" th:text="${cliente.nome}">Nome</td>

<td id="endereco" th:text="${cliente.endereco}">Endereço</td> </tr></tbody>
</table>
```

E, finalmente, montamos dois links para alterar e remover os dados desse cliente:

```
<div class="pull-left"> <a href="form.html" th:href= "@{/clientes/alterar/' + ${cliente.id}}" >alterar</a>
```

|

```
<a href="clientes" th:href= "@{/clientes/remover/' + ${cliente.id}}" >remover</a>
```

|

```
<a th:href="@{/clientes/}" href="list.html">voltar</a> </div></div> </body> </html>
```

A tela de cadastro de um novo cliente é chamada pelo método:

```
@GetMapping("/novo") public String createForm(@ModelAttribute Cliente cliente)

{
    return "clientes/form"
;
}
```

O formulário é renderizado pelo template form.html, iniciando com o cabeçalho:

```
<!DOCTYPE html> <html xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
layout:decorate="layout"> <head> <title>Cliente</title> </head> <body>

<h1 layout:fragment="header">Cadastro de cliente</h1>
```

Em seguida, usamos th:action para buscar o formulário, th:object para relacionar o formulário ao objeto cliente, e th:each para listar os erros de formulário (se eles existirem).

```
<div layout:fragment="content" class="input-form">      <div class="well">
```

```
<form id="clienteForm" th:action="@{/clientes/(form)}"  
th:object="${cliente}" action="#" method="post"  
  
class="form-horizontal">  
  
<div th:if="#{#fields.hasErrors('*')}"  
class="alert alert-error">  
  
<p th:each="error : #fields.errors('*')"  
th:text="${error}">  
  
Erro de validação  
</p> </div>
```

Então, usamos o th:field para relacionar os campos de input do formulário às propriedades do objeto cliente definido em th:object:

```
<input type="hidden" th:field="*{id}"  
th:class="${#fields.hasErrors('id')} ? 'field-error'" /> <div class="form-group">  
  
<label class="control-label col-sm-2" for="nome">Nome</label>  
  
<input class="col-sm-10" type="text" th:field="*{nome}"  
th:class="${#fields.hasErrors('nome')} ? 'field-error'" /> </div> <div  
class="form-group">  
  
<label class="control-label col-sm-2" for="text">Endereço</label>  
  
<textarea class="col-sm-10" th:field="*{endereco}"  
th:class="${#fields.hasErrors('endereco')} ? 'field-error'">  
  
</textarea> </div>
```

E finalmente, adicionamos o botão de enviar os dados do formulário e um link para voltar à lista de cliente:

```
<div class="col-sm-offset-2 col-sm-10">  
  
<input type="submit" value="Gravar" /> </div> <br/>  
  
<a th:href="@{/clientes/}" href="clientes.html"> voltar </a> </form> </div>  
</div> </body> </html>
```

O formulário de cadastro de novo cliente chama este método:

```
@PostMapping(params = "form") public ModelAndView create  
(@Valid Cliente cliente,  
BindingResult result,RedirectAttributes redirect)  
{  
    if (result.hasErrors()) { return new ModelAndView  
        ("clientes/form","formErrors",result.getAllErrors())  
    ; }  
    cliente =  
    this
```

```
.clienteRepository.save(cliente);

redirect.addFlashAttribute(
"globalMessage"
,

"Cliente gravado com sucesso"

);

return new ModelAndView("redirect:/clientes/{cliente.id}"
,
"cliente.id",cliente.getId())
;
}
```

A tela de alteração de cliente é chamada pelo método:

```
@GetMapping(value = "alterar/{id}") public ModelAndView
alterarForm(@PathVariable("id")
Cliente cliente)
```

```
{  
    return new ModelAndView("clientes/form","cliente",cliente)  
;  
}  
}
```

E finalmente, a opção de excluir cliente chama o método:

```
@GetMapping(value = "remover/{id}") public ModelAndView  
remover(@PathVariable("id")
```

```
Long id,  
RedirectAttributes redirect)
```

```
{  
    this  
.clienteRepository.delete(id);
```

```
Iterable<Cliente> clientes =
```

```
this  
.clienteRepository.findAll();
```

```
ModelAndView mv =
```

```
new  
ModelAndView  
(
```

```
"clientes/list","clientes"  
,clientes);  
  
mv.addObject(  
"globalMessage","Cliente removido com sucesso"  
);  
  
return  
  
mv;  
}  
  
}
```

Os cadastros de itens e pedidos funcionam de forma semelhante. No final, o seu projeto deve ter esses arquivos.

Arquitetura atual

Podemos dividir o nosso cenário atual nas seguintes camadas:

Usuários acessando o sistema.

Sistema web de pedidos com Spring Boot.

Banco de dados MySQL.

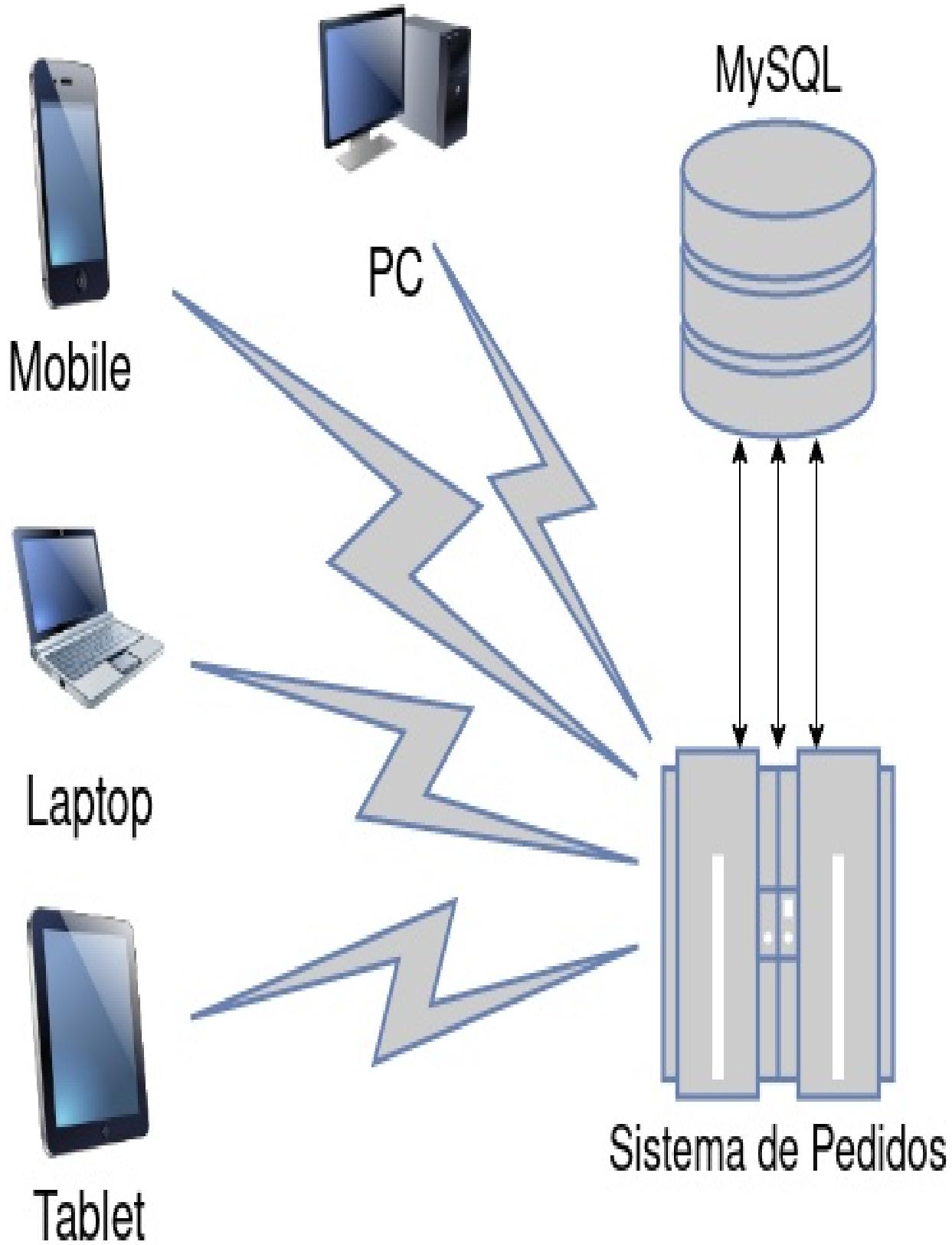


Figura 5.8: Arquitetura atual

5.4 O que aprendemos

Para acompanhar o projeto completo até aqui, acesse o branch crud, em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/crud>.

Certifique-se de que você aprendeu:

- o conceito de template natural;
- os principais atributos do Thymeleaf;
- como o Thymeleaf integra-se aos formulários.

No próximo capítulo, veremos algumas dicas de como aumentar a produtividade do desenvolvimento com Spring Boot.

Capítulo 6

Desenvolvimento produtivo

Quando chega a hora da entrega do projeto, Rodrigo sabe melhor do que ninguém que vale a pena conhecer bem as suas ferramentas para ter a maior produtividade possível. O Spring Boot tem algumas opções interessantes que aceleram bastante o desenvolvimento.

Podemos usar o devtools para acelerar o desenvolvimento, pois ele faz o reload automático da aplicação a cada mudança, entre outras coisas. O LiveReload também é interessante, já que faz o recarregamento do web browser automaticamente, sem a necessidade de pressionar F5. Também podemos usar o Docker para ajudar no deploy do desenvolvimento.

Vamos detalhar essas opções a seguir.

6.1 Devtools

O devtools é um módulo nativo do Spring Boot que oferece algumas vantagens interessantes:

Restart automático — Ao alterar uma classe, o Spring Boot faz o restart do seu contexto rapidamente.

Debug remoto — Permite fazer um debug remoto em uma porta específica configurada nas propriedades.

LiveReload — Ativa a opção de LiveReload, que faz o browser carregar automaticamente a página toda vez que o seu código-fonte é alterado.

Para ativá-lo em um projeto Spring Boot, o primeiro passo é adicioná-lo como dependência:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-devtools</artifactId>
```

```
<optional>true</optional> </dependency>
```

O Boot Dashboard já destaca os projetos com devtools ativo.

The screenshot shows a window titled "Boot Dashboard" with a green circular icon containing a white gear symbol. The window has a light gray header bar with three small red rectangular icons and a green circular arrow icon on the right side. Below the header is a search bar containing the placeholder text "Type tags, projects, or working set names to match (incl. sub-projects)". Underneath the search bar is a list of items. The first item, "local", is preceded by a green circular icon with a white gear symbol and a small downward arrow. The second item, "springboot-greendogdelivery [devtools]", is preceded by a green circular icon with a white gear symbol and a small circle. This item is highlighted with a thick red border. The third item, "springbootproperties", is preceded by a green circular icon with a white gear symbol and a small circle.

- local
- springboot-greendogdelivery [devtools]
- springbootproperties

Figura 6.1: Devtools no Boot Dashboard

Em seguida, subindo o projeto em modo debug, o devtools é ativado automaticamente. O projeto será reiniciado rapidamente a qualquer alteração de classe.

Para forçar o restart, podemos usar o botão do Console do Eclipse:

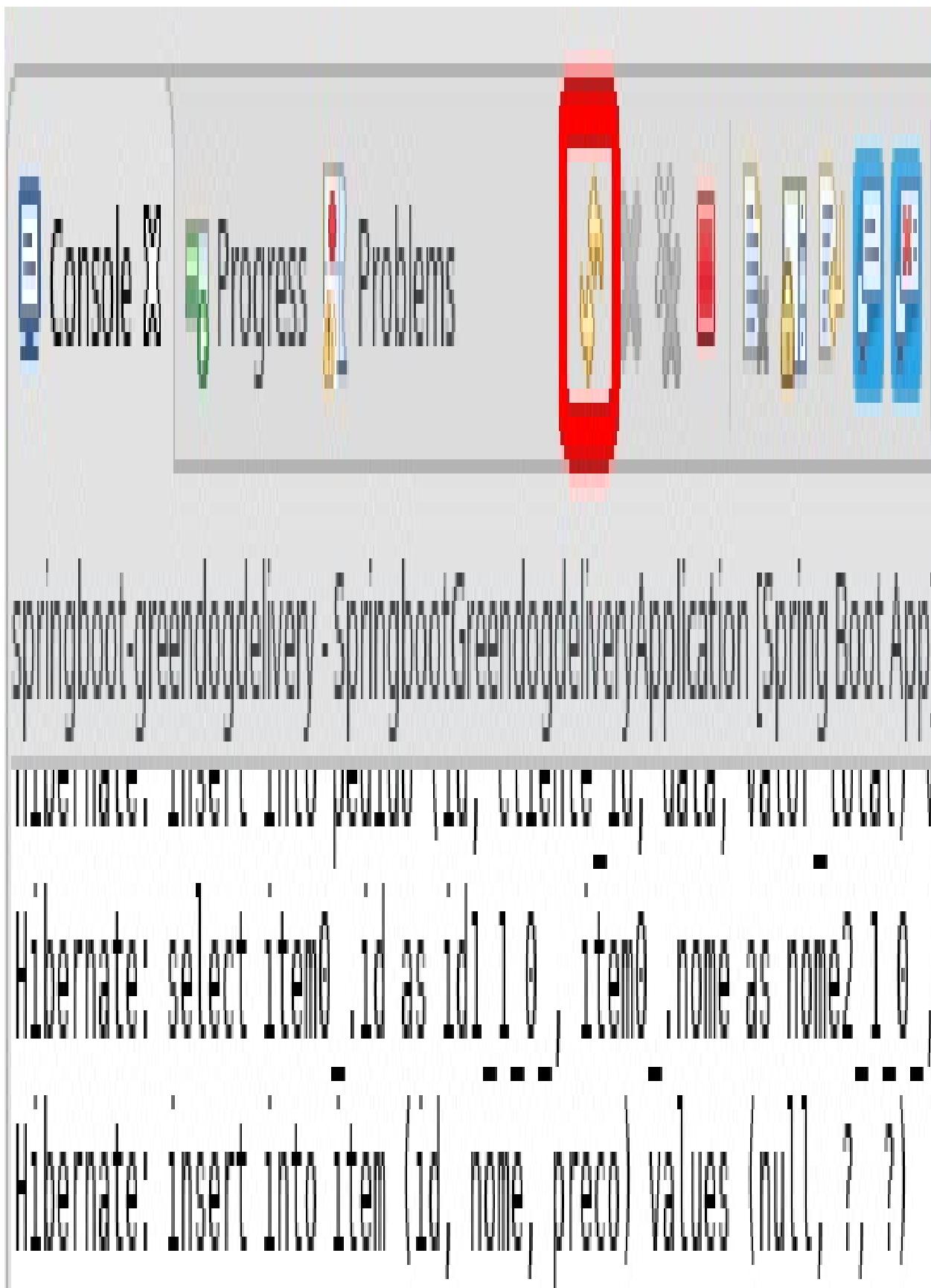


Figura 6.2: Restart forçado do devtools

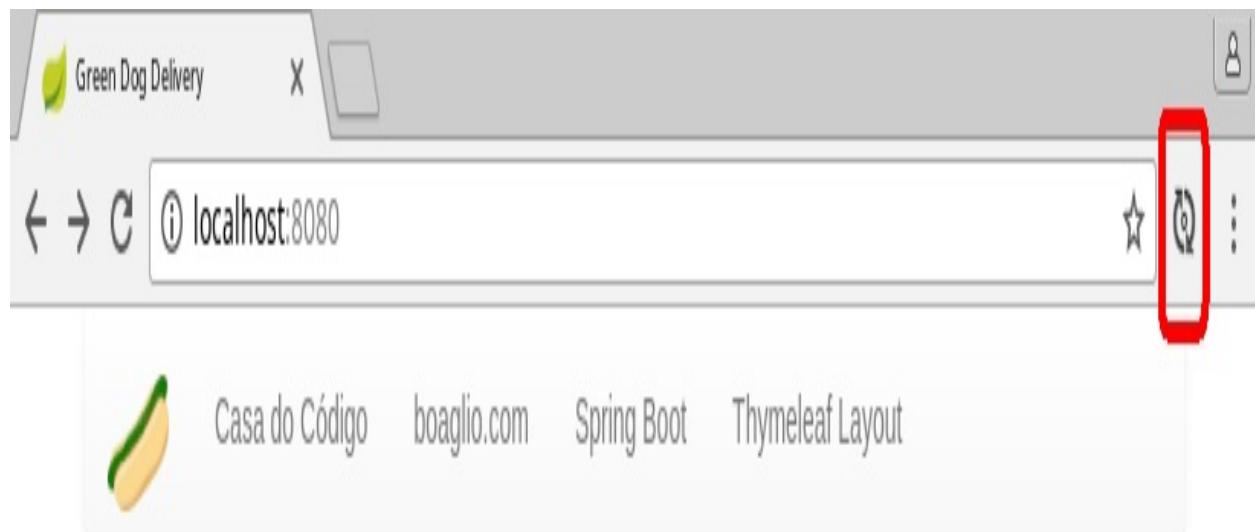
Com esse módulo ativo, não precisamos nos preocupar em parar e subir a aplicação, o restart será automático. A produtividade aumenta muito, o que torna esse item quase que obrigatório no desenvolvimento com Spring Boot.

6.2 LiveReload

O LiveReload é um serviço instalado no seu web browser que permite um refresh automático de página cada vez que o código-fonte é alterado. Isso é muito útil no desenvolvimento de telas web.

A sua instalação é muito simples e pode ser feita acessando <http://livereload.com/extensions/>.

Em seguida, ao acessarmos a aplicação, percebemos no canto superior direito o logotipo do LiveReload, que são duas setas indicando o reload e um círculo ao meio. Se o círculo estiver branco no meio, o LiveReload está desligado; se estiver preto, ligado.



Green Dog Delivery Painel de controle

The dashboard interface features a central light gray rounded rectangle containing five blue rectangular buttons arranged in two rows. The top row contains three buttons: "Ambiente" (light blue), "H2 Console" (light blue), and "Cadastro de clientes" (dark blue). The bottom row contains two buttons: "Cadastro de itens" (dark blue) and "Cadastro de pedidos" (dark blue). All buttons have white text.

Figura 6.3: LiveReload desligado

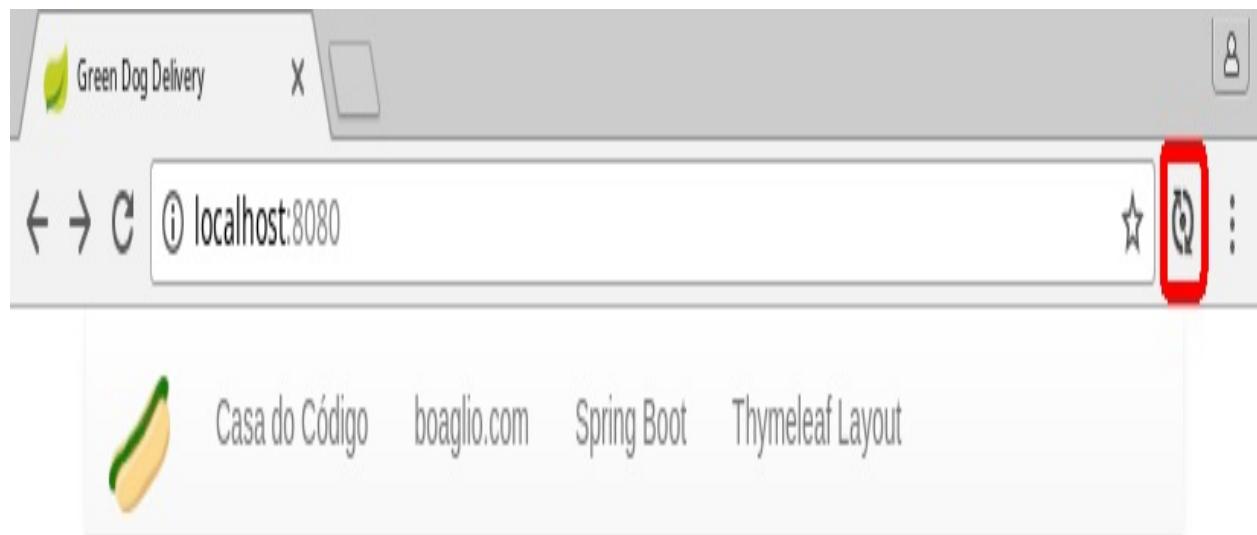
Para testarmos, inicialmente clicamos no ícone para ativar o LiveReload. Depois, editamos o template index.html adicionando o link:

```
<a href="http://livereload.com" class="btn btn-lg btn-info">
```

LiveReload

```
</a>
```

Logo após gravarmos o arquivo HTML, sem fazer nada, o browser atualiza automaticamente a página via LiveReload, mostrando a alteração feita:



Green Dog Delivery Painel de controle

A screenshot of the application's control panel. It features a grid of management links arranged in two rows. The top row contains four links: "Live Reload" (highlighted with a red box), "Ambiente", "H2 Console", and "Cadastro de clientes". The bottom row contains two links: "Cadastro de itens" and "Cadastro de pedidos". All links are in white text on blue buttons.

Figura 6.4: LiveReload ativo

Com esse plugin ativo, economizamos algum tempo ao evitarmos fazer reload manual das páginas web no web browser.

6.3 Docker

Docker é uma virtualização de sistema operacional que é cada vez mais comum no desenvolvimento, pois possibilita de maneira muito fácil publicar uma aplicação em novos ambientes.

A Spotify, empresa que oferece um serviço de música comercial em streaming, criou um plugin que gera uma imagem do Docker de uma aplicação (mais detalhes em <https://spotify.github.io>).

Com esse plugin, poderemos criar uma imagem Docker de nossa aplicação e publicar em qualquer outro ambiente Docker facilmente. Para usarmos esse plugin, precisamos adicionar dentro da tab build:

```
<plugin>
```

```
<groupId>com.spotify</groupId>
```

```
<artifactId>dockerfile-maven-plugin</artifactId>
```

```
<version>1.4.13</version>
```

<configuration>

<repository>boaglio/green-dog-delivery</repository>

<tag>2.4</tag>

<baseImage>openjdk/11</baseImage>

<entryPoint>["java", "-jar", "/\${project.build.finalName}.jar"]</entryPoint>

<exposes>8080</exposes>

<resources>

```
<resource>
```

```
<targetPath>/</targetPath>
```

```
<directory>${project.build.directory}</directory>
```

```
<include>${project.build.finalName}.jar</include>
```

```
</resource>
```

```
</resources>
```

```
</configuration> </plugin>
```

E depois, para criarmos a imagem, usamos o comando do Maven:

```
$ mvn -DskipTests clean package dockerfile:build
```

No console, verificamos que a imagem foi criada com sucesso:

```
[INFO] --- dockerfile-maven-plugin:1.4.13
```

```
:build (default-cli) @ green-dog-delivery ---
```

```
[INFO]
```

```
[INFO] Image will be built as boaglio/green-dog-delivery:
```

```
2.4
```

```
[INFO]
```

```
[INFO] Step
```

```
1/4 : FROM openjdk:11
```

```
[INFO]
```

```
[INFO] Pulling from library/openjdk
```

```
[INFO] Status: Image is up to date
```

```
for openjdk:11
```

```
[INFO] ---> ecef10
```

cd

1bef

[INFO] Step

2/4

: ARG JAR_FILE=target/*.jar

[INFO]

[INFO] ---> Using cache

[INFO] --->

4702

eb576347

[INFO] Step

3/4 : COPY \${JAR_FILE}

app.jar

[INFO]

[INFO] ---> ab07b833f1f4

[INFO] Step

4/4 : ENTRYPOINT ["java","-jar","/app.jar"

]

[INFO]

[INFO] ---> Running

in

bb5fa544e8bd

[INFO] Removing intermediate container bb5fa544e8bd

[INFO] --->

cd

12cc6f891c

[INFO] Successfully built

cd

12cc6f891c

[INFO] Successfully tagged boaglio/green-dog-delivery:

2.4

[INFO]

[INFO] Detected build of image with id

cd

12cc6f891c

[INFO] Building jar: green-dog-delivery-

2.4.0

-SNAPSHOT.jar

[INFO] Successfully built boaglio/green-dog-delivery:

2.4

```
[INFO] -----
```

```
[INFO] BUILD SUCCESS
```

```
[INFO] -----
```

Usando a linha de comando, podemos conferir a imagem do docker criada:

```
$ docker images | grep green
```

```
boaglio/green-dog-delivery
```

```
2.4 12 minutes ago 684MB
```

Para subir a aplicação dentro do contêiner, usamos o comando:

```
$ docker run -p 8080:8080 boaglio/green-dog-delivery:2.4
```

Depois de executado, a aplicação pode ser acessada em <http://localhost:8080>.

O Docker possui várias ferramentas para gerenciar os seus contêineres. Vamos usar aqui o Portainer (<http://portainer.io/>). Para subir essa interface, usamos o comando:

```
$ docker run -d -p 9000:9000
```

```
-v /var/run/docker.sock:/var/run/docker.sock
```

```
portainer/portainer
```

Acessando a interface em <http://localhost:9000/>, verificamos o contêiner de nossa aplicação rodando.

portainer.io

Container list

Containers

Home LOCAL

Dashboard

App Templates

Stacks

Containers

Images

Networks

Volumes

Events

Host

Containers

Start Stop Kill Restart Pause Resume Remove Add container

Search...

Name	State	Quick actions	Stack	Image
epic_nightingale	running	⋮ ⚡ ⏪ ⏴ ⏵ ⏶	-	portainer/portainer
loving_grothendieck	running	⋮ ⚡ ⏪ ⏴ ⏵ ⏶	-	boaglio/green-dog-delivery:2.4

Figura 6.5: Portainer

Com a imagem pronta, conseguimos testar localmente e, se necessário, publicar a imagem Docker em outros ambientes.

6.4 O que aprendemos

Certifique-se de que você aprendeu:

- a ativar e usar o devtools em projetos Spring Boot;
- a configurar e usar o LiveReload;
- a gerar uma imagem Docker da aplicação.

No próximo capítulo, vamos ver algumas customizações de nossa aplicação web dentro do Spring Boot.

Capítulo 7

Customizando nossa aplicação web

Como toda aplicação web, Rodrigo precisa customizar algumas coisas. As páginas de erro do Spring Boot, por exemplo, não são muito amigáveis para os usuários, e é importante também obter algumas informações do seu ambiente rodando no servidor.

Vamos ver algumas opções que o Spring Boot oferece.

7.1 Banner

Com o aumento de quantidade de sistemas, uma opção de layout na saída do console pode ajudar a identificá-la no meio de tanto texto do Spring Boot.

Existe uma opção que, ao adicionar uma imagem banner.png ao diretório src/main/resources, o Spring Boot automaticamente a transforma em ASCII colorido:

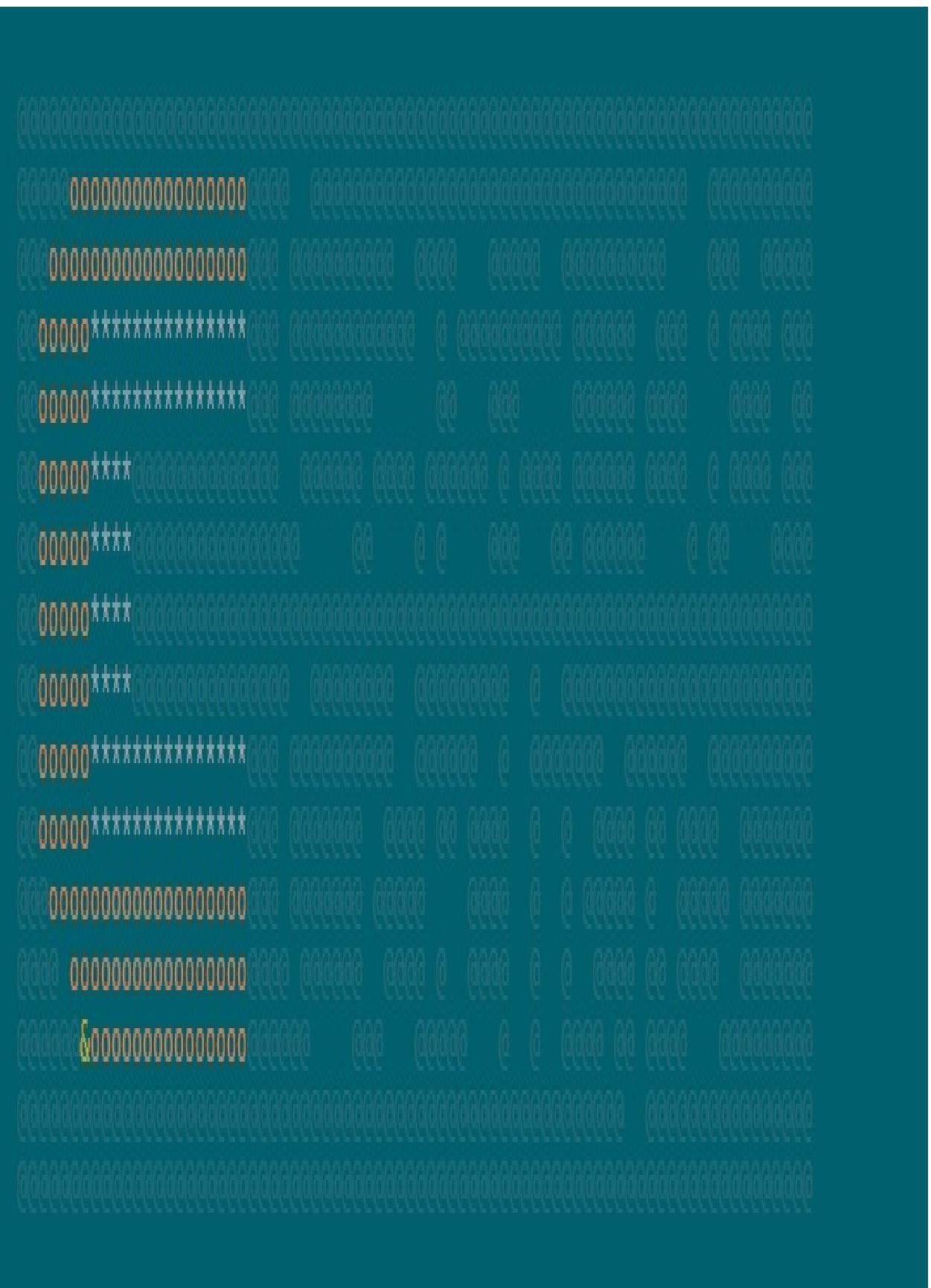


Figura 7.1: Banner comum

Existem outras opções que podem ser usadas para customizar esse banner. Por exemplo, ao adicionarmos `spring.banner.image.width=150` ao arquivo de `application.properties`, temos o banner aumentado:

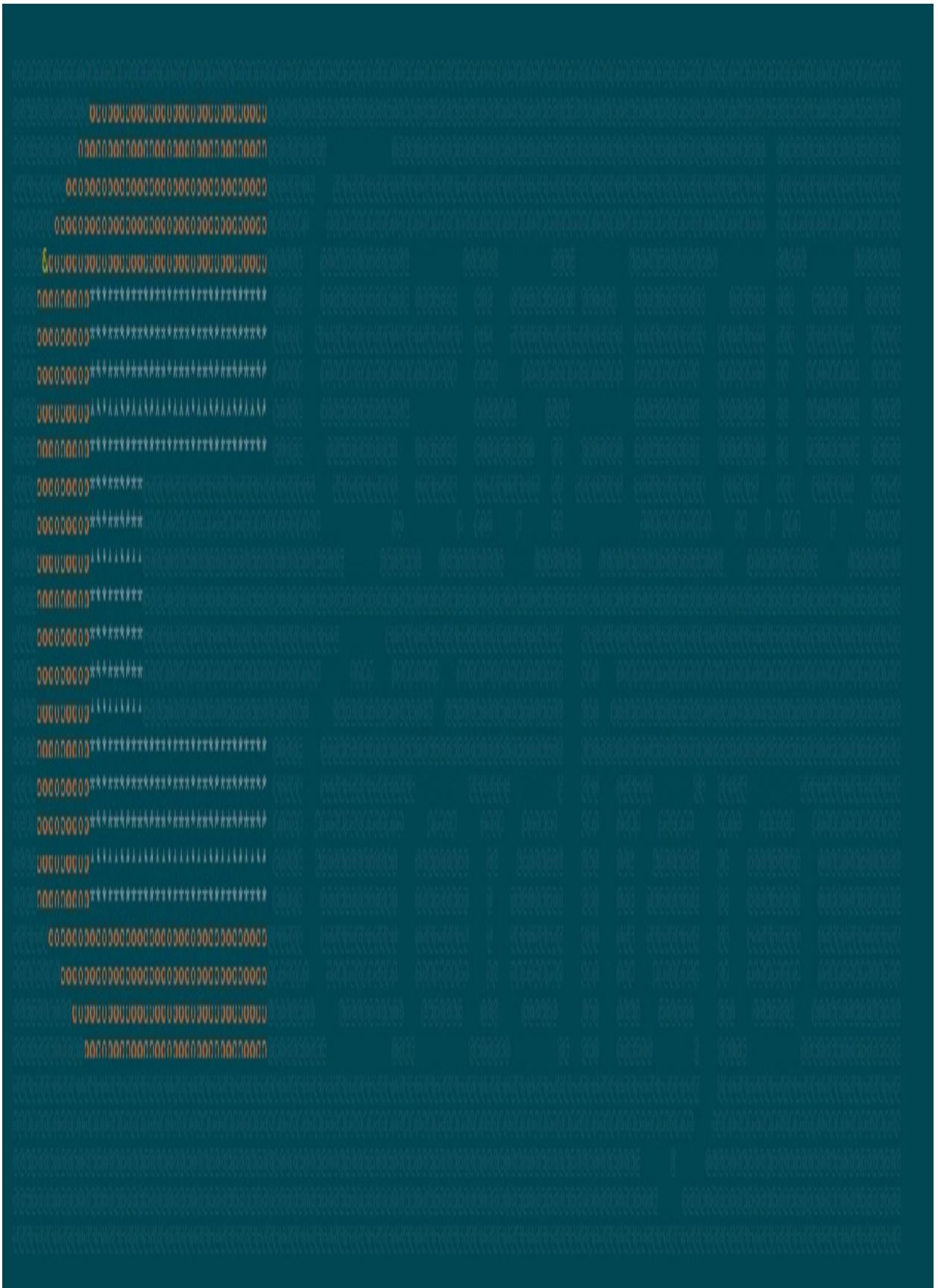
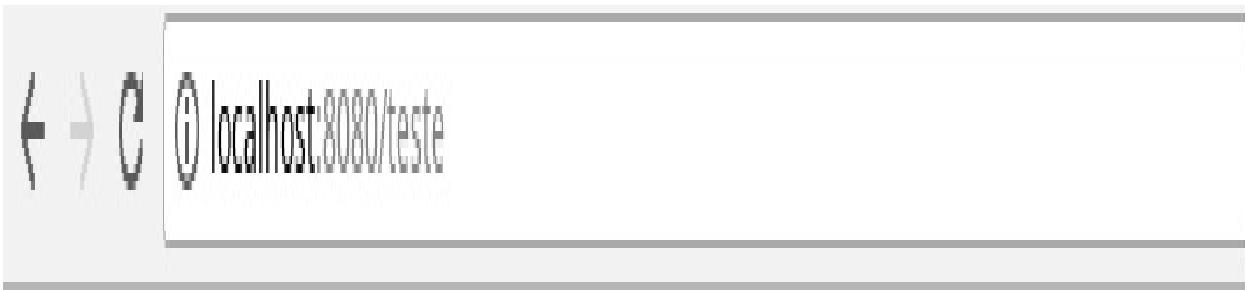


Figura 7.2: Banner aumentado

Apesar de o banner ser apenas um recurso visual para o console, ele é bem customizável e pode ser útil para identificar nele o seu projeto.

7.2 Páginas de erro

O erro 404 (de página não encontrada) passa uma mensagem um pouco confusa ao usuário: Whitelabel Error Page. O termo que significa etiqueta em branco (Whitelabel) é uma referência às gravadoras que produziam discos de vinil com uma etiqueta em branco para uma eventual promoção ou teste.



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu Dec 31 01:46:03 BRST 2020

There was an unexpected error (type=Not Found, status=404).

Figura 7.3: Página não encontrada padrão

Para trocarmos a página de erro, basta criarmos um arquivo 404.html dentro de src/main/resources/public/error, com o seguinte conteúdo:

```
<!DOCTYPE html> <html xmlns="http://www.w3.org/1999/xhtml"> <head>

<meta charset="utf-8">

<title>404 - essa página não existe</title>

<style> .. muito CSS... </style> </head> <body> <div id="content">

<div class="clearfix"></div>

<div id="main-body">

<p class="enormous-font bree-font">404</p>
```

```
<p class="big-font">Página não encontrada...</p> <hr> <p class="big-font">
```

Voltar para

```
<a href="/" class="underline">home page</a> </p>
```

```
</div> <div> </body> </html>
```

Sem alterar mais nada, ao reiniciar a aplicação, a nova página de erro aparece:

← → C

localhost:8080/teste

404

Página não encontrada...

Voltar para [home page](#)

Figura 7.4: Página não encontrada customizada

Da mesma maneira e com o mesmo código, a página de erro 500 (500.html) pode exibir uma mensagem amigável de sistema indisponível. É só criar um arquivo com este nome e colocar no mesmo lugar (src/main/resources/public/error).

7.3 Actuator

A tradução mais comum para a palavra em inglês Spring é primavera, mas outra tradução válida é mola. O Spring Actuator pode ser traduzido por Atuador de Mola, que é uma máquina elétrica ou hidráulica, usada em indústrias, que produz movimento. No nosso contexto, o Actuator é um subprojeto do Spring Boot que ajuda a monitorar e a gerenciar a sua aplicação quando ela for publicada (estiver em execução).

O Actuator é muito útil para ver o ambiente em produção, pois permite que sejam monitoradas informações do servidor rodando. Sua ativação é feita adicionando a dependência:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-actuator</artifactId> </dependency>
```

Apenas adicionando essa dependência, conseguimos acessar alguma informação em /actuator.

HTTPie com /actuator

Em vez de acessar com o web browser, vamos mostrar o resultado com a ferramenta cliente de linha de comando HTTPie (<https://httpie.io>), muito usada para ver resultados de chamadas ReST.

```
$ http localhost:8080
```

```
/actuator
```

```
HTTP/
```

```
1.1 200
```

```
Connection: keep-alive
```

```
Content-Type: application/vnd.spring-boot.actuator.v3+json
```

```
{
```

```
"_links"
```

```
: {
```

```
"health"
```

```
: {
```

```
"href": "http://localhost:8080/actuator/health"
```

,

"templated": false

},

"health-path"

: {

"href": "http://localhost:8080/actuator/health/{*path}"

,

"templated": true

},

"info"

: {

"href": "http://localhost:8080/actuator/info"

,

```
"templated": false
```

```
},
```

```
"self"
```

```
: {
```

```
"href": "http://localhost:8080/actuator"
```

```
,
```

```
"templated": false
```

```
}
```

```
}
```

endpoints health e info

Vemos apenas dois endpoints ativos: health (saúde do sistema) e info (informações do sistema), que podem ser acessados como este exemplo:

```
$ http localhost:8080
/actuator/health

HTTP/
1.1 200

Connection: keep-alive
Content-Type: application/vnd.spring-boot.actuator.v3+json

{

"status": "UP"

}
```

O endpoint info por padrão vem vazio:

```
$ http localhost:8080
/actuator/info

HTTP/
1.1 200

Connection: keep-alive
Content-Type: application/vnd.spring-boot.actuator.v3+json

{}
```

Mais exemplos de endpoints do Actuator

Existem propriedades para ativar cada endpoint restante por vez, mas podemos usar a opção a seguir para ativá-los de uma só vez:

```
management.endpoints.web.exposure.include=*
```

Existem várias opções disponíveis dentro do Actuator através de serviços ReST, vamos destacar algumas:

actuator - Lista todos os links disponíveis.

threaddump - Faz um thread dump.

env - Mostra propriedades do ConfigurableEnvironment do Spring.

health - Mostra informações do status da aplicação.

metrics - Mostra métricas da aplicação.

mappings - Exibe os caminhos dos @RequestMapping.

httptrace - Exibe o trace dos últimos 100 requests HTTP.

A lista completa está em <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints>.

Vamos alterar a nossa página de ambiente.html e adicionar a chamada desses serviços, adicionando os links da lista anterior:

```
<div layout:fragment="content" class="container"> <div><table><thead> <tr>
<th>link</th><th>Descrição</th></tr></thead><tbody> <tr><td><p><a
href="#" th:href="@{/actuator}">actuator</a></p> </td><td><p>Lista todos os
links disponíveis.</p></td></tr> <tr><td><p><a href="#">
th:href="@{/actuator/threaddump}">threaddump</a></p> </td><td><p>Faz um
thread dump.</p></td></tr> <tr><td><p><a href="#">
th:href="@{/actuator/env}">env</a></p> </td><td><p>Mostra properties do
CE do Spring.</p></td></tr> <tr><td><p><a href="#">
th:href="@{/actuator/health}">health</a></p> </td><td><p>Mostra
informações do status da aplicação.</p> </td></tr> <tr><td><p><a href="#">
th:href="@{/actuator/metrics}">metrics</a></p> </td><td><p>Mostra métricas
da aplicação.</p></td></tr> <tr><td><p><a href="#">
th:href="@{/actuator/mappings}">mappings</a></p> </td><td><p>Exibe os
caminhos dos @RequestMapping.</p> </td></tr> <tr><td><p><a href="#">
th:href="@{/actuator/httptrace}">httptrace</a></p> </td><td><p>Exibe o trace
dos últimos 100 requests HTTP.</p> </td></tr> </tbody></table><br/> <a
href="#" th:href="@{/properties}" class="btn btn-large btn-success">System
Properties</a> </div></div>
```

Apenas alterando uma página de template, temos várias funcionalidades interessantes rodando.



Ambiente - Spring Actuator

link descrição

[actuator](#) Lista todos os links disponíveis.

[threaddump](#) Faz um thread dump.

[env](#) Mostra properties do ConfigurableEnvironment do Spring.

[health](#) Mostra informações do status da aplicação.

[metrics](#) Mostra métricas da aplicação.

[mappings](#) Exibe os caminhos dos @RequestMapping.

[httptrace](#) Exibe o trace dos últimos 100 requests HTTP.

[System Properties](#)

[Lista completa de endpoints](#)

Figura 7.5: Links do Actuator

Ajustando o health

A opção `health` mostra poucas informações da saúde do sistema por padrão. Para aumentar o escopo, precisamos adicionar esta propriedade:

```
management.endpoint.health.show-details=always
```

Repetindo a chamada, verificamos informações adicionais do banco de dados, rede e espaço em disco:

```
$ http localhost:8080
```

```
/actuator/health
```

```
HTTP/
```

```
1.1 200
```

```
Connection: keep-alive
```

```
Content-Type: application/vnd.spring-boot.actuator.v3+json
```

```
{
```

"components"

: {

"db"

: {

"details"

: {

"database": "H2"

,

"validationQuery": "isValid()"

},

"status": "UP"

},

"diskSpace"

: {

"details"

: {

"exists": true

,

"free": 99381198848

,

"threshold": 10485760

,

"total": 426206126080

},

"status": "UP"

```
    },  
  
    "ping":  
    : {  
  
        "status": "UP"  
  
    },  
  
    "status": "UP"  
  
},
```

Entendendo o metrics

A opção metrics lista algumas métricas do sistema, como a memória em uso, quantidade de classes, threads, processos, operações do Garbage Collector (GC), entre outras coisas.

Para obtermos as métricas, basta usarmos qualquer nome retornado do endpoint /actuator/metrics, como o http.server.requests para informações de HTTP ou

jvm.classes.loaded para informações da quantidade de classes que a JVM carrega (na ferramenta HTTPie, podemos omitir o localhost):

```
$ http :8080
```

```
/actuator/metrics/jvm.classes.loaded
```

```
HTTP/
```

```
1.1 200
```

```
Connection: keep-alive
```

```
Content-Type: application/vnd.spring-boot.actuator.v3+json
```

```
{
```

```
"availableTags"
```

```
: [],
```

```
"baseUnit": "classes"
```

```
,
```

```
"description":
```

```
"The number of classes that are
```

```
currently loaded in the Java virtual machine"
```

```
,
```

```
"measurements"
: [
{
  "statistic": "VALUE"
,
  "value": 16535.0
}

],
}

"name": "jvm.classes.loaded"
}
```

Configurando o httptrace

Dos endpoints da página que criamos, todos funcionam, com exceção do /actuator/httptrace. Isso acontece porque esse recurso ocupa um pouco de

processamento e memória que pode nem ser usado, então ele vem desativado por padrão e precisa ser explicitamente ativado e configurado.

Para ativar esse poderoso recurso do Spring Boot, primeiro declaramos o componente usando a implementação do armazenamento em memória:

```
@Configuration public class HttpTraceActuatorConfiguration  
{  
  
    @Bean  
  
    public HttpTraceRepository httpTraceRepository()  
    {  
  
        return new InMemoryHttpTraceRepository()  
    ;  
    }  
}
```

E depois declaramos a propriedade para ativar o httptrace:

```
management.trace.http.enabled=true
```

Depois disso, toda chamada HTTP será armazenada e podemos consultar o resultado das últimas requisições no endpoint do httptrace. Para fazermos um teste simples, vamos acessar um contexto qualquer:

```
$ http :8080
```

```
/teste123
```

```
HTTP/
```

```
1.1 404
```

```
...
```

E depois consultar essa chamada no httptrace:

```
$ http :8080
```

```
/actuator/httptrace
```

```
HTTP/
```

```
1.1 200
```

```
...
```

```
{
```

"traces"

: [

{

"request"

: {

"headers"

: {

"accept"

: [

"*/*"

],

"accept-encoding"

: [

"gzip, deflate"

],

"connection"

: [

"keep-alive"

],

"host"

: [

"localhost:8080"

],

"user-agent"

: [

```
"HTTPie/2.3.0"
```

```
]
```

```
},
```

```
"method": "GET"
```

```
,
```

```
"remoteAddress"
```

```
: null,
```

```
"uri": "http://localhost:8080/teste123"
```

```
},
```

```
...
```

Customizando o info

O endpoint info permite exibir qualquer tipo de informação do seu sistema. Em algumas delas, ao colocar as propriedades com prefixo info, o Spring Boot automaticamente converte em JSON, inclusive variáveis de ambiente do Maven/Gradle. Outras variáveis que podemos usar estão nesse link da

documentação do Maven:

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Available_Variables.

Exemplo de propriedades:

info.app.name=greendogdelivery

info.app.description=Greendog Delivery

info.app.version=@pom.version@

Resultado:

\$ http :8080

/actuator/info

HTTP/

1.1 200

{

"app"

: {

```
"description": "Greendog Delivery "
```

```
,
```

```
"name": "greendogdelivery"
```

```
,
```

```
"version": "2.4-SNAPSHOT"
```

```
}
```

```
}
```

Adicionando um valor qualquer:

```
info.fernando=boaglio
```

Ele é automaticamente adicionado ao resultado:

```
$ http :8080
```

```
/actuator/info
```

```
HTTP/
```

1.1 200

{

"app"

: {

"description": "Greendog Delivery "

,

"name": "greendogdelivery"

,

"version": "2.4-SNAPSHOT"

},

"fernando": "boaglio"

}

7.4 O que aprendemos

Os fontes do projeto estão em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/customizando>.

Certifique-se de que você aprendeu:

- a customizar um banner na saída de console do Spring Boot;
- a criar páginas personalizadas de erros HTTP;
- a ativar e usar as páginas do Spring Actuator.

No próximo capítulo, veremos a facilidade existente no Spring Boot de expor seus serviços ReST.

Capítulo 8

Expondo a API do seu serviço

O sistema do Rodrigo já tem cadastros de clientes, itens e pedidos, que podem ser usados pelo administrador. O que resta agora é criar a tela de novos pedidos, que será utilizada pelos clientes do Green Dog Delivery.

Rodrigo não conhece muito os frameworks JavaScript mais novos, mas sabe que a primeira versão do AngularJS é bem rápida e muito usada pela comunidade. Para uma boa performance, ele escolheu esse framework para enviar os pedidos. Precisamos agora criar os serviços ReST para este fim.

8.1 HATEOAS

Já percebemos que, com o Spring Data, não precisamos mais escrever repositórios. Agora mostraremos que existe uma facilidade maior para não escrever serviços ReST também.

O termo HATEOAS (Hypermedia as the Engine of Application State) significa hipermídia como a engine do estado da aplicação e serve como um agente que mapeia e limita a arquitetura ReST. De acordo com o modelo de maturidade ReST de Richardson, HATEOAS é o nível máximo que introduz a descoberta de serviços, fornecendo uma maneira de fazer um protocolo autodocumentado.

Resumindo, HATEOAS é uma arquitetura mais completa do que ReST.

No nosso sistema, colocando apenas uma anotação, temos o serviço no formato HATEOAS pronto para uso. Adicionando um starter, temos uma tela de consulta aos serviços on-line.

Veremos um exemplo do HATEOAS começando com uma alteração no nosso repositório ItemRepository. Adicionaremos a anotação RepositoryRestResource, que define o acesso aos serviços na URI /itens.

`@RepositoryRestResource`

```
(collectionResourceRel=
"itens",path="itens") public interface ItemRepository
extends JpaRepository<Item, Long> {
```

Em seguida, vamos implementar o navegador de serviços adicionando uma dependência ao pom.xml:

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-rest-hal-browser</artifactId>
<version>3.3.6.RELEASE</version> </dependency> <dependency>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-rest</artifactId> </dependency>
```

O comportamento padrão da documentação é ficar na raiz do site. Como já temos algumas páginas lá, vamos alterar para /api adicionando uma propriedade no arquivo application.properties:

```
#rest  
spring.data.rest.base-path=/api
```

Depois, colocamos um link na página inicial (index.html):

```
<a th:href="@{/api/browser/index.html#/api/}" class="btn btn-lg btn-info">HAL Browser</a>
```

Ao subirmos a aplicação, temos um novo link:



Casa do Código boaglio.com Spring Boot Thymeleaf Layout

Green Dog Delivery Painel de controle

A screenshot of a web application interface. At the top, there is a navigation bar with the text "Casa do Código", "boaglio.com", "Spring Boot", and "Thymeleaf Layout". Below this, the main title "Green Dog Delivery Painel de controle" is displayed. The interface features a horizontal menu bar with four items: "Ambiente", "H2 Console", "HAL Browser", and "Cadastro de clientes". The "HAL Browser" button is highlighted with a red border, indicating it is the active or selected item.

- Ambiente
- H2 Console
- HAL Browser
- Cadastro de clientes

Figura 8.1: Link do HAL Browser

Acessando o HAL Browser, podemos navegar entre os serviços existentes nos links de pedidos, clientes e itens.

Explorer

[Go!](#)

Custom Request Headers

Properties

{
}

Links

rel	title	name / index	docs	GET	NON-GET
clientes					
pedidoes					
itens					
profile					

Inspector

Response Headers

200 success

connection: keep-alive
content-type: application/hal+json
date: Thu, 31 Dec 2020 19:41:39 GMT
keep-alive: timeout=60
transfer-encoding: chunked
vary: Origin, Access-Control-Request-Method
headers

Response Body

```
{  
  "_links": {  
    "clientes": {  
      "href": "http://127.0.0.1:8080/api/  
      "templated": true  
    },  
    "pedidoes": {  
      "href": "http://127.0.0.1:8080/api/  
      "templated": true  
    },  
    "itens": {  
      "href": "http://127.0.0.1:8080/api/  
    }  
  }  
}
```

Figura 8.2: Serviços do HAL Browser

No HATEOAS, cada serviço sempre retorna mais informações além das suas próprias, como os endereços para o próprio serviço (`self`), entre outras.

Embedded Resources

items[0]

Properties

```
{  
  "nome": "Green Dog tradicional picante",  
  "preco": 27  
}
```

Links

ref	title	name / index	docs	GET	NON-GET
self					
item					

```
"preco": 30,  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/api/items/2"  
    },  
    "item": {  
      "href": "http://localhost:8080/api/items/2"  
    }  
  },  
},
```

```
{  
  "nome": "Green Dog tradicional",  
  "preco": 25,  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/api/items/3"  
    },  
    "item": {  
      "href": "http://localhost:8080/api/items/3"  
    }  
  },  
},
```

Figura 8.3: Serviço de itens no HAL Browser

O serviço HATEOAS não retorna o ID nos serviços, o que normalmente é um problema para sistemas feitos com AngularJS ou similares. Além disso, nos serviços, ele não retorna o MIME type application/json, e sim application/hal+json.

Para resolver esse problema, precisamos adicionar um parâmetro específico do HATEOAS com valor false no arquivo application.properties.

```
# Hypermedia As The Engine Of Application State  
spring.hateoas.use-hal-as-default-json-media-type=false
```

Também precisamos criar uma classe para informar quais repositórios precisam expor o valor do campo ID nos serviços.

```
@Component public class SpringDataRestCustomization  
implements RepositoryRestConfigurer  
{  
  
    @Override
```

```
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config){  
    config.exposeIdsFor(Item.class,ClienteRepository.class);  
}  
}
```

8.2 Angular acessando ReST

Rodrigo ficou empolgado ao descobrir que também não precisaria escrever os serviços ReST, mas somente consumi-los de um novo sistema que faz apenas pedidos, usando AngularJS. Então, o que precisa ser feito é criar um controller novo para receber os novos pedidos e uma rotina em JavaScript para chamar esse serviço.

Inicialmente, criaremos uma classe RespostaDTO usando o pattern DTO para receber e enviar os valores de um novo pedido:

```
public class RespostaDTO
```

```
{
```

```
private
```

```
Double valorTotal;
```

```
private
```

```
Long pedido;
```

```
private
```

```
String mensagem;
```

```
public RespostaDTO  
(Long pedido,  
 Double valorTotal, String mensagem)  
{  
  
    this  
    .pedido = pedido;  
  
    this  
    .valorTotal = valorTotal;  
  
    this  
    .mensagem = mensagem;  
}  
  
// getters e setters
```

Em seguida, criaremos o serviço que receberá novos pedidos na classe NovoPedidoController, colocando no construtor as dependências do repositório de clientes e itens:

```
@RestController public class NovoPedidoController
```

```
{
```

```
@Autowired
```

```
public NovoPedidoController
```

```
(ClienteRepository clienteRepository,
```

```
ItemRepository itemRepository )
```

```
{
```

```
this
```

```
.clienteRepository =clienteRepository;
```

```
this
```

```
.itemRepository=itemRepository;
```

```
}
```

```
private final ClienteRepository clienteRepository; private final ItemRepository  
itemRepository;
```

Depois, declaramos o serviço que recebe o ID do cliente e uma lista de IDs de itens de pedidos, separados por vírgulas:

```
@GetMapping("/rest/pedido/novo/{clienteId}/{listaDeItens}") public  
RespostaDTO novo(@PathVariable("clienteId")  
  
Long clienteId,@  
  
PathVariable("listaDeItens")  
  
String listaDeItens)  
  
{
```

```
RespostaDTO dto =  
  
new  
RespostaDTO();  
  
try  
  
{  
  
Cliente c = clienteRepository.findOne(clienteId);  
  
  
String[] listaDeItensID = listaDeItens.split(  
",");
```

Então, instanciamos um novo pedido e atualizamos a informação do cliente.

```
Pedido pedido = new  
Pedido();
```

```
double valorTotal = 0

;

List<Item> itensPedidos =

new

ArrayList<Item>();

for

(String itemId : listaDeItensID) {

    Item item = itemRepository.findOne(Long.parseLong(itemId));

    itensPedidos.add(item);

    valorTotal += item.getPreco();

}

pedido.setItens(itensPedidos);

pedido.setValorTotal(valorTotal);

pedido.setData(

new

Date());

pedido.setCliente(c);

c.getPedidos().add(pedido);

this.clienteRepository.saveAndFlush(c);
```

Então, o resultado do pedido e o valor são retornados no serviço dentro da

variável dto.

```
List<Long> pedidosID = new
```

```
ArrayList<Long>();
```

```
for
```

```
(Pedido p : c.getPedidos()) {
```

```
pedidosID.add(p.getId());
```

```
}
```

```
Long ultimoPedido = Collections.max(pedidosID);
```

```
dto =
```

```
new
```

```
RespostaDTO(ultimoPedido, valorTotal,
```

"Pedido efetuado com sucesso"

```
);
```

```
}
```

```
catch
```

```
(Exception e) {
```

```
dto.setMensagem(
```

"Erro: "

```
+ e.getMessage());
```

```
}
```

```
return
```

```
dto;
```

```
}
```

E para terminar o back-end, adicionamos o método à classe IndexController para chamar a página inicial de novo pedido.

```
@GetMapping("/delivery") public String delivery()
```

```
{
```

```
    return "delivery/index"
```

```
;
```

```
}
```

No front-end, precisamos colocar no index.html um link chamando a página de pedidos:

```
<a th:href="@{/delivery/}" class="btn btn-lg btn-info">Delivery</a>
```

Finalmente, dentro de src/main/resources/templates, vamos criar a pasta delivery

e, dentro dela, o arquivo index.html com o nosso código em AngularJS para chamar o serviço de novo pedido.

Inicialmente, começamos com a rotina que lista os itens com o nome e o preço dentro de um checkbox:

```
... <div ng-controller="pedidoController"> <form class="form-horizontal">
```

```
<fieldset>
```

```
<legend>Delivery - Novo Pedido</legend>
```

```
<div class="form-group"> <label class="col-md-12" for="checkboxes">
```

```
 </div> <div class="form-group">
```

```
<div class="col-md-12"> <div class="checkbox checkbox-primary" ng-repeat="i in itens"> <label for="checkboxes-0" class="opcao"> <input name="checkboxes" class=" " checklist-model="pedidoItens" checklist-value="i" ng-click="isItemSelected(i)" type="checkbox">  
    &nbsp;{{i.nome}} [R$ {{i.preco}}]  
  </label> </div></div></div>
```

Em seguida, a rotina para fazer o pedido, que chama a função fazerPedido:

```
<div class="form-group"> <label class="col-md-12" for="btnSubmit">
```

Subtotal:

```
R${{subTotal}}
```

```
</label> </div> <div class="form-group"> <div class="col-md-12"> <button id="btnSubmit" name="btnSubmit" ng-click="fazerPedido(pedidoItens)"
```

```
class="btn btn-primary">Fazer o pedido</button> </div> </div> </fieldset> </form> <div class="alert alert-success" ng-show="idPedido!=null">
```

```
<strong>Pedido {{idPedido}} {{mensagem}} </div> <div class="alert alert-warning" ng-show="idPedido!=null">
```

Valor do pedido:

```
<strong>{{valorTotal}} reais. </div> <div class="alert alert-warning" ng-show="idPedido!=null">
```

Chamada do serviço:

```
<strong>{{urlPedido}} </strong> </div> <fieldset>{{message}} </fieldset> </div></div>
```

A função fazerPedido está declarada dentro do arquivo delivery.js, o que foi feito, inicialmente, declarando-a no formato do AngularJS e registrando a função carregarItens:

```
var app = angular.module("delivery",["checklist-model"])
```

```
],  
function($locationProvider)  
{  
  $locationProvider.html5Mode({  
    enabled:  
    true  
  
,  
    requireBase:  
    false  
  });  
});
```

```
app.controller(  
  'pedidoController'  
,  
  function($scope,$location,$http)  
{  
  $scope.itens = [];  
  $scope.subTotal =  
  0
```

```

;

$scope.pedidoItens=[];

var carregarItens= function ()
{
  $http.get(
"/api/itens").success(function (data)
{
  $scope.itens = data[
"_embedded"]["itens"
];
}).error(
function (data, status)
{
  $scope.message =
"Aconteceu um problema: "
+ data;
});
};


```

Depois, registramos a função fazerPedido que chama o serviço ReST:

```
$scope.fazerPedido = function(pedidoItens)
{
  $scope.message =
  ""; var pedidoStr=""; var prefixo=""; for (var i=0
  ; i< $scope.pedidoItens.length; i++) {
    pedidoStr+=prefixo+$scope.pedidoItens[i].id;
    prefixo=
    ",";
  }
  $scope.urlPedido=
  "/rest/pedido/novo/2/"
  +pedidoStr;
  $http.get( $scope.urlPedido).success(
  function (data)
  {
    $scope.idPedido= data[
    "pedido"
  ];
    $scope.mensagem= data[
```

```
"mensagem"
];
$scope.valorTotal= data[
"valorTotal"
];
}).error(
function (data, status)
{
$scope.message =
"Aconteceu um problema: "
+
>Status:"+ data.status+ " - error:"
+data.error;
});
};

$scope.isItemSelected =
function()
{
if (this
.checked)
```

```
$scope.subTotal+=
```

```
this
```

```
.i.preco;
```

```
else
```

```
$scope.subTotal-=
```

```
this
```

```
.i.preco;
```

```
}
```

```
carregarItens();
```

```
});
```

Subindo a aplicação no link delivery, temos a lista dos itens carregada e chamando o serviço:



Casa do Código

boaglio.com

Spring Boot

Thymeleaf Layout

Delivery - Novo Pedido



Cardápio

Green Dog tradicional picante [R\$27]

Green Dog max salada [R\$30]

Green Dog tradicional [R\$25]



Subtotal: R\$30

Fazer o pedido

Figura 8.4: Fazendo novo pedido

Ao clicar em fazer o pedido, o serviço ReST é chamado e a resposta é devolvida com o número do pedido.

Delivery - Novo Pedido



- Green Dog tradicional picante [R\$27]
- Green Dog max salada [R\$30]
- Green Dog tradicional [R\$25]



Subtotal: R\$30

Fazer o pedido

Pedido 5 Pedido efetuado com sucesso

Valor do pedido: **30** reais.

Chamada do serviço: /rest/pedido/novo/2/2

Figura 8.5: Novo pedido efetuado

Pronto, o nosso sistema de delivery está pronto para receber pedidos on-line.

Customizando via propriedades

As promoções relâmpago ou eventos com um intervalo de tempo definido, como a tradicional Black Friday, exigem uma quantidade considerável de alterações no sistema, e um atraso nessas atualizações pode causar uma queda nas vendas e um prejuízo indesejável.

Uma maneira de evitar o atraso da publicação é subir o sistema com a atualização implementada, mas desligada, e que por meio de alguma alteração no banco de dados é ativada. No caso do nosso sistema de pedidos, vamos fazer um mecanismo de ofertas que é acionado com uma alteração no arquivo de propriedades:

mensagem de oferta

mensagem=Compre

1 hot dog e ganhe 1

suco de laranja !

```
debug=
```

```
1
```

A oferta será exibida através de um serviço que devolve uma classe do tipo MensagemDTO:

```
public class MensagemDTO
```

```
{
```

```
private
```

```
String mensagem;
```

```
private
```

```
String servidor;
```

```
private
```

```
String debug;
```

```
public MensagemDTO(String mensagem, String servidor, String debug)
```

```
{
```

```
this
```

```
.mensagem = mensagem;
```

```
this
```

```
.servidor = servidor;
```

```
this
```

```
.debug = debug;
```

```
}
```

A classe IndexController contém o serviço que devolve a oferta:

```
@GetMapping("/oferta") @ResponseBody public MensagemDTO  
getMessage(HttpServletRequest request)
```

```
{
```

```
return new MensagemDTO(this.message,request.getServerName()
```

```
+ ":" + request.
```

```
getServerPort(),this.debug)
```

```
;
```

```
}
```

O método getMessage usa as propriedades message e debug, esperando parâmetros com o mesmo nome.

```
@Value("${mensagem:nenhuma}") private
```

```
String message;
```

```
@Value("${debug:0}") private String debug;
```

No arquivo index.html, adicionamos um tratamento para exibir uma promoção se o conteúdo da variável oferta for diferente de nenhuma, e exibir a informação de debug se for igual a 1.

```
<div class="alert alert-danger" ng-show="oferta!='nenhuma'">
```

```

```

```
 {{oferta}}
```

```
<div ng-show="debug==1">
```

```
<br/>
```

```

```

```
<strong>{ {servidor} }</strong>
```

```
</div> </div>
```

E também o arquivo delivery.js será adicionado à rotina para ler a oferta:

```
var carregaOferta= function ()  
{  
$http.get(  
"/oferta").success(function (data)  
{  
$scope.oferta = data[  
"mensagem"  
];  
$scope.servidor = data[  
"servidor"  
];  
$scope.debug = data[  
"debug"  
];  
}).error(
```

```
function (data, status)
{
  $scope.message =
  "Aconteceu um problema: "
  + data;
}

};

...
carregaOferta();
```

Para ajudar no debug de nossos testes, criaremos um serviço para exibir de qual servidor vem a solicitação (o que será mais útil quando testarmos com vários servidores no capítulo sobre Cloud):

```
@GetMapping("/servidor") @ResponseBody public String
server(HttpServletRequest request)

{
  System.out.println(
    ">>> Chamada end-point servidor..."
);

  return request.getServerName() + ":"
```

```
+request.getServerPort());  
}
```

Agora o serviço de oferta retorna a seguinte mensagem:

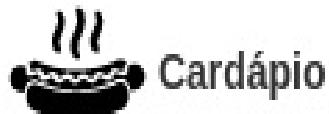


```
{  
    mensagem: "Compre 1 hot dog e ganhe 1 suco de laranja | Aproveite!",  
    servidor: "cascao:8080",  
    debug: "1"  
}
```

Figura 8.6: Servidor com oferta

O nosso sistema agora exibe com sucesso a nossa oferta na tela de pedidos:

Delivery - Novo Pedido



- Green Dog tradicional picante [R\$27]
- Green Dog max salada [R\$30]
- Green Dog tradicional [R\$25]

Subtotal: R\$0

Fazer o pedido



Compre 1 hot dog e ganhe 1 suco de laranja ! Aproveite!



cascao:8081

Figura 8.7: Servidor com oferta

Pronto! O Rodrigo pode comemorar, pois ele conseguirá publicar novas ofertas facilmente.

8.3 Região de entrega

Qualquer pessoa conectada à internet pode fazer um pedido, mas o sistema, de alguma maneira, precisa validar se o cliente está dentro da região de atendimento. Considerando que a Green Dog Delivery fica em Aracaju, como podemos verificar se o CEP da região de entrega é válido?

Para resolver esse problema, vamos usar o serviço gratuito do ViaCEP (<https://viacep.com.br>), no qual, ao informar o CEP, descobrimos se o endereço é válido, como esse:

```
$ http https://viacep.com.br/ws/49037475
```

```
/json
```

```
{
```

```
  "bairro": "Atalaia"
```

```
,
```

```
  "cep": "49037-475"
```

```
,
```

"complemento": "até 1088/1089"

,

"ddd": "79"

,

"gia": ""

,

"ibge": "2800308"

,

"localidade": "Aracaju"

,

"logradouro": "Avenida Santos Dumont"

,

"siafi": "3105"

,

"uf": "SE"

}

Ou inválido, como esse de São Paulo:

\$ http https://viacep.com.br/ws/04101300

/json

{

"bairro": "Vila Mariana"

,

"cep": "04101-300"

,

"complemento": "de 2771 a 5049 - lado ímpar"

,

"ddd": "11"

,

"gia": "1004"

,

"ibge": "3550308"

,

"localidade": "São Paulo"

,

"logradouro": "Rua Vergueiro"

,

"siafi": "7107"

,

"uf": "SP"

}

Precisamos criar um serviço `ValidaCEPEntrega` que receba o CEP do cliente, consulte o site ViaCEP e retorne true se for um CEP de Aracaju e, em qualquer outro caso, retorne false.

A lógica parece simples, mas como consultar o serviço de um site remoto?
Agora entra a mágica do `RestTemplate`.

8.4 Chamadas ReST remotas

O RestTemplate foi criado para fazer chamadas remotas HTTP, mas nas versões posteriores do Spring Framework ele será descontinuado. Desde o Spring 5 junto com o WebFlux veio o seu substituto: WebClient, que deve ser a opção considerada em novos projetos (teremos mais detalhes dessa história no capítulo Spring reativo).

O RestTemplate retorna em sua chamada um objeto do tipo ResponseEntity que possui, além do body, o status code da resposta.

Vamos chamar o serviço do ViaCEP. Note que estamos usando o método getForEntity (existem outros, como o postForEntity) e passamos como parâmetro a URL de chamada (cepURL) e o tipo de objeto que deve retornar (nesse caso String).

```
RestTemplate restTemplate = new  
RestTemplate();  
  
String cepURL =  
"https://viacep.com.br/ws/49037475/json"  
;  
ResponseEntity<String> response =
```

```
restTemplate.getForEntity(cepURL, String.class);
```

Em seguida, exibimos os valores obtidos na chamada:

```
System.out.println("Response Status:"  
+response.getStatusCode());  
  
System.out.println(  
"Response Body:"  
);  
  
System.out.println(response.getBody());
```

O resultado obtido é a resposta do JSON junto com o status 200:

Response Status:200

OK

Response Body:

{

"cep": "49037-475"

,

"logradouro": "Avenida Santos Dumont"

,

"complemento": "até 1088/1089"

,

"bairro": "Atalaia"

,

"localidade": "Aracaju"

,

"uf": "SE"

,

"ibge": "2800308"

,

"gia": ""

,

```
"ddd": "79"  
,  
  
"siafi": "3105"  
  
}
```

Para facilitar o trabalho com a resposta do serviço, podemos usar uma classe customizada em vez de trabalhar com String. Vamos usar a classe CEP:

```
public class CEP  
{  
  
    private  
    String cep;  
  
    private  
    String logradouro;
```

```
private  
String complemento;
```

```
private  
String bairro;  
  
private  
String localidade;
```

```
private String uf;
```

A chamada agora muda um pouco para obtermos um retorno do tipo CEP:

```
ResponseEntity<CEP> responseCEP = restTemplate.getForEntity(urlBuscaCEP,  
CEP.class);
```

O que resta agora é apenas uma lógica para comparar se o resultado da localidade é "Aracaju" e se a UF é "SE":

```
boolean resultado = false  
;
```

```
CEP wsCEP = responseCEP.getBody();

if (wsCEP!=null

) {

if (wsCEP.getUf().equals("SE"
) &&
wsCEP.getLocalidade().equals(
"Aracaju"

)) {

resultado =
true

;

}

}
```

Com isso, completamos o nosso serviço para testar o CEP do cliente, que pode ser usado da seguinte maneira:

```
if

(validaCEPEntrega.processa(cepDoCliente))

processaPedido();
```

8.5 O que aprendemos

Os códigos-fontes do projeto estão em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/rest>.

Certifique-se de que você aprendeu:

- sobre o uso e a aplicação do HATEOAS;
- sobre o uso e a ativação do HAL Browser;
- sobre a integração do Spring Data HATEOAS com o sistema em AngularJS;
- a usar variáveis personalizadas no application.properties;
- a usar o RestTemplate para integrar serviços fora da aplicação.

No próximo capítulo, veremos como testar a aplicação com as soluções integradas do Spring Boot.

Capítulo 9

Testando sua aplicação

O Rodrigo tem uma equipe pronta para trabalhar com os serviços ReST do seu sistema. Mas como garantir que eles estão funcionando corretamente? Como garantir que, depois de uma alteração, algum serviço não alterou um resultado de outro? A resposta para isso tudo é trabalhar com testes.

9.1 Testes unitários

O Spring Framework é bem famoso pela sua facilidade em criar testes unitários. Desde a versão 4.3, a sua classe principal de rodar testes SpringJUnit4ClassRunner foi substituída pela SpringRunner. Com ela, conseguimos subir o application context do Spring e fazer funcionar todas as injeções de dependência de seu teste. Sem ela, seria necessário instanciar manualmente cada objeto envolvido no teste.

Podemos usar a SpringRunner padrão para executar os nossos testes unitários. Para testar, por exemplo, se o nosso serviço de busca de clientes está funcionando adequadamente, fazemos:

```
@RunWith(SpringRunner.class) @SpringBootTest public class  
ClienteRepositoryTest
```

```
{
```

```
@Autowired
```

```
ClienteRepository repository;
```

O nosso sistema faz uma carga inicial de dois clientes. Portanto, vamos testar se o método findAll retorna um total maior do que o valor 1.

```
@Test public void buscaClientesCadastrados()  
{  
  
    Page<Cliente> clientes =  
        this.repository.findAll(new PageRequest(0, 10  
    ));  
  
    assertThat(clientes.getTotalElements()).isGreaterThan(  
        1  
    );  
  
}  
  
}
```

Rodar o teste inteiro demorou pouco mais de seis segundos e o teste do método bem menos de um segundo. Essa diferença existe porque o teste faz o Spring Boot subir uma instância para rodar os testes.

Package Explorer JUnit

Finished after 6,121 seconds

Runs: 1/1 Errors: 0 Failures: 0

buscaClientesCadastrados [Runner: JUnit 4] (0,145 s)

Figura 9.1: Rodando o teste de busca

Agora vamos testar a busca pelo nome, inicialmente com um valor não existente, e depois com um valor válido.

```
@Test public void buscaClienteFernando()
{
    Cliente clienteNaoEncontrado =
        this.repository.findByName("Fernando"
    );
    assertThat(clienteNaoEncontrado).isNull();

    Cliente cliente =
        this.repository.findByName("Fernando Boaglio"
    );
    assertThat(cliente).isNotNull();
    assertThat(cliente.getName()).isEqualTo(
        "Fernando Boaglio"
    )
}
```

```
};

assertThat(cliente.getEndereco()).isEqualTo(
    "Sampa"
);

}

}
```

Os testes unitários funcionam, pois testam integralmente algumas rotinas de busca. Entretanto, eles ainda não testam o resultado de um serviço ReST, ou seja, não testam o resultado de todo o conjunto de serviços, o que é chamado de teste de integração.

9.2 Testes de integração

Para testar o serviço como um todo, usamos uma opção interna do Spring chamada MockMvc, que é uma maneira fácil e eficiente de testar um serviço ReST. Com apenas uma linha de código, é possível testar o serviço, o seu retorno, exibir a saída no console, entre outras opções.

Outra alternativa seria usar o RestTemplate para fazer testes de integração, mas ele é bem mais trabalhoso de codificar. A nossa classe de teste usa o WebApplicationContext para instanciar o objeto mvc usado em todos os testes:

```
@RunWith(SpringRunner.class) @SpringBootTest public class  
GreenDogDeliveryApplicationTests  
{  
  
    @Autowired private  
    WebApplicationContext context;  
  
    private  
    MockMvc mvc;
```

```
@Before public void setUp()  
{  
    this  
.mvc = MockMvcBuilders.webAppContextSetup  
        (  
    this  
.context).build();  
}
```

Nesse teste inicial, chamamos a raiz dos serviços /api e testamos se ela contém um serviço chamado clientes. Usamos o método print para exibir a saída do serviço no console.

```
@Test public void testHome() throws Exception
```

```
{
```

```
String URL1=
```

```
"/api"
```

```
;
```

```
System.out.println(
```

```
this
```

```
.mvc.perform(get(URL1))  
.andDo(print());
```

```
this
```

```
.mvc.perform(get(URL1))  
.andExpect(status().isOk())  
.andExpect(content().string(containsString(  
"clientes"  
)));  
}
```

Em seguida, testamos se o preço do item 2 é igual a 30:

```
@Test public void findItem2() throws Exception
```

```
{
```

```
String URL5=
```

```
"/api/itens/2"  
;
```

```
System.out.println(  
this  
.mvc.perform(get(URL5).andDo(print()));
```

```
this  
.mvc.perform(  
get(URL5))  
.andExpect(status().isOk())  
.andExpect(jsonPath(  
"preco", equalTo(30.0  
)));  
}
```

E finalmente, testamos o serviço de novos pedidos, em que o valor total é 57, entre outras validações.

```
@Test public void cadasraNovoPedido() throws Exception  
{
```

```
String URL4=  
"/rest/pedido/novo/1/1,2"
```

;

```
System.out.println(  
this  
.mvc.perform(get(URL4))  
.andDo(print()));  
  
this  
.mvc.perform(  
get(URL4))  
.andExpect(status().isOk())  
.andExpect(jsonPath(  
"valorTotal", is(57.0  
)))  
.andExpect(jsonPath(  
"pedido", greaterThan(3  
)))  
.andExpect(jsonPath(  
"mensagem"  
,
```

```
equalTo(  
    "Pedido efetuado com sucesso"  
));  
}
```

Dessa maneira, conseguimos testar integralmente o serviço de pedidos e a busca de itens.

9.3 O que aprendemos

Os fontes do projeto estão em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/testes>.

Certifique-se de que você aprendeu:

- a rodar testes unitários com Spring Boot;
- a rodar testes de integração com Spring Boot;

No próximo capítulo, veremos como disponibilizar a nossa aplicação em um banco de dados NoSQL.

Capítulo 10

Conversando com banco de dados NoSQL

Mais pedidos, mais problemas

Como estratégia de crescimento da Green Dog Delivery, Rodrigo criou uma franquia, terceirizou seus pedidos para outras filiais e passou para elas o segredo de montar o produto e revender a matéria prima. Nesse novo contexto surgiu a necessidade de criar um controle de estoque centralizado, que recebe pedidos de vários lugares e depois centraliza o material gasto. Sendo o controle de estoque uma responsabilidade diferente da de fazer pedidos, Rodrigo decidiu fazer um microsserviço que recebe os pedidos junto com uma área de consulta.

10.1 Arquitetura

A arquitetura do controle de estoque armazenará os dados usando um banco de dados NoSQL chamado MongoDB. O MongoDB é um document database (banco de dados de documentos), mas não os documentos da família Microsoft, e sim aqueles com informações no formato JSON. A ideia é o documento representar toda a informação necessária, sem a restrição dos bancos relacionais.

O uso do Spring Boot com Spring Data facilita muito o trabalho com bancos de dados NoSQL, pois, assim como os bancos relacionais, muitas rotinas estão prontas, o nosso trabalho é apenas declarar a interface adequadamente.

Com o novo projeto, temos um novo repositório em <https://github.com/boaglio/spring-boot-greendogdelivery-estoque-casadocodigo>. Para subir o ambiente, precisamos baixar o projeto e rodar o comando:

```
$ docker-compose up
```

Com o ambiente no ar, teremos os serviços disponíveis:

MongoDB - Banco de dados acessível na porta 27017.

Mongo Express - Interface web para o MongoDB acessível em

http://localhost:8081.



Mongo Express



Viewing Database: test

Collections

Collection Name [+ Create collection](#)

[View](#) [Export](#) [\[JSON\]](#) [Import](#) estoque [Del](#)

Database Stats

Collections (incl. system.namespaces)	1
Data Size	13.5 MB
Storage Size	2.06 MB
Avg Obj Size #	135 Bytes
Objects #	100000
Indexes #	1
Index Size	909 KB

Figura 10.1: Mongo Express

Implementação

No MongoDB as tabelas são chamadas de collections, e no nosso sistema temos a tabela de estoque, que é representada pela classe:

```
@Document(collection = "estoque") public class Estoque implements  
Serializable
```

```
{
```

```
@Id
```

```
private
```

```
String id;
```

```
private
```

```
Long itemId;
```

```
private Long quantidade;
```

A anotação Document faz a ligação da classe com a collection, da mesma maneira que a anotação Entity liga a classe a uma tabela.

A implementação do CRUD é feita apenas com essa classe herdando todas as rotinas prontas do Spring Boot:

```
public interface EstoqueRepository extends MongoRepository<Estoque, String>
{
}
```

E o serviço que recebe a requisição chama o método save para persistir no banco de dados a informação de estoque recebida.

```
@PostMapping("/atualiza") public String atualiza(@RequestBody Estoque
estoque)
{
    System.out.println(
        "Recebido via REST: "
        +estoque);
    estoqueRepository.save(estoque);

    return "Ok"
```

```
;
```

```
}
```

Simulando uma chamada do cliente, informamos um pedido de 99 Green Dog tradicional:

```
$ http POST :9000/api/atualiza itemId=3 quantidade=99
```

Ok

No log do Spring Boot aparece:

```
Recebido via REST: Estoque [itemId=3, quantidade=99]
```

Acessando o serviço para consultar os últimos pedidos, conseguimos confirmar o cadastro corretamente:

```
$ http GET :9000
```

```
/api/ultimos
```

```
[
```

{

"id": "5ff00bc898aa26573fef9bb0"

,

"itemId": 1

,

"quantidade": 99

},

...

E também verificar pelo browser:



Mongo Express



Viewing Collection: estoque

[New Document](#)[New Index](#)[Simple](#)[Advanced](#)

Key

Value

String

[Find](#) [Delete all 100001 documents retrieved](#)[← First](#)[← Prev](#)[Next →](#)[Last →](#)

_id ▼	itemId	quantidade ▲	_class
 5ff00e9f98aa26573fef9bb1	3	99	com.boaglio.casadocodigo.greendogdelivery.es

Figura 10.2: Novo registro cadastrado

Atualizando a aplicação

No sistema atual do Green Dog Delivery, usaremos o RestTemplate para fazer essa atualização, criando uma classe apenas para esse serviço:

```
public class AtualizaEstoqueService
{
    public void send(Pedido pedido) throws Exception
    {
        RestTemplate restTemplate =
            new RestTemplate();
        String resourceUrl =
            "http://localhost:9000/api/atualiza";
```

Definida a URL de chamada, enviaremos uma chamada ao pedido recebido para a rotina atualizar o estoque de cada item:

```
for  
(Item item : pedido.getItens()) {  
    System.out.println(  
        "Enviando requisicao - atualizando estoque - [ "  
        +  
        item.getNome() +  
        " ] ..."  
    );  
  
    Estoque estoque =  
        new Estoque(item.getId(), 1  
    );  
  
    HttpEntity<Estoque> requestEstoque =  
        new  
        HttpEntity<>(estoque);  
  
    String responseEstoque =  
        restTemplate.postForObject(resourceUrl, requestEstoque, String.class);  
  
    System.out.println(  
        responseEstoque  
    );  
}
```

```
"Resposta: "  
+responseEstoque);  
}  
  
}
```

Agora todo pedido novo terá uma chamada para a atualização de estoque. A integração dos dois sistemas está completa.

Um exemplo de chamada de pedido:

Enviando requisicao - atualizando estoque - [Green Dog max salada] ...

Resposta: Ok

Enviando requisicao - atualizando estoque - [Green Dog tradicional] ...

Resposta: Ok

Arquitetura atual

O nosso cenário atual pode ser dividido nas seguintes camadas:

Usuários acessando o sistema.

Sistema web de pedidos com Spring Boot.

Banco de dados MySQL.

Sistema de controle de estoque com Spring Boot.

Banco de dados MongoDB.

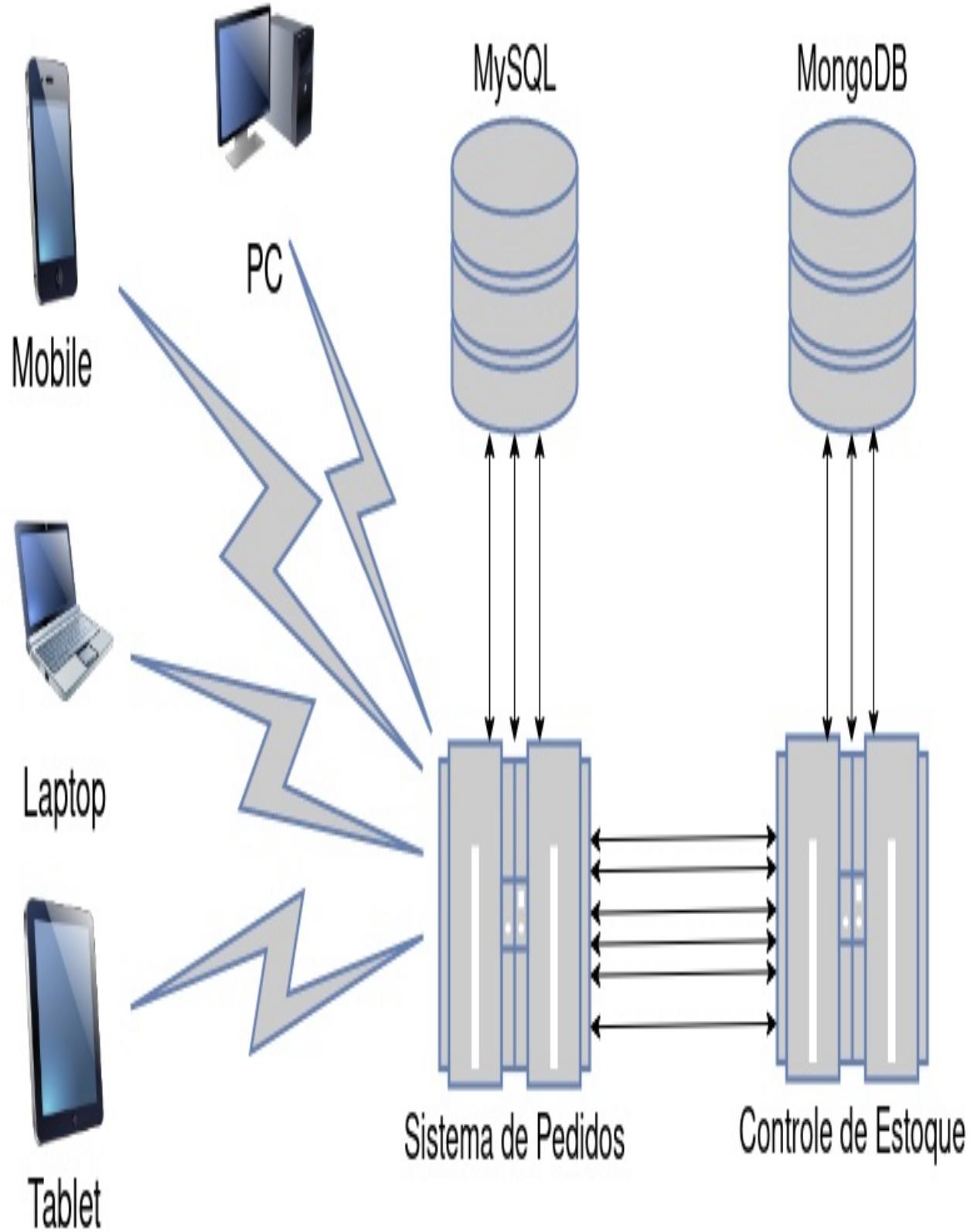


Figura 10.3: Arquitetura atual

10.2 Documentando a API

Agora Rodrigo precisa expor para suas filias a API de atualização de estoque. Uma maneira bem prática de fazer isso é usando a especificação OpenAPI (<https://github.com/OAI/OpenAPI-Specification>).

A melhor forma de ativar essa documentação para o nosso projeto é através da biblioteca SpringFox:

```
<dependency>
```

```
<groupId>io.springfox</groupId>
```

```
<artifactId>springfox-boot-starter</artifactId>
```

```
<version>3.0.0</version> </dependency>
```

Ativando a biblioteca, todos os objetos do tipo Controller são analisados e uma documentação em HTML é gerada; entretanto algumas internas do Spring Boot também aparecem, o que nos obriga a criar uma classe de configuração para declarar uma documentação apenas das classes do tipo RestController.

```
@Configuration public class SwaggerConfig  
{  
  
    @Bean  
  
    public Docket api()  
    {  
  
        return new Docket(DocumentationType.OAS_30)  
            .  
            select()  
            .  
            apis(RequestHandlerSelectors.withClassAnnotation(RestController.class))
```

```
)  
.  
paths(PathSelectors.any()  
)  
.  
build()  
;  
}  
}
```

Observem também que na configuração escolhemos o tipo OAS_30 (OpenAPI Specification 3).

Reiniciando o projeto temos a documentação pronta para ver os atributos e testar a API através do endereço <http://127.0.0.1:9000/swagger-ui/index.html>.



Api Documentation

1.0 OAS3

<http://127.0.0.1:9000/v3/api-docs>

Api Documentation

[Terms of service](#)

[Apache 2.0](#)

Servers

[http://127.0.0.1:9000 - Inferred Url](http://127.0.0.1:9000) ▾

estoque-controller Estoque Controller >

log-controller Log Controller >

Schemas >

Figura 10.4: Documentação total

Entretanto, a documentação mostra o LogController e vários métodos de consulta de estoque que desejamos esconder, exibindo apenas as rotinas de atualização. Podemos fazer um novo ajuste na classe de configuração, mas a maneira mais fácil é usar a anotação ApiIgnore nas classes e métodos que desejamos omitir:

```
@ApiIgnore @GetMapping("/lista") public List<Estoque> getTodoEstoque()  
{  
  
    return  
    estoqueRepository.findAll();  
}
```

Depois dessa alteração, a documentação fica correta para as filiais exibindo apenas o necessário.

 Swagger
Supported by SMARTBEAR

Select a definition ▾

Api Documentation 1.0 OAS3

<http://127.0.0.1:9000/v3/api-docs>

[Api Documentation](#)

[Terms of service](#)

[Apache 2.0](#)

Servers

[http://127.0.0.1:9000 - Inferred Url](http://127.0.0.1:9000) ▾

estoque-controller Estoque Controller ▾

POST /api/atualiza atualiza

Schemas >

Figura 10.5: Documentação parcial

10.3 O que aprendemos

Para acompanhar o projeto completo até aqui, acesse o branch `mongodb`, em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/mongodb>.

Certifique-se de que você aprendeu:

- a usar Spring Data com MongoDB;
- a usar o Mongo Express para navegar nos dados do MongoDB;
- a integrar sistemas diferentes com RestTemplate;
- a documentar suas APIs com Swagger.

No próximo capítulo, veremos como disponibilizar a nossa nova aplicação utilizando o conceito de filas.

Capítulo 11

Utilizando filas

Mais demanda, mais problemas

O negócio cresceu bastante e um novo problema chegou para Rodrigo resolver: os usuários estão reclamando que o pedido está demorando muito para responder, principalmente em horário de pico.

Fazendo uma análise em toda a arquitetura, o problema identificado foi que em alguns momentos a integração com o controle de estoque fica lenta e demora para responder.

A lentidão no sistema de controle de estoque atrapalha o sistema de pedidos. Isso acontece porque a integração entre eles é on-line na chamada de RestTemplate, que existe na classe AtualizaEstoqueService vista no capítulo anterior. Enquanto o controle de estoque não responde, o sistema de pedidos fica parado esperando a resposta.

A solução para esse problema é fazer a integração assíncrona por dois motivos:

Já foi observado que a lentidão atrapalha o sistema de pedidos.

O pedido não precisa da informação do controle de estoque, portanto a atualização de estoque pode acontecer depois do pedido sem problemas.

A solução assíncrona mais usada no mercado é a fila de mensagens, ou simplesmente filas.

11.1 Filas em todo lugar

Nos sites de e-commerce mais acessados do mundo, o uso de filas é a solução utilizada para garantir que o site não caia e que a sua compra aconteça com certeza.

A ideia de fila é bem simples, em vez de executar o processo na hora (síncrono), ele é armazenado em algum lugar para ser processado depois. A analogia pode ser feita com uma fila de verdade, quando em um restaurante há vários clientes e só um cozinheiro. Por conseguir fazer apenas um pedido por vez, o cozinheiro coloca os pedidos em uma fila e pega um por um.

Na literatura, esse tipo de fila é conhecida como fila de mensagem (message queue). O processo em si é chamado de mensagem, dentro dela podemos colocar as informações de qualquer coisa, no nosso caso será do pedido para atualização de estoque. Quem manda a mensagem para processar é chamado de produtor (producer), e quem lê e processa essa mensagem é chamado de consumidor (consumer).

Portanto, um processo que seria síncrono pode ser dividido em duas partes: colocar parte dele em uma mensagem e deixar para processar depois.

Um bom exemplo do que causa lentidão em vários sistemas é a geração de relatórios. Normalmente, esse processo usa complexos SQLs que usam muitos recursos do banco de dados e, por isso, demoram para responder ao usuário.

Cenário de exemplo:

O usuário solicita ao sistema um relatório mensal.

O sistema repassa para o banco de dados processar.

O sistema pega o resultado e devolve um PDF para o usuário.

Se muitos usuários resolvem gerar relatórios, corremos o risco do sistema ficar indisponível, pois o banco de dados não consegue atender a todas as requisições.

A solução aqui é simples: colocamos o pedido de processamento do relatório em uma fila e enviamos o PDF por e-mail. Nesse exemplo, além de resolver o problema da lentidão da resposta para o usuário, resolvemos o problema de uma possível indisponibilidade do banco de dados.

Cenário de exemplo melhorado:

Parte do usuário:

O usuário solicita ao sistema um relatório mensal.

O sistema envia/produz a mensagem para a fila de processamento.

O sistema avisa o usuário que o relatório será enviado por e-mail.

Parte do sistema:

O sistema lê/consume a mensagem para a fila de processamento.

O sistema repassa para o banco de dados processar.

O sistema pega o resultado e devolve um PDF para o e-mail do usuário.

Existem várias opções pagas e gratuitas que gerenciam filas, no nosso sistema vamos utilizar o RabbitMQ (<https://www.rabbitmq.com>).

Fazendo uma análise do cenário melhorado, o processamento assíncrono poderia ser feito manualmente sem um servidor de filas, gravando as mensagens em um banco de dados comum e fazendo o envio para os sistemas via RestTemplate. Entretanto, o servidor de filas oferece vários recursos interessantes, como o reprocessamento de mensagens, a configuração dinâmica de consumidores/produtores, o processamento da mensagem em diferentes filas, o controle de mensagens via interface web, além de abordagens diferentes de fila, como a de envio/notificação (push/subscribe), por meio da qual o servidor de filas avisa/notifica todos os consumidores que o produtor enviou uma nova mensagem (essa implementação é muito usada em redes sociais: quando seguimos algum perfil, recebemos notificações das novidades).

Fila fazendo o pedido mais rápido

Em todo o processo de pedido, faremos uma mudança importante, pois atualmente ele é síncrono:

O cliente faz o pedido no sistema.

O sistema de pedidos atualiza o seu banco de dados.

O sistema de pedidos chama controle de estoque.

O controle de estoque atualiza o seu banco de dados.

O cliente recebe a confirmação do pedido.

A ideia é que o mesmo processo divida-se em duas partes:

Parte do pedido do cliente:

O cliente faz pedido no sistema.

O sistema de pedidos atualiza o seu banco de dados.

O sistema de pedidos envia mensagem de estoque.

O cliente recebe a confirmação do pedido.

Parte da integração:

O controle de estoque lê a mensagem.

O controle de estoque atualiza o seu banco de dados.

A nova arquitetura terá o RabbitMQ recebendo as mensagens de pedido e

enviando mensagens para o controle de estoque. Nessa nova arquitetura, se o sistema receber milhares de pedidos, a performance não será afetada, pois o controle de estoque consumirá as mensagens no seu ritmo, pedido por pedido.

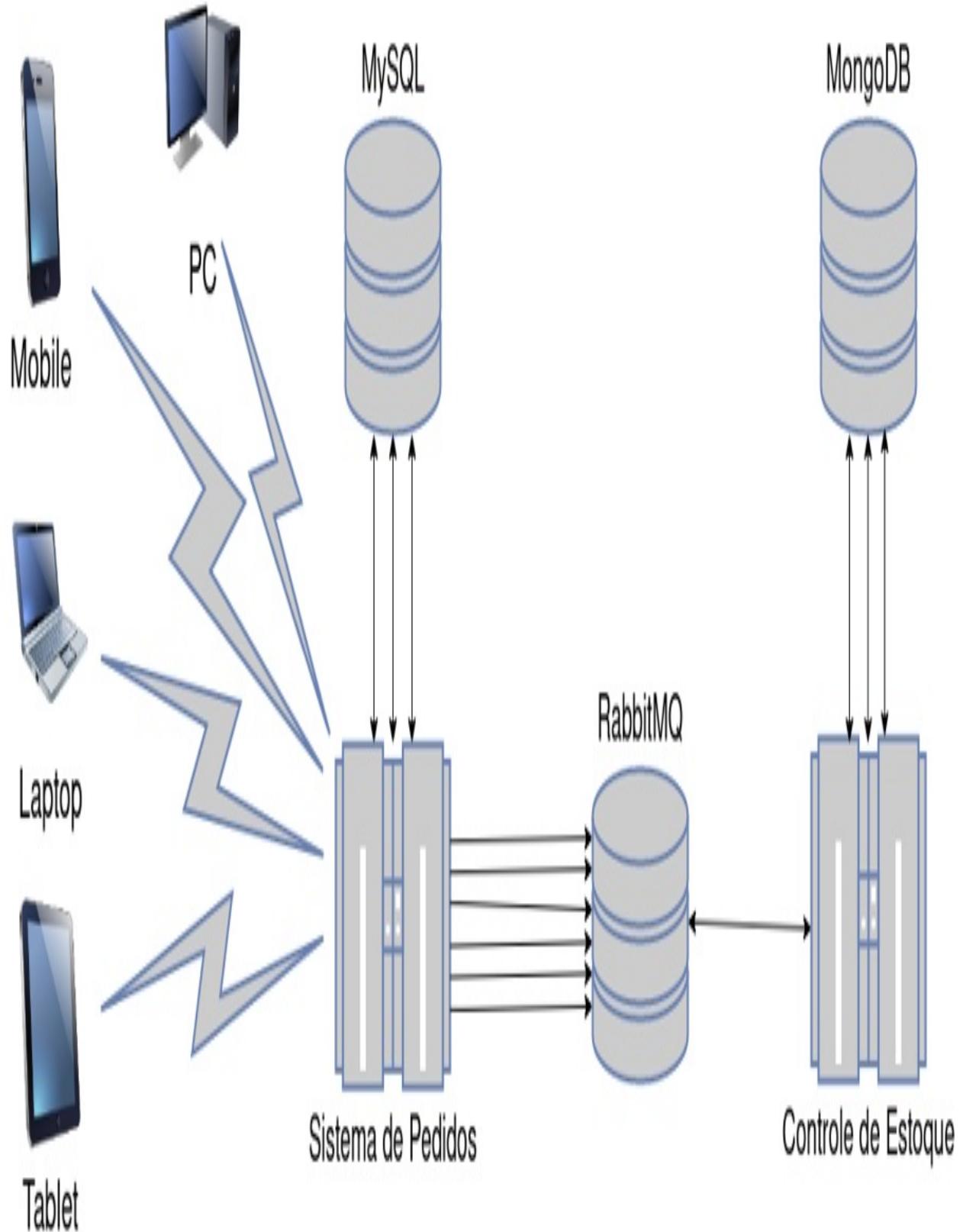


Figura 11.1: Arquitetura planejada

11.2 Ajustando a produção de mensagens

No projeto de controle de estoque, dentro do arquivo docker-compose.yaml, além do MongoDB também temos a configuração do RabbitMQ, com os serviços disponíveis:

RabbitMQ - Fila de mensagens acessível na porta 5672;

RabbitMQ Management - Interface web para o MongoDB acessível em http://localhost:8082 (usuário guest / senha guest).

Nos dois sistemas, precisamos adicionar as dependências de bibliotecas:

```
<!-- RabbitMQ --> <dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-amqp</artifactId> </dependency> <dependency>
```

```
<groupId>org.springframework.amqp</groupId>
```

```
<artifactId>spring-rabbit-test</artifactId>
```

```
<scope>test</scope> </dependency>
```

A classe de configuração para filas, além de exigir a anotação `EnableRabbit`, nos obriga a declarar vários beans. Vamos destacar o mais importante deles, que é a implementação de filas usada com `SimpleMessageListenerContainer`:

```
@EnableRabbit @Configuration public class RabbitmqConfig
```

```
{
```

```
    @Bean
```

```
        SimpleMessageListenerContainer container
```

```
(
```

```
            ConnectionFactory connectionFactory,
```

```
    MessageListenerAdapter listenerAdapter)

{

    SimpleMessageListenerContainer container =  
  
        new  
            SimpleMessageListenerContainer();  
            container.setConnectionFactory(connectionFactory);  
            container.setQueueNames(  
                "springboot.boaglio.queue"  
            );  
            container.setMessageListener(listenerAdapter);

    return  
        container;  
}
```

Com essa declaração nos sistemas, podemos declarar e usar produtores e consumidores da fila chamada `springboot.boaglio.queue`.

No nosso sistema de controle de estoque, vamos criar uma classe para gravar as mensagens que chegam ao banco de dados apenas para consulta:

```
@Document(collection = "logDaFila") public class LogFila  
{  
  
    @Id private String id;    private String descricao; private String mensagem;
```

Temos também um repositório LogFilaRepository para gravar os dados:

```
public interface LogFilaRepository extends MongoRepository<LogFila, String>  
{  
}
```

E finalmente temos a classe Consumer, que se conecta ao RabbitMQ e recebe a nova mensagem. Inicialmente declaramos as variáveis necessárias de repositório:

```
@Component public class Consumer
```

```
{
```

```
@Autowired private  
LogFilaRepository logFilaRepository;
```

```
@Autowired private EstoqueRepository estoqueRepository;
```

Para ligar o consumidor na fila correta, usamos a anotação RabbitListener no método receiveMessage, que receberá do RabbitMQ a mensagem convertida no objeto do tipo Estoque, para posteriormente ser gravado nas collections de logDaFila e estoque.

```
@RabbitListener(queues = { "springboot.boaglio.queue" }) public void receiveMessage(@Payload Estoque mensagem)
```

```
{
```

```
    System.out.println(
```

```
        "Recebido via fila: <" + mensagem + ">"
```

```
);
```

```
    logFilaRepository.save(
```

```
        new LogFila("Recebendo"
```

```
        , mensagem.toString()));
```

```
    System.out.println(
```

```
        "Gravando: <" + mensagem + ">"
```

```
);
```

```
    estoqueRepository.save(mensagem);
```

```
}
```

No sistema de pedidos, além de arrumar as dependências e de criar a classe de configuração, precisamos alterar a chamada da atualização de estoque, mudando de RestTemplate para RabbitMQ.

Para o envio de mensagens, o Spring Boot oferece o RabbitTemplate, que permite facilmente o envio com uma linha de código.

```
@Component public class Producer
```

```
{
```

```
@Autowired
```

```
private RabbitTemplate rabbitTemplate;
```

Para cada item de pedido, a rotina produz uma mensagem na fila:

```
public void send(Pedido pedido) throws Exception
```

```
{
```

```
for  
(Item item : pedido.getItens()) {  
    System.out.println(  
        "Enviando mensagem - atualizando estoque - [ "  
        +  
        item.getNome() +  
        " ] ..."  
    );  
  
    Estoque estoqueMSG =  
        new Estoque(item.getId(), 1  
    );  
  
    rabbitTemplate.convertAndSend(  
        "springboot.boaglio.queue"  
        , estoqueMSG);  
}  
}
```

Agora só precisamos atualizar o método processar da classe AtualizaEstoque para substituir o RestTemplate pela chamada do Producer:

```
@Autowired private  
Producer producer;  
  
public void processar(Pedido pedido)  
{  
  
try  
{  
  
// atualiza estoque  
  
producer.send(pedido);  
}  
  
catch  
(Exception e) {  
    e.printStackTrace();  
}
```

Nossa integração está completa, vamos aos testes.

11.3 Validando a integração com filas

Fazendo um pedido de um Green Dog max salada, observamos no log do Spring Boot a saída:

Enviando mensagem - atualizando estoque - [Green Dog max salada] ...

Acessando a interface web do RabbitMQ, verificamos que existe uma mensagem na fila `springboot.boaglio.queue` usada no producer esperando para ser consumida:



RabbitMQ 3.8.9 Erlang 23.1.2

Refresh every 5 seconds ▾

Virtual host All ▾

[Overview](#) [Connections](#) [Channels](#) [Exchanges](#)[Queues](#)

Cluster rabbit@5e51be79dee5

Admin

User guest Log out

Queues

[All queues \(1\)](#)

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
springboot.boaglio.queue	classic		idle	1	0	1	0.00/s			

[Add a new queue](#)[HTTP API](#) [Server Docs](#) [Tutorials](#) [Community Support](#) [Community Slack](#) [Commercial Support](#) [Plugins](#) [GitHub](#)[Changelog](#)

Figura 11.2: Interface web do RabbitMQ

Iniciando a aplicação do controle de estoque, o consumer se conecta ao RabbitMQ automaticamente e baixa as mensagens gravando no estoque local.

Na saída do log do Spring Boot aparecem as informações das mensagens enviadas pelo sistema de pedidos:

Recebido via fila: <Estoque [itemId=2, quantidade=1

]>

Gravando: <Estoque [itemId=

2, quantidade=1]>

Consultando o MongoDB conseguimos ver a mensagem gravada no controle de estoque:



Viewing Collection: estoque

[New Document](#)[New Index](#)[Simple](#)[Advanced](#)

Key

Value

String

[Find](#)[Delete all 100009 documents retrieved](#)[First](#)[Prev](#)[Next](#)[Last](#)

_id ▼	itemId	quantidade	class
	2	1	com.boaglio.casadocodigo.greendogdelivery.estoque...

Figura 11.3: Controle de estoque atualizado

E na log das filas:



Viewing Collection: logDaFila

[New Document](#)[New Index](#)[Simple](#)[Advanced](#)

Key

Value

String

[Find](#)[Delete all 1 documents retrieved](#)

_id	descricao	mensagem	_class
5ff0e9c3e129ae4423d03d3e	Recebendo	Estoque [itemId=2, quantidade=1]	com.boaglio.casadocodigo.greendogdelivery.estoque

Figura 11.4: Log das filas atualizado

11.4 O que aprendemos

Certifique-se de que você aprendeu:

o conceito básico de filas de mensagens;

a usar o Spring Boot com RabbitMQ;

a usar o RabbitMQ Management para navegar nos dados do RabbitMQ;

a integrar sistemas diferentes com producer/consumer.

No próximo capítulo, veremos as novidades do Spring reativo e como isso influencia em nossos sistemas.

Capítulo 12

Spring Reativo

Mundo Reativo

O movimento reativo está em evidência hoje com vários frameworks, mas ele já existe há vários anos, sendo o projeto Netty (<https://netty.io>) a implementação de servidor mais conhecida.

O manifesto reativo (<https://www.reactivemanifesto.org/pt-BR>) resume os sistemas reativos em: responsivos, resilientes, elásticos e orientados a mensagens. Todas as implementações reativas usam soluções sem bloqueio de I/O e controlam as requisições por eventos.

No mundo Java, quando foi lançado no Java 1.4, o pacote `java.nio` (NIO significa Non-blocking I/O) abriu espaço para a comunidade implementar alguma solução reativa.

Em nosso contexto do Spring Framework, podemos entender que até a versão 4, a única opção existente era o Spring MVC, que oferece a solução com bloqueio (blocking), ou seja, qualquer processo/thread que use I/O, como uma chamada ao banco de dados, o sistema espera terminar a chamada para continuar o processamento de todo o sistema.

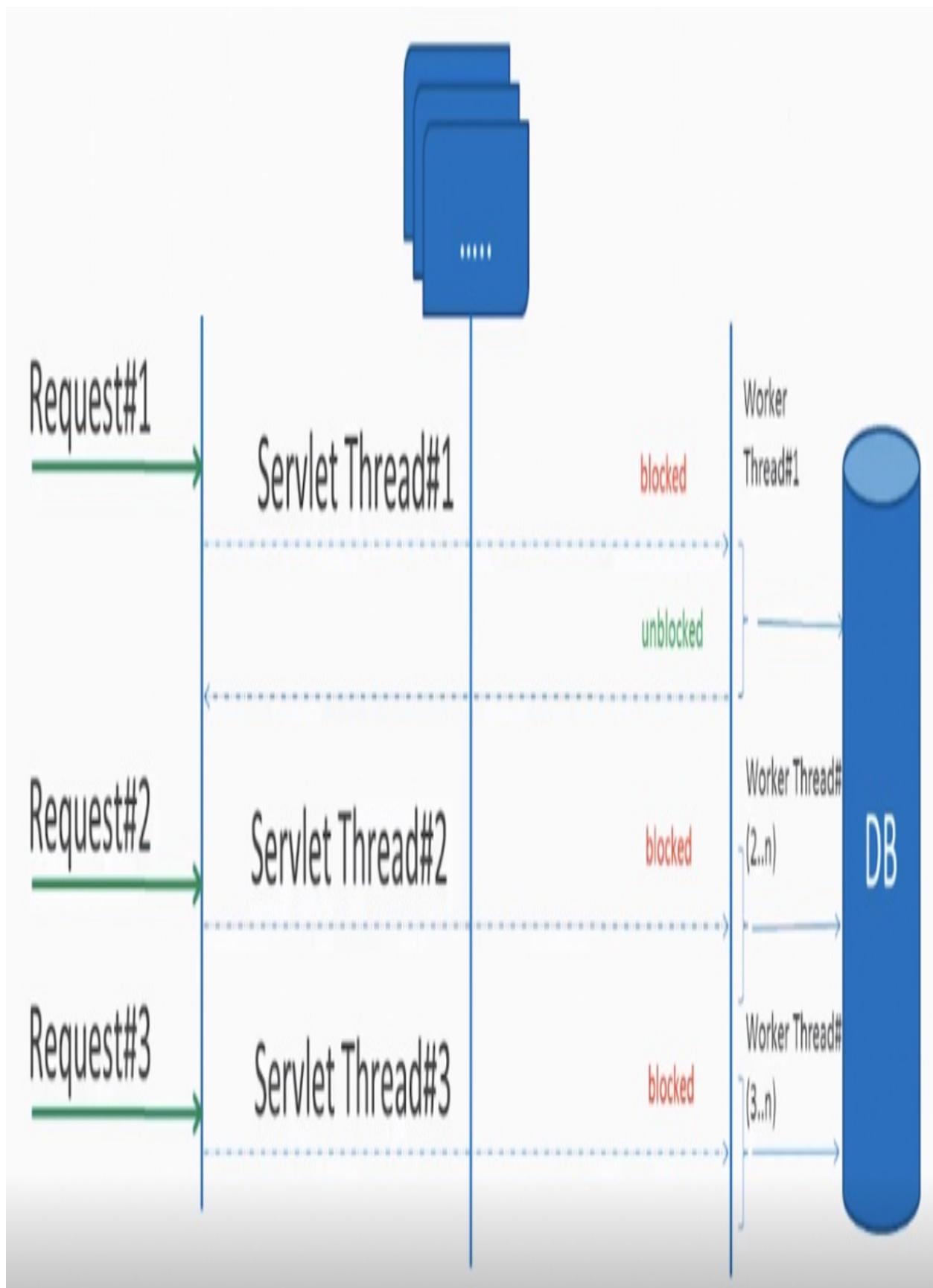


Figura 12.1: Acesso com I/O

Portanto, cada thread faz esse caminho:

O sistema solicita uma consulta ao banco de dados.

A JVM abre uma thread com uma chamada ao banco de dados enviando o SQL.

A thread espera o banco de dados responder.

O banco de dados responde com o resultado da consulta.

A thread pega a resposta e devolve para o sistema.

Cada thread então faz o seu trabalho e espera terminar. A solução reativa não faz isso, ela pega a thread e a coloca em um loop de eventos. Na implementação do Java NIO isso pode ser representado pelas classes:

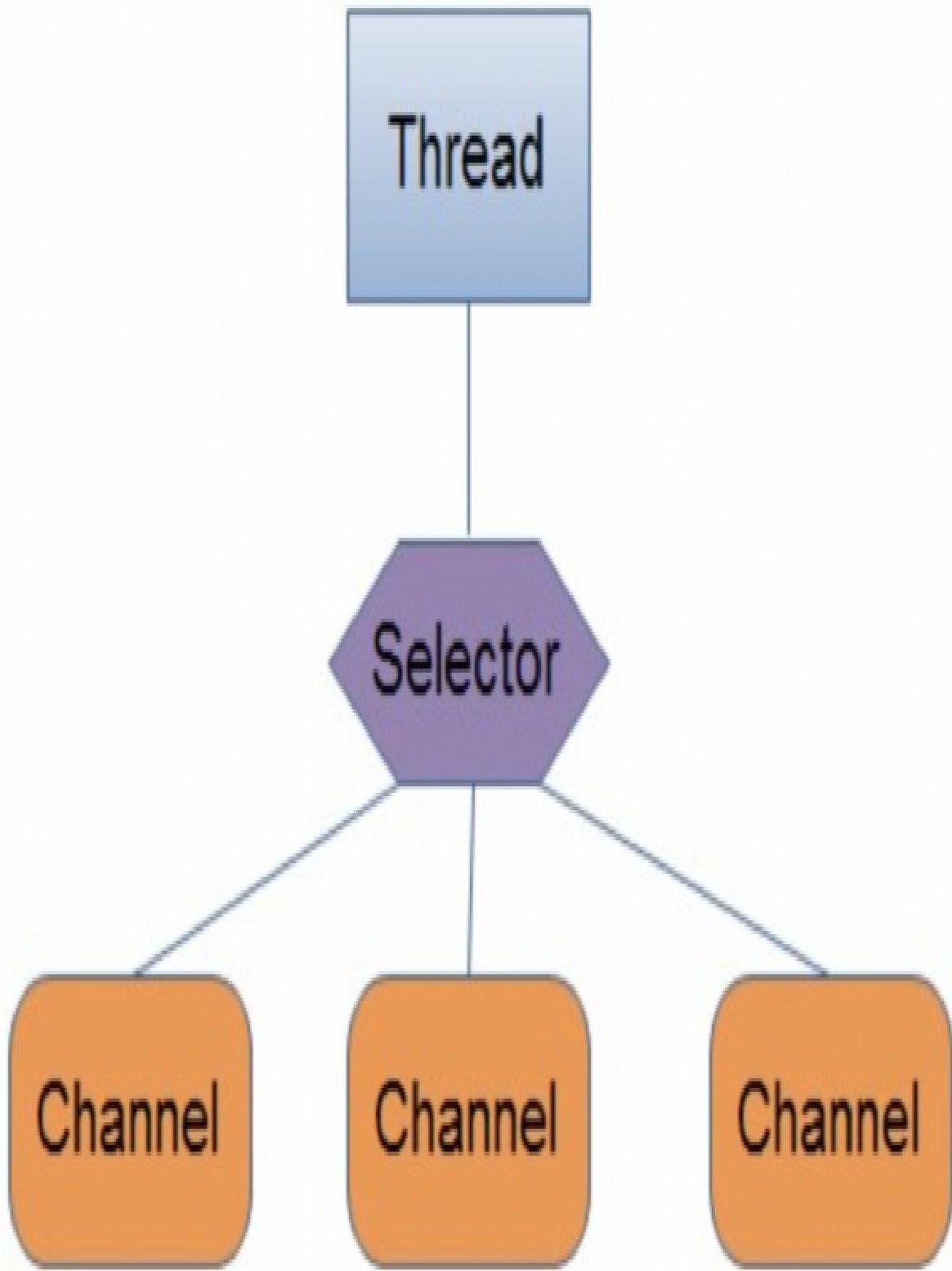


Figura 12.2: Thread Selector e Channel

Esse loop de eventos faz a chamada que a thread pediu e devolve depois por callback.

Em um sistema reativo, o mesmo exemplo faz esse caminho:

O sistema solicita consulta ao banco de dados.

A JVM cadastra um evento para solicitar a consulta ao banco de dados.

O Event Loop coloca o evento na fila dentro de um controle interno dele (número fixo de threads).

O Event Handler pega o evento do Event Loop para executar e repassa ao banco de dados.

O banco de dados responde com o resultado da consulta.

O Event Handler pega o resultado e coloca de volta no Event Loop.

O Event Loop pega o resultado e devolve para o sistema via callback.

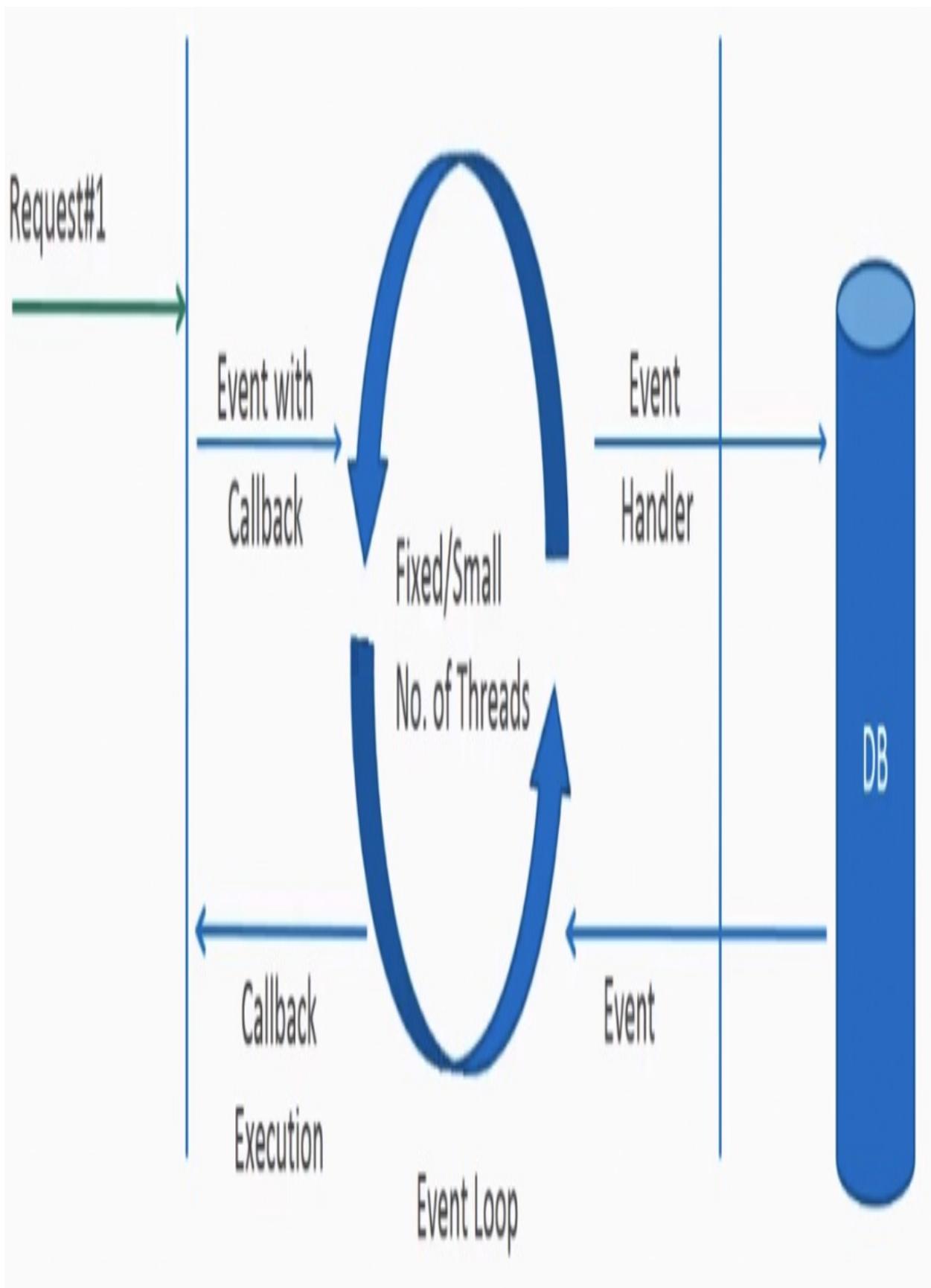


Figura 12.3: O caminho de um sistema reativo

A informação que atravessa esses canais não são os tipos primitivos que conhecemos do Java, são os tipos assíncronos Mono e Flux do projeto Reactor:

```
Mono<String> nenhumDado = Mono.empty();
```

```
Mono<String> livro = Mono.just(  
    "Spring Boot"  
)
```

```
Flux<Integer> numerosDe5ate7 = Flux.range(  
    5, 3);
```

Note que o Mono mantém sua declaração reativa genérica mesmo se for vazio, e que o range tem como primeiro parâmetro o início do intervalo e como segundo, a quantidade de números para produzir.

Com a versão 5 do Spring Framework, surgiram algumas opções reativas que podemos usar. Ela permite também que qualquer sistema seja híbrido: parte reativa com Spring Webflux e parte imperativa (não reativa) com Spring MVC.

Spring MVC

Imperative logic,
simple to write
and debug

JDBC, JPA,
blocking deps

Spring WebFlux

Functional endpoints
Event loop
concurrency model

Netty

@Controller

Reactive clients

Tomcat, Jetty,
Undertow

Figura 12.4: Acesso com NIO

Além do Spring Framework, os bancos de dados oferecem opções reativas, como o MongoDB ou o projeto R2DBC (<https://r2dbc.io>) que é uma implementação reativa do padrão JDBC.

Vale a pena ser reativo?

Essa questão de usar ou não uma solução reativa varia de acordo com o contexto. É uma análise de custo/benefício, pois trocamos uma solução que não tem bloqueio de I/O por algo assíncrono que não é tão simples de programar e debugar.

A Pivotal recomenda que nos projetos novos seja usado o Spring Webflux.

12.1 Ajustando o nosso projeto

Um projeto reativo precisa destas dependências:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-webflux</artifactId> </dependency>
```

E para usar o MongoDB reativo, esta dependência:

```
<dependency>
```

```
<groupId>org.mongodb</groupId>
```

```
<artifactId>mongodb-driver-reactivestreams</artifactId> </dependency>
```

A classe de configuração do MongoDB é necessária para informar que estamos usando a implementação reativa também:

```
@Configuration public class MongoReactiveConfig  
extends AbstractReactiveMongoConfiguration  
{  
  
@Bean  
  
public MongoClient mongoReactiveClient()  
{  
  
return  
MongoClients.create();  
}}
```

```
@Override
```

```
protected String getDatabaseName()  
{  
    return "test"  
};  
}
```

As classes de domain (Estoque e LogFila) são iguais no MongoDB sendo reativo ou não usando a notação Document, mas na classe EstoqueGreenDogDeliveryApplication é necessário declarar que o Spring Boot deve procurar por repositórios reativos:

```
@EnableReactiveMongoRepositories("com.boaglio")
```

Vamos adicionar ao controle de estoque uma consulta reativa com o seguinte repositório:

```
public interface ReactiveEstoqueRepository
```

```
extends ReactiveCrudRepository<Estoque, String>

{

    Flux<Estoque> findAll();

    ;

    Flux<Estoque> findTop10ByOrderByIdDesc()

    ;

}
```

E para fazer a chamada do repositório pela API dos últimos dez lançamentos do estoque:

```
@GetMapping("/ultimos") @ResponseBody public Flux<Estoque> ultimos()

{

    return

        reactiveEstoqueRepository.findTop10ByOrderByIdDesc();

}
```

E a chamada para a API retorna normalmente:

```
$ http GET :9000
```

```
/api/ultimos
```

```
[
```

```
{
```

```
"id": "5ff0e9c3e129ae4423d03d3f"
```

```
,
```

```
"itemId": 2
```

```
,
```

```
"quantidade": 1
```

```
},
```

```
{
```

```
"id": "5ff00e9f98aa26573fef9bb1"
```

```
,
```

"itemId": 3

,

"quantidade": 99

},

...

{

"id": "5fef6c6e98aa26573fef9bae"

,

"itemId": 3

,

"quantidade": 1

}

]

12.2 Sai RestTemplate, entra WebClient

Junto com o WebFlux veio WebClient (substituto do RestTemplate) para acessar endpoints reativos e não reativos.

Começamos declarando o WebClient com create:

```
WebClient client = WebClient.create("http://localhost:9000");
```

Depois fazemos um get, retrieve e convertemos o resultado para a variável ultimosFlux:

```
Flux<Estoque> ultimosFlux =
```

```
    client.get().uri(  
        "/api/ultimos"  
    )  
        .retrieve()  
        .bodyToFlux(Estoque.class);
```

E finalmente exibimos o conteúdo da variável:

```
ultimosFlux.subscribe(System.out::println);
```

Na saída do Spring Boot verificamos o resultado:

```
Estoque [itemId=2, quantidade=1
```

```
]
```

```
Estoque [itemId=
```

```
4, quantidade=1
```

```
]
```

```
Estoque [itemId=
```

```
3, quantidade=1
```

```
]
```

```
Estoque [itemId=
```

```
2, quantidade=1
```

```
]
```

```
Estoque [itemId=
```

```
1, quantidade=1
```

```
]
```

Estoque [itemId=2, quantidade=1]

Estoque [itemId=1, quantidade=1]

Estoque [itemId=1, quantidade=1]

Estoque [itemId=3, quantidade=99]

Estoque [itemId=3, quantidade=1]

12.3 Integração reativa

Vamos atualizar os sistemas para que a atualização do estoque use o modo reativo.

A primeira alteração é no back-end de controle de estoque, no qual criaremos uma nova API:

```
@PostMapping("/atualiza-reativo") public Mono<String>
atualizaReativo(@RequestBody Estoque estoque)

{
    System.out.println(
        "Recebido via REST: "
        +estoque);
    estoqueRepository.save(estoque);

    return Mono.just("Ok"
    );
}
```

No sistema de pedidos, vamos fazer a chamada com WebClient, inicialmente declarando o client e o estoque para atualizar:

```
WebClient client = WebClient.create("http://localhost:9000"
);
```

```
for
```

```
(Item item : pedido.getItens()) {
    Estoque estoque =
    new Estoque(item.getId(), 1l);
```

Em seguida, declaramos a chamada da API reativa via POST enviando o estoque e retornando uma String:

```
Mono<String> atualizaReativo =
    client.post().
        uri(
            "/api/atualiza-reativo"
        ).
        body(Mono.just(estoque), Estoque.class).
        retrieve().bodyToMono(String.class);
```

```
System.out.print(  
    "Resultado POST reativo = "  
);  
atualizaReativo.subscribe(System.out::println);
```

Ao testarmos a integração fazendo um novo pedido pelo sistema, na saída do Spring Boot verificamos o resultado:

Resultado POST reativo = Ok

12.4 API de stream de dados

Uma característica interessante de uma API reativa é a capacidade de fazer stream do retorno dos dados, ou seja, mandar o dado em partes de maneira contínua, semelhante ao streaming de som ou vídeo.

Nesse exemplo, criamos uma API que retorna toda a base de dados:

```
@GetMapping(path = "/lista-stream", produces = "application/stream+json")
public Flux<Estoque> getListaEstoqueStream()

{
    return
        reactiveEstoqueRepository.findAll();
}
```

Fazendo a chamada usando o cliente CURL (<https://curl.se>):

```
$ curl -v http://localhost:9000
/api/lista-stream
```

```
{  
    "id": "5ff08de5c00168744f97bb67", "itemId": 1, "quantidade": 1  
}  
...  
{  
    "id": "5ff3e385ce9ecc42b4b36bc0", "itemId": 1, "quantidade": 1}
```

Dessa maneira, os dados ainda vêm todos de uma vez, então precisamos colocar um intervalo de envio da informação. Vamos fazer isso, como exemplo, para trazer os últimos dez registros com um intervalo de 300 milissegundos.

```
@GetMapping(value = "/lista-stream-com-pausa", produces = "text/event-stream") public Flux<Estoque> getListaEstoqueStreamComPausa()  
{  
  
    return reactiveEstoqueRepository.findTop10ByIdDesc().  
        delayElements(Duration.ofMillis(  
            300  
        ));  
}
```

Fazendo o novo teste, verificamos que os dados aparecem com um pequeno intervalo, pouco a pouco até terminar:

```
$ curl -v http://localhost:9000  
/api/lista-stream-com-pausa  
data:{  
"id":"5ff3e385ce9ecc42b4b36bc0","itemId":1,"quantidade":1  
}  
...  
data:{  
"id":"5ff08df2c00168744f97bb68","itemId":1,"quantidade":1}
```

12.5 O que aprendemos

Para acompanhar o projeto completo até aqui, acesse o branch reactive em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/reactive>.

Certifique-se de que você aprendeu:

- a usar os conceitos reativos básicos;
- os conceitos do WebFlux e WebClient;
- a trabalhar com variáveis Mono e Flux;
- a integrar sistemas diferentes com WebClient;
- a trabalhar com APIs reativas.

No próximo capítulo, veremos como aumentar a segurança da nossa aplicação.

Capítulo 13

Segurança

Segurança com Spring Framework

O comportamento comum de quem nunca usou o Spring Security é tentar usar uma vez e constatar:

Parece complexo demais.

Precisa declarar muita coisa.

Certamente se eu implementar o meu próprio esquema de segurança será bem mais fácil.

E então começa a aventura de trabalhar com Servlet Filters e diferentes autenticações de diferentes escopos e, ao perceber que não é tão fácil assim, o normal é voltar ao Spring Security e usar suas configurações.

Antes de avançar, precisamos entender a diferença entre autenticação e autorização.

Autenticação define quem pode entrar no sistema, autorização define o que pode ser feito no sistema.

Existem diferentes tipos de autenticação no Spring Security, mas vamos mostrar aqui o mais simples.

Para algo mais complexo, foi criado pelo Twitter, e largamente adotado pelo mercado, o padrão de autorização OAuth2. Sobre isso, sugiro um excelente livro que tem muitos exemplos com Spring e explica com detalhes várias abordagens: OAuth 2.0: Proteja suas aplicações com o Spring Security OAuth2 (<https://www.casadocodigo.com.br/products/livro-oauth>).

Protegendo o nosso sistema

A atualização de estoque continua sendo feita pela matriz e pela filial frequentemente, entretanto as consultas ao estoque, que deveriam ser restritas ao Rodrigo, são públicas, qualquer pessoa na internet pode acessá-lo.

Temos, portanto, a necessidade de restringir o acesso nessas APIs de consulta (/api/lista e /api/ultimos) e criar um usuário para que o Rodrigo possa acessar o estoque com segurança.

13.1 Usando Spring Security

Começamos colocando as dependências do Spring Security e das tags de segurança do Thymeleaf:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-security</artifactId> </dependency>
<dependency>
```

```
<groupId>org.springframework.security</groupId>
```

```
<artifactId>spring-security-test</artifactId>
```

```
<scope>test</scope> </dependency> <dependency>
```

```
<groupId>org.thymeleaf.extras</groupId>
```

```
<artifactId>thymeleaf-extras-springsecurity5</artifactId> </dependency>
```

Depois declaramos a principal classe que resume todas as configurações desejadas de segurança:

```
@Configuration @EnableWebSecurity public class WebSecurityConfig extends  
WebSecurityConfigurerAdapter {
```

Em seguida, sobrescrevemos o método de configuração de segurança, permitindo o acesso a várias páginas (home, login, logout, /api/logs e arquivos estáticos) e forçando a autenticação em qualquer outra página:

```
@Override protected void configure(HttpSecurity http) throws Exception  
{
```

```
http.authorizeRequests()

// libera arquivos em "/css","/js","/images","/webjars" e "favicon.ico"

.requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()

// libera páginas home e API logs

.antMatchers(
  "/", "/home","/api/logs"
).permitAll()

// todo o resto pede autenticação

.anyRequest().authenticated().and()

// libera página de login

.formLogin().loginPage(
  "/login"
).permitAll().and()
```

```
// libera página de logout
```

```
.logout().permitAll();  
}
```

Aqui definimos o serviço de usuários que podem entrar no sistema, escolhendo a abordagem mais simples: criaremos um usuário ("fernando") com um único perfil ("USER") e usaremos a implementação de armazenar esse usuário na memória.

```
@Bean @Override public UserDetailsService userDetailsService()
```

```
{
```

```
UserDetails user =
```

```
User.withUsername(
```

```
"fernando"
```

```
)
```

```
.password(
```

```
"boaglio123"
```

```
)
```

```
.roles(
```

```
"USER"

)

.build();

return new InMemoryUserDetailsManager(user)

;

}
```

Agora no front-end, vamos editar a página home.html adicionando o código do Thymeleaf. Se o usuário não estiver autenticado, será exibido um botão "entrar".

```
<div sec:authorize="isAuthenticated()" class="d-flex justify-content-around">

<h1 th:inline="text">Olá [[${#httpServletRequest.remoteUser}]]!</h1>

<form th:action="@{/logout}" method="post">

<input type="submit" value="sair" class="btn btn-warning"/>
```

```
</form>
```

```
</div> <div sec:authorize="!isAuthenticated()" class="d-flex justify-content-around">
```

```
<form th:action="@{/login}" method="get">
```

```
<input type="submit" value="entrar" class="btn btn-primary"/>
```

```
</form>    </div>
```

spring® Controle de estoque



Logs

Logs da API

Logs

Consultas

Consulta ao estoque

Completo Últimos

Figura 13.1: Página home

O botão entrar redireciona para a página de login. Ao chamar uma URL não autorizada, como /teste123, o sistema também redireciona para a mesma página de login.

Área Restrita

Usuário

fernando



Senha



entrar

Figura 13.2: Página de login

Depois de entrar no sistema, é exibido o usuário logado junto com o botão de sair:

spring® Controle de estoque

Olá fernando!

→  sair

Logs

Logs da API

Logs

Consultas

Consulta ao estoque

Completo Últimos

Figura 13.3: Página home logada

A navegação web está segura, mas se tentarmos acessar diretamente a API, verificamos outro comportamento:

```
$ curl -v http://localhost:9000
```

```
/api/ultimos
```

```
...
```

```
HTTP/
```

```
1.1 302
```

```
$ http :
```

```
9000
```

```
/api/ultimos
```

```
HTTP/
```

```
1.1 302
```

```
...
```

```
Location: http://localhost:
```

```
9000/login
```

O HTTP status de 302 significa que a página foi encontrada, mas ela será redirecionada para outra (definida em Location).

A solução do nosso teste no caso de APIs é mudar a autenticação para HTTP Basic (lembrando que a solução mais robusta para autenticação de APIs é OAuth2).

A mudança do tipo de autenticação é feita na classe WebSecurityConfig, trocando o formLogin por httpBasic:

```
@Override protected void configure(HttpSecurity http) throws Exception
{
    http.authorizeRequests()
        .requestMatchers()
            (PathRequest.toStaticResources().atCommonLocations()).permitAll()
        .antMatchers(
            "/", "/home", "/api/logs"
        ).permitAll()
        .anyRequest().authenticated().and()
        .httpBasic();
```

Agora, repetindo a chamada o retorno muda:

```
$ curl http://localhost:9000
```

```
/api/ultimos
```

```
$ http :
```

```
9000
```

```
/api/ultimos
```

```
{
```

```
  "error": "Unauthorized"
```

```
,
```

```
  "message": "Unauthorized"
```

```
,
```

```
  "path": "/api/ultimos"
```

```
,
```

```
  "status": 401
```

```
}
```

Precisamos repetir a chamada de maneira autenticada:

```
$ curl -u fernando:boaglio123 http://localhost:9000
```

```
/api/ultimos
```

```
$ http --auth fernando:boaglio123 :
```

```
9000
```

```
/api/ultimos
```

```
[
```

```
{
```

```
    "id": "5ff3e385ce9ecc42b4b36bc0"
```

```
,
```

```
    "itemId": 1
```

```
,
```

```
    "quantidade": 1
```

```
},
```

```
...
```

```
{
```

```
    "id": "5ff08df2c00168744f97bb68"
```

,

"itemId": 1

,

"quantidade": 1

}

]

A página web continua protegida, mas em vez da elegante página de login, aparece uma janela modal padrão do navegador solicitando as credenciais:

ⓘ 127.0.0.1:9000/api/ultimos



Fazer login

<http://127.0.0.1:9000>

Nome de usuário

Senha

[Cancelar](#)

[Fazer login](#)

Figura 13.4: Página protegida

13.2 Protegendo senhas do código

Para o nosso sistema, a proteção HTTP Basic com apenas um usuário é suficiente, entretanto qualquer um que tiver acesso ao código-fonte consegue ler as informações de usuário e senha.

Para evitar esse problema, o Spring nos fornece a classe PasswordEncoder para codificar senhas. Em outro projeto, podemos fazer uma classe temporária para gerar a senha:

```
public static void main(String[] args)
{
    String senhaAdmin =
    "boaglio123"
;

PasswordEncoder encoder = PasswordEncoderFactories.
createDelegatingPasswordEncoder();

System.out.println(
    "senha = "
    + encoder.encode(senhaAdmin));
```

```
}
```

Executando o programa, vemos o resultado:

```
senha =  
{bcrypt}$2a$10$N/JkyAmIDX70am/U3PPP7uiWuRHH9VklzpjKP9ugAe2t6tAr
```

E podemos substituir no método userDetailsService a senha criptografada:

```
public UserDetails userDetailsService() {
```

```
    UserDetails user =  
        User.withUsername("fernando")  
            .password("{bcrypt}
```

```
$2a$10$N/JkyAmIDX70am/U3PPP7uiWuRHH9VklzpjKP9ugAe2t6tAnNWLjq'
```

```
            .roles("USER")  
            .build();
```

```
    return new InMemoryUserDetailsManager(user);
```

```
}
```

13.3 O que aprendemos

Para acompanhar o projeto completo até aqui, acesse o branch reactive em <https://github.com/boaglio/spring-boot-greendogdelivery-estoque-casadocodigo/tree/seguro>.

Certifique-se de que você aprendeu:

- os conceitos básicos do Spring Security;
- a proteger uma API e uma página web;
- a utilizar autenticação por HTTP basic e por login;
- a armazenar senhas do código de maneira segura.

No próximo capítulo, veremos como disponibilizar a nossa aplicação em diferentes ambientes com diferentes servidores.

Capítulo 14

Empacotando e disponibilizando sua aplicação

Depois de validar todos os testes, Rodrigo está confiante em subir sua aplicação para o servidor e começar a usá-la em produção. Neste cenário, existe a dúvida de qual opção usar. Vamos entender as opções existentes.

14.1 JAR simples

O jeito padrão do Spring Boot é disponibilizar o sistema inteiro (servidor e aplicação) dentro de um pacote JAR. É a alternativa mais simples possível.

Para obter o JAR, basta executar o comando:

```
$ mvn install  
$ java -jar target/green-dog-delivery-  
2.4.0-SNAPSHOT.jar
```

E o seu JAR está pronto para uso e pode ser publicado no servidor de produção.

14.2 JAR executável

Usamos manualmente o comando `java -jar` para subir o sistema. Uma desvantagem é que, se acontecer algum reboot do servidor, o sistema ficará fora do ar.

Para solucionar esse problema, é sugerido usar o sistema como serviço. Com isso, se a máquina reiniciar, o sistema operacional automaticamente sobe o serviço e, consequentemente, o sistema.

Para usar o JAR como serviço, é preciso informar no arquivo `pom.xml` esta opção:

```
<plugin>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-maven-plugin</artifactId>
```

```
<configuration>
```

```
<executable>true</executable>
```

```
</configuration> </plugin>
```

Ao gerar o pacote, ele pode ser usado como serviço, por exemplo, em servidores Linux atuais que são gerenciados pelo Systemd (Ubuntu 15.4 ou superior). É só renomear o JAR gerado do projeto para gdd.jar e mover para o diretório /var/run/springboot. Em seguida, no diretório /etc/systemd/system, adicionamos o arquivo gdd.service:

```
[Unit]
```

```
Description=gdd
```

```
After=syslog.target
```

```
[Service]
```

```
User=root
```

```
ExecStart=/var/run/springboot/gdd.jar
```

```
SuccessExitStatus=
```

[Install]

WantedBy=multi-user.target

Com isso, podemos ativar o serviço com:

\$ systemctl enable gdd.service

Derrubar o serviço com:

\$ systemctl stop gdd.service

E subir o serviço com:

\$ systemctl start gdd.service

Veja um exemplo de como exibir o status do serviço:

```
$ systemctl status gdd.service
```

- gdd.service - gdd

 Loaded: loaded (/etc/systemd/system/gdd.service;

 Active: active (running)

 Main PID: 11880

 (gdd.jar)

 Tasks: 34 (limit: 4915

)

 Memory:

141.4

 M

 CPU:

44.370

 s

 CGroup: /system.slice/gdd.service

 └─

11880

 /bin/bash /var/run/springboot/gdd.jar

 └─

```
11911 /usr/sbin/java -Dsun.misc.URLClassPath
```

Para as distribuições mais antigas, baseadas no System V, a instalação é mais simples. Basta criar um link simbólico:

```
$ sudo ln -s /var/run/springboot/gdd.jar /etc/init.d/gdd
```

Com isso, podemos derrubar o serviço com:

```
$ service gdd stop
```

E subir o serviço com:

```
$ service gdd start
```

Para customizar parâmetros da VM do serviço criado, usamos um arquivo no mesmo lugar do JAR com a extensão .conf e editamos o arquivo /var/run/springboot/gdd.conf, adicionando a seguinte linha:

JAVA_OPTS=-Xmx1024M

Ao reiniciar o serviço, as novas configurações serão aplicadas.

Veja também outras maneiras de iniciar uma aplicação Spring Boot no Apêndice B — Propriedades.

Outros sistemas operacionais

Não existe um suporte oficial para Windows, mas existe a alternativa de usar o Windows Service Wrapper (<https://github.com/kohsuke/winsw>). Veja mais em: <https://github.com/snical-scratches/spring-boot-daemon>.

Para Mac OS X, também não existe suporte. Entretanto, há uma alternativa interessante que é o Launchd (<http://launchd.info>).

14.3 WAR

Em alguns ambientes de produção mais conservadores, não existe a opção de subir um JAR simples; é preciso subir um pacote WAR, de acordo com o padrão Java EE. Para esses casos, existe a opção de gerar o WAR. O Spring Boot automaticamente colocará todas as bibliotecas necessárias dentro desse pacote.

Para gerar um pacote WAR, basta alterar no pom.xml:

```
<packaging>war</packaging>
```

E depois gerar o pacote com Maven:

```
$ mvn install
```

```
$ du -h target/*.war
```

```
54M  target/green-dog-delivery-2.4.0-SNAPSHOT.war
```

Com isso, o pacote WAR pode ser publicado em um Tomcat, Jetty, JBoss ou WildFly (Undertow) sem nenhum problema.

14.4 Tomcat/Jetty/Undertow

Os diferentes contêineres de aplicação existentes possuem vantagens e desvantagens, e foge do escopo deste livro discutir cada uma delas. O que é importante saber é de que maneira podemos alterar o contêiner usado.

A implementação de contêiner web padrão é o Tomcat, mas podemos mudar para o Jetty. Veja:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-web</artifactId>
```

```
<exclusions>
```

```
<exclusion>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-tomcat</artifactId>
```

```
</exclusion>
```

```
</exclusions> </dependency> <dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-jetty</artifactId> </dependency>
```

O resto continua a mesma coisa, subindo com o Maven:

```
o.s.b.a.e.EndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'  
o.e.j.s.h.ContextHandler.application : Initializing Spring DispatcherServlet 'dispatcherServlet'  
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'  
o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms  
o.e.jetty.server.AbstractConnector : jetty-7.0.5.v20110208  
o.s.b.web.embedded.jetty.JettyWebServer : Jetty started on port(s) 8080 [http/1.1] with context  
c.b.c.g.GreenLogDeliveryApplication : Application successfully initialized in 5,888 seconds
```

Figura 14.1: Subindo com o servidor Jetty

Da mesma maneira, podemos trocar pelo Undertow:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-web</artifactId>
```

```
<exclusions>
```

```
<exclusion>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions> </dependency> <dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-undertow</artifactId> </dependency>
```

```
o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'  
io.undertow : starting server: Undertow - 2.2.3.Final  
org.xnio : XNIO version 3.8.0.Final  
org.xnio.nio : XNIO NIO Implementation Version 3.8.0.Final  
org.jboss.threads : JBoss Threads version 2.1.0.Final  
o.s.b.a.undertow.UndertowServer : Undertow started on port(s) 8080  
com.craig GreenDogDeliveryApplication : Started GreenDogDeliveryApplication in 5.935 seconds
```

Figura 14.2: Subindo com o servidor Undertow

Existem dezenas de configurações de contêiner (todas começam com server.) para colocar no arquivo application.properties. Algumas são específicas para Tomcat, Jetty ou Undertow, e outras são genéricas, como essa:

```
server.compression.enabled=true
```

Todas as configurações genéricas serão aplicadas ao contêiner escolhido para rodar. Se por acaso existir uma configuração específica para Tomcat e o sistema subir com o Jetty, a configuração é simplesmente ignorada e o sistema sobe sem problemas.

14.5 Spring Boot CLI

O Spring Command Line Interface, ou Spring CLI, é uma maneira de fazer rápidos protótipos com Spring Boot. Para fazer um teste rápido de uma tela, não é preciso fazer um sistema inteiro, podemos facilmente criar uma classe em Groovy e subi-la com o Spring CLI.

A instalação dele é bem simples. É só baixar o arquivo -bin.zip do site <https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/> e descompactar em algum diretório.

Teste simples

Vamos fazer um protótipo simples de teste para exibir as propriedades do sistema. Para isso, criaremos um arquivo teste.groovy com o conteúdo:

```
@RestController class Teste  
{  
    @GetMapping("/") Properties properties()  
}
```

```
return  
System.getProperties();  
}  
}
```

Quando executarmos o comando:

```
$ spring run teste.groovy
```

Teremos facilmente a página pronta para testes no browser.

← → C

ⓘ localhost:8080

{

```
awt.toolkit: "sun.awt.X11.XToolkit",
java.specification.version: "11",
sun.cpu.isalist: "",
sun.jnu.encoding: "UTF-8",
java.class.path: "/tmp/spring-2.4.1/bin:/tmp/s
java.vm.vendor: "AdoptOpenJDK",
sun.arch.data.model: "64",
java.vendor.url: "https://adoptopenjdk.net/",
catalina.useNaming: "false",
user.timezone: "America/Sao_Paulo",
os.name: "Linux",
java.vm.specification.version: "11",
sun.java.launcher: "SUN_STANDARD",
user.country: "BR",
```

Figura 14.3: Teste com Spring CLI

14.6 O que aprendemos

Certifique-se de que você aprendeu:

- a fazer um deploy da aplicação como pacote JAR;
- a fazer um deploy da aplicação como serviço;
- a fazer um deploy da aplicação como pacote WAR;
- a fazer um deploy de protótipos simples com Spring CLI.

No próximo capítulo, vamos ver como usar o Spring Boot na nuvem.

Capítulo 15

Subindo na nuvem

Agora que Rodrigo entendeu as diversas maneiras de se fazer deploy de um sistema, ele precisa colocar isso na nuvem. Quando usamos ambientes diferentes, temos um problema comum: os servidores mudam, usuários e senhas também. Para gerenciar essas mudanças, usamos o esquema de perfil (profile) do Spring Boot.

Profiles

Os profiles (perfis) são usados para a aplicação rodar com uma configuração diferenciada. Isso é muito útil para rodar o sistema em bancos de dados diferentes ou para definir algum comportamento no sistema que só é ativado em produção.

Envio de e-mails

O envio de e-mails é um bom exemplo de uma funcionalidade que deve existir apenas em produção. Porém, como diferenciar isso no sistema?

Vamos, inicialmente, implementar uma notificação para confirmar um pedido realizado no nosso sistema de delivery. A rotina de envio será uma interface:

```
package  
com.boaglio.casadocodigo.greendogdelivery.dto;  
  
public interface Notificacao  
{  
  
    boolean envioAtivo()  
;  
}
```

Uma classe utilitária de envio de e-mail fará um teste com o método envioAtivo para cada pedido e, apenas em caso afirmativo, fará o envio da notificação ao cliente.

```
@Component public class EnviaNotificacao  
{  
  
    @Autowired  
    Notificacao notificacao;
```

```
public void enviaEmail(Cliente cliente, Pedido pedido)  
{  
    if  
        (notificacao.envioAtivo()) {  
  
        /* codigo de envio */  
  
        System.out.println(  
            "Notificacao enviada!"  
        );  
    }  
  
    else  
    {  
        System.out.println(  
            "Notificacao desligada!"  
        );  
    }  
}  
}
```

Vamos alterar a nossa classe de NovoPedidoController para enviar a notificação

após um novo pedido:

this

```
.clienteRepository.saveAndFlush(c);  
enviaNotificacao.enviaEmail(c,pedido);
```

Criaremos uma implementação da interface de produção chamada ProdNotificacaoConfig, que retorna true (pois o envio de e-mails deve funcionar apenas em produção):

```
@Component @Profile("prod") public class ProdNotificacaoConfig implements  
Notificacao
```

```
{
```

```
@Override
```

```
public boolean envioAtivo()
```

```
{
```

```
return true
```

```
;  
}
```

```
}
```

E outra implementação de desenvolvimento chamada DevNotificacaoConfig, que retorna false (já que o envio de e-mails não pode funcionar em desenvolvimento):

```
@Component @Profile("!prod") public class DevNotificacaoConfig implements  
Notificacao
```

```
{
```

```
@Override
```

```
public boolean envioAtivo()
```

```
{
```

```
return false
```

```
;  
}  
  
}
```

A classe ProdNotificacaoConfig possui a anotação `@Profile("prod")`, indicando que ela será instanciada apenas quando esse perfil estiver ativo. Já a classe DevNotificacaoConfig possui a anotação `@Profile("!prod")`, indicando que ela será instanciada quando o perfil ativo for diferente de prod.

Para definirmos um perfil padrão, podemos colocar no `application.properties`:

```
# profile  
  
spring.profiles.active=dev
```

Ao fazer o pedido, observamos nos logs que a mensagem não foi enviada (como era esperado):

...

```
Hibernate: insert into pedido_itens (pedido_id, itens_id)  
values (?, ?)
```

Hibernate: insert into pedido_itens (pedido_id, itens_id)
values (?, ?)

Enviar notificacao para Fernando Boaglio - pedido
\$52.0

Notificacao desligada!

...

Nas variáveis de ambiente, também é possível visualizar o ambiente usado:

← → ⌂ ⓘ localhost:8080/actuator/env

```
{  
  activeProfiles: [  
    "dev"  
  ],  
  propertySources: [  
    {  
      name: "server.ports",  
      value: "8080"  
    }  
  ]  
}
```

Figura 15.1: Ambiente utilizado profile de dev

Para testar o perfil prod e sobrescrever o valor definido em application.properties, use a opção de linha de comando:

```
$ java -jar target/green-dog-delivery-2.4.0  
-SNAPSHOT.jar  
--spring.profiles.active=prod
```

Veja que agora o log de pedido mudou:

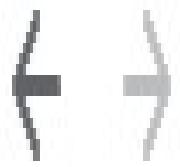
```
Hibernate: insert into pedido_itens (pedido_id, itens_id)  
values (?, ?)
```

```
Hibernate: insert into pedido_itens (pedido_id, itens_id)  
values (?, ?)
```

```
Enviar notificacao para Fernando Boaglio - pedido  
$55.0
```

Notificacao enviada!

Nas variáveis de ambiente, vemos o ambiente utilizado:



localhost:8080/actuator/env

```
{  
  "activeProfiles": [  
    "prod"  
  ],  
  "propertySources": [  
    {  
      "name": "server.ports"  
    }  
  ]  
}
```

Figura 15.2: Ambiente utilizado profile de prod

Aqui temos uma poderosa ferramenta para manipular os sistemas com Spring Boot. Sem nenhuma alteração no código-fonte, apenas passando um parâmetro definindo o profile, conseguimos mudar completamente o comportamento do sistema.

Usamos um exemplo simples de envio de e-mails apenas em produção. Mas isso pode se estender a outros níveis, como profile de diferentes bancos de dados ou diferentes servidores.

Usando profiles para o Heroku

Heroku é uma empresa prestadora de serviço de hospedagem na nuvem (Platform-as-a-Service ou PaaS) que suporta várias linguagens de programação. Existem outras opções no mercado, mas a maioria delas é paga.

Além de executar localmente, a nossa aplicação vai rodar no Heroku, um ambiente diferente, e por esse motivo será necessário criar um perfil exclusivo.

Cada perfil novo do Spring Boot é um arquivo properties novo. Vamos criar um em nosso exemplo para usar o banco de dados MySQL local e outro da nuvem.

É bem comum o banco de dados local ser diferente do servidor de nuvem, pois as informações de conexão certamente não são iguais. Portanto, o profile ajuda a aplicação a se ajustar conforme a necessidade.

Vamos criar o arquivo para trabalhar localmente, o application-mysql.properties:

```
# jpa
```

```
spring.jpa.show-sql=
```

```
true
```

```
spring.datasource.url=jdbc:mysql://localhost:
```

```
3306
```

```
/greendogdelivery
```

```
spring.datasource.username=greendogdelivery
```

```
spring.datasource.password=greendogdelivery
```

```
spring.jpa.hibernate.ddl-auto=create-drop
```

```
# rest
```

```
spring.data.rest.base-path=/api
```

```
# template
```

```
spring.thymeleaf.cache =  
false # Hypermedia As The Engine Of Application State
```

```
spring.hateoas.use-hal-as-default-json-media-type=  
false # Actuator
```

```
management.endpoints.web.exposure.include=*<br/>  
# health
```

```
management.endpoint.health.show-details=always<br/>  
# http trace
```

```
management.trace.http.enabled=  
true
```

Criaremos também o arquivo para trabalhar na nuvem application-heroku.properties:

```
# jpa
```

```
spring.jpa.show-sql=  
true
```

```
spring.datasource.url=  
${CLEARDB_DATABASE_URL}
```

```
spring.jpa.hibernate.ddl-auto=create-drop  
#rest
```

```
spring.data.rest.base-path=/api  
# template
```

```
spring.thymeleaf.cache =  
false # Hypermedia As The Engine Of Application State
```

```
spring.hateoas.use-hal-as-default-json-media-type=  
false # Actuator
```

```
management.endpoints.web.exposure.include=*  
# health
```

```
management.endpoint.health.show-details=always
```

```
# http trace
```

```
management.trace.http.enabled=
```

```
true
```

Como esperado, os parâmetros de conexão ao banco de dados (spring.datasource.url) são diferentes nos arquivos properties.

Para subir o perfil MySQL, basta chamar na linha de comando:

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

```
-Drun.arguments=
```

```
--spring.profiles.active=mysql"
```

Dentro do Eclipse, é possível escolher o perfil na opção Run e Debug Configurations:

Name: spring-boot-greendogdelivery-casadocodigo - GreenDogDeliveryApplication

Spring Boot

Arguments

JRE

Classpath

Source

Environment

Common

Project

spring-boot-greendogdelivery-casadocodigo

Main type

com.boaglio.casadocodigo.greendogdelivery.GreenDogDeliveryApplication

Search...

Profile

mysql

Enable debug output

Hide from Boot Dash

Enable ANSI console output

Enable JMX Port: 0

Enable Live Bean support.

Enable Life Cycle Management. Termination timeout (ms): 15000

Override properties:

property

value

Figura 15.3: Escolher perfil no Eclipse

15.1 Publicando o sistema no Heroku

Vamos usar o Heroku que oferece uma opção gratuita bem simples de usar.

Depois de criarmos uma conta gratuita no Heroku (<https://www.heroku.com>), podemos criar uma nova aplicação green-dog-delivery, ou podemos utilizar qualquer outro nome que desejarmos.



Jump to Favorites, Apps, Pipelines, Spaces...

Create New App

App Name (optional)

Leave blank and we'll choose one for you.

green-dog-delivery



green-dog-delivery is available

Runtime Selection



United States



Your app can run in your choice of region in
the Common Runtime.

Create App

Figura 15.4: Nova aplicação

Em seguida, adicionamos um recurso (resource) de banco de dados MySQL, pois o nosso sistema precisa armazenar os dados em algum lugar.



HEROKU

Jump to Favorites, Apps, Pipelines, S



Personal apps >



green-dog-delivery

Overview

Resources

Deploy

Metrics

Activity

Access

Settings

Dynos

This app has no processes

Add a Procfile to your app in order to define

Add-ons

mysql



ClearDB MySQL



JawsDB MySQL

Figura 15.5: Novo recurso de banco de dados

E escolhemos a opção gratuita:



ClearDB MySQL



green-dog-delivery

Plan name

Ignite – Free



[View add-on details in Elements Marketplace](#)

Provision

Figura 15.6: Opção gratuita do MySQL

Em seguida, obtemos as informações de acesso remoto ao banco de dados na opção Settings dentro da variável CLEARDB_DATABASE_URL.



HEROKU

Jump to Favorites, Apps, Pipelines, Spaces...



Personal apps



green-dog-delivery



Open

Overview

Resources

Deploy

Metrics

Activity

Access

Settings

Name

green-dog-delivery



Config Variables

Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.

Config Vars

CLEARDB_DATABASE_URL

east-03.cleardb.net/heroku_94b4dbe

Figura 15.7: Configurações da aplicação

As informações vêm no formato: mysql://usuário:senha@servidor/database?reconnect=true. Dessa maneira, faremos uma carga inicial das tabelas do MySQL em linha de comando:

```
mysql -u usuário -h servidor database -p < script-sql
```

Veja um exemplo:

```
$ mysql -u b8bc44c -h us.cleardb.net heroku_94b3 -p < gdd.sql
```

Enter password:

```
$
```

Em seguida, subimos os fontes no repositório Git do Heroku. Existe a opção de ligar o projeto ao GitHub ou ao Dropbox se os fontes estiverem lá. Entretanto, não utilizaremos os fontes de nenhum repositório remoto, e sim do nosso diretório local da máquina.

Vamos criar um exemplo no qual os fontes do Heroku estão no diretório heroku, e o projeto spring-boot-greendogdelivery-casadocodigo foi copiado para dentro dele.

```
$ cd  
heroku  
$  
cd  
spring-boot-greendogdelivery-casadocodigo  
$ rm -rf .git  
$ git init
```

Aqui precisamos criar o arquivo Profile na raiz do projeto, já que ele é o ponto de partida que o Heroku vai usar logo depois de baixar e compilar os seus fontes. Nele especificaremos que o nosso sistema é uma aplicação JAR e que usará o profile heroku:

```
web java -Dserver.port=$PORT $JAVA_OPTS  
-jar  
target/green-dog-delivery-  
2.4.0  
-SNAPSHOT.jar  
-Dspring.profiles.active=heroku
```

Também precisamos informar o Heroku da versão do Java utilizada e fazemos

isso dentro do arquivo system.properties:

```
java.runtime.version=11
```

Em seguida, vamos adicionar o código-fonte do sistema ao repositório git, commitar e fazer um push (enviar) ao servidor do Heroku.

```
$ git add .
```

```
$ git commit -am
```

"teste na nuvem"

```
$ git push heroku master
```

Ao fazer esses comandos do Git, os nossos arquivos fontes são enviados ao repositório remoto.

Uma opção interessante para acompanhar os processos e logs da aplicação na nuvem é usar o Heroku CLI (antigo Heroku Toolbelt), que existe para diversas plataformas. Acesse <https://devcenter.heroku.com/articles/heroku-cli>.

```
$ heroku ps -a
```

```
green-dog-delivery
```

==== web (Free): java -Dserver.port=

\$PORT \$JAVA_OPTS

-jar

target/green-dog-delivery-

2.4.0

-SNAPSHOT.jar

-Dspring.profiles.active=heroku (

1

)

web.

1

: up

\$ heroku logs

-a green-dog-delivery

Com o deploy no ar, podemos testar no browser:



Seguro | <https://green-dog-delivery.herokuapp.com>



Casa do Código

boaglio.com

Spring Boot

Thymeleaf Layout

Green Dog Delivery Painel de controle

Ambiente

H2 Console

HAL Browser

Cadastro de clientes

Cadastro de itens

Cadastro de pedidos

Delivery

Figura 15.8: Sistema rodando na nuvem

Podemos acompanhar também pelo site da Heroku como está o nosso sistema. É só logar e entrar nos detalhes da aplicação criada:



Personal apps



green-dog-delivery

Overview

Resources

Deploy

Metrics

Activity

Access

Settings

Activity Feed



boaglio@gmail.com: Deployed 5fba8ba

about 10 hours ago • v4



boaglio@gmail.com: Build succeeded

about 10 hours ago • [View build log](#)

boaglio@gmail.com: Attach CLEARDB_DATABASE (@ref:cleardb-tapered-25134)

about 11 hours ago • v3 • Roll back to here

Figura 15.9: Atividades do sistema

15.2 O que aprendemos

Os fontes do projeto estão em <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/heroku>.

Certifique-se de que você aprendeu:

- a trabalhar com profiles (perfis) no Spring Boot;
- a subir uma aplicação na nuvem da Heroku.

No próximo capítulo, veremos um pouco sobre microsserviços e Spring Cloud.

Capítulo 16

Alta disponibilidade em sua aplicação

Rodrigo está com aquele sentimento que nem Camões consegue explicar. Ele está feliz que seu site de delivery está com bastante acesso, mas, ao mesmo tempo, está triste que as reclamações de lentidão estão aumentando e ele não sabe exatamente o que deve fazer. Além disso, ele se pergunta se simplesmente melhorar o hardware dos servidores, o que causará um significativo aumento nos gastos, trará o retorno desejado.

O cenário no qual Rodrigo se encontra apresenta de um lado uma alta demanda, com os pedidos on-line, e uma baixa do outro, com o controle de estoque nos cadastros.

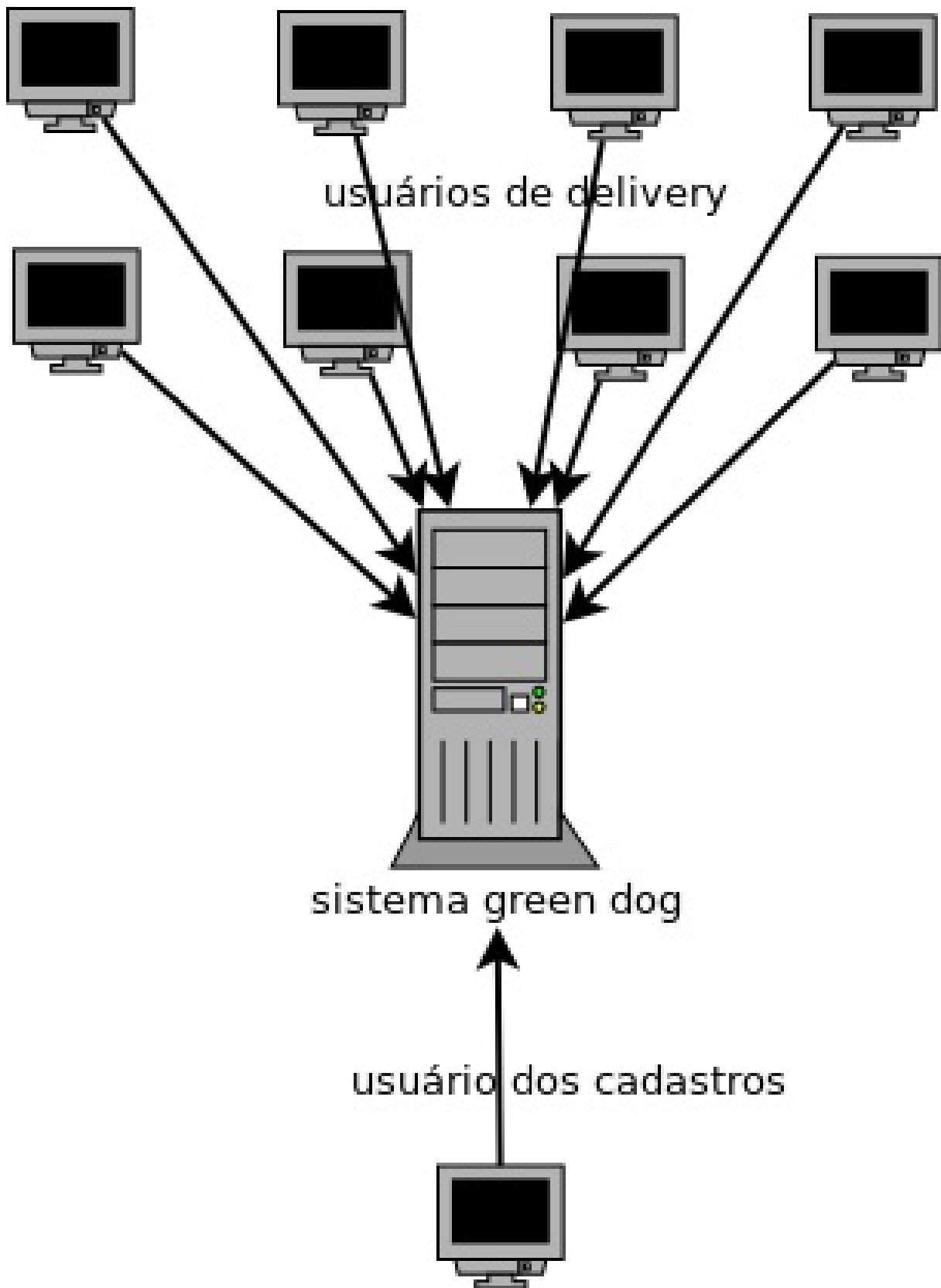


Figura 16.1: Standalone

Hoje temos apenas um sistema que cuida dos cadastros e também faz os pedidos. Com o conceito do microservices (microsserviços), podemos separar os serviços existentes em dois grupos e crescer apenas os que realmente necessitam. Portanto, no atual cenário da empresa do Rodrigo, apenas os serviços de delivery precisam crescer, os de cadastro não.

Levando em conta a simplicidade do sistema, eles são empacotados juntos. Porém, conforme a complexidade aumenta, é interessante diferenciá-los também no pacote de deploy para reduzir o consumo de recursos.

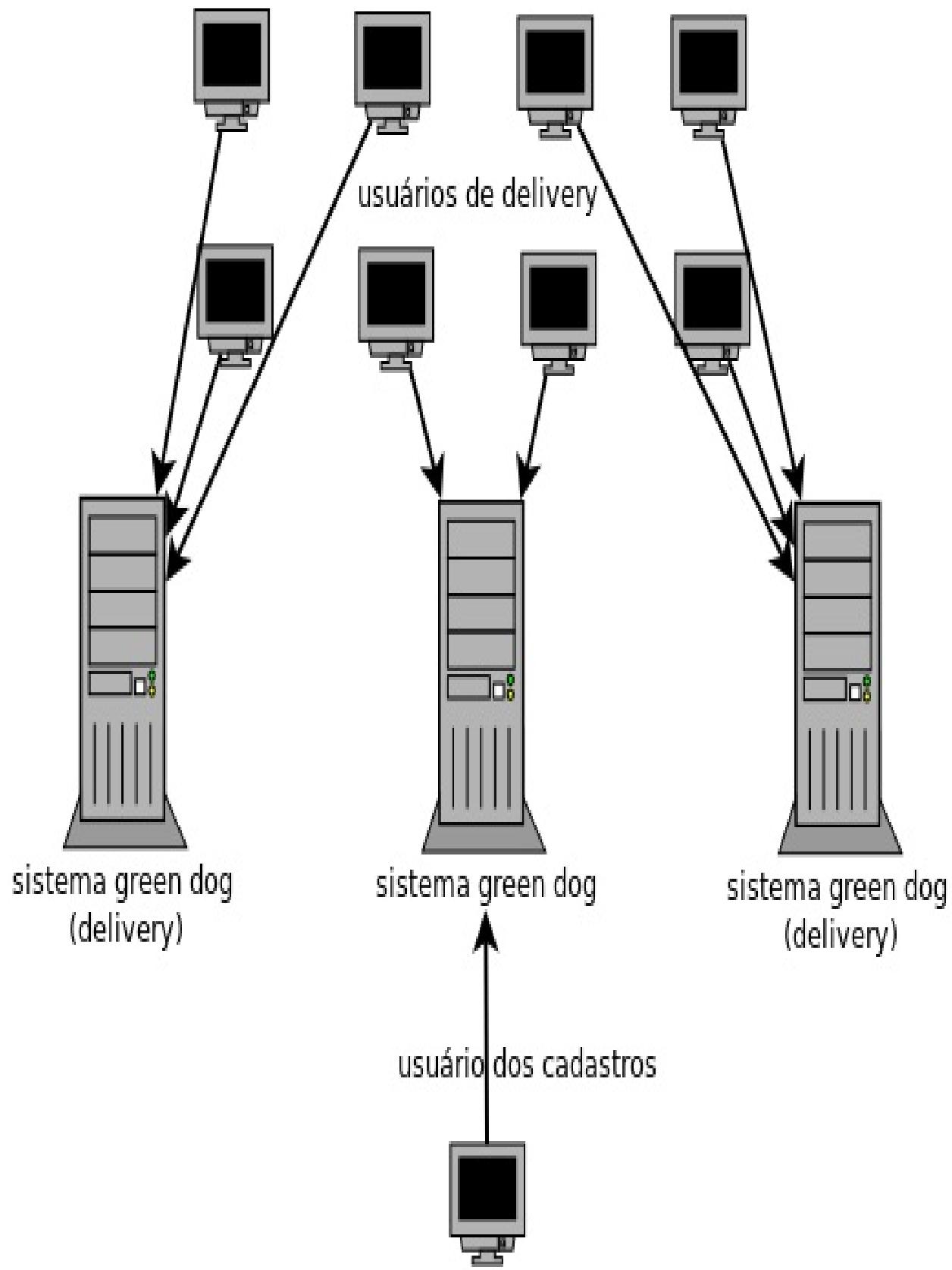


Figura 16.2: Microservices

Spring Cloud

O Spring Cloud é um extenso conjunto de ferramentas para que os desenvolvedores possam rapidamente disponibilizar uma estrutura em cloud (nuvem) usando as melhores práticas do mercado.

Uma dessas práticas é o uso de um Gateway.

Spring Cloud Gateway

O Spring Cloud Gateway é uma solução de padrão Gateway feita com Spring Boot 2 e Spring WebFlux, baseada no servidor Netty (<https://netty.io>).

Com ela, podemos implementar diversos padrões de segurança e alta disponibilidade para microsserviços.

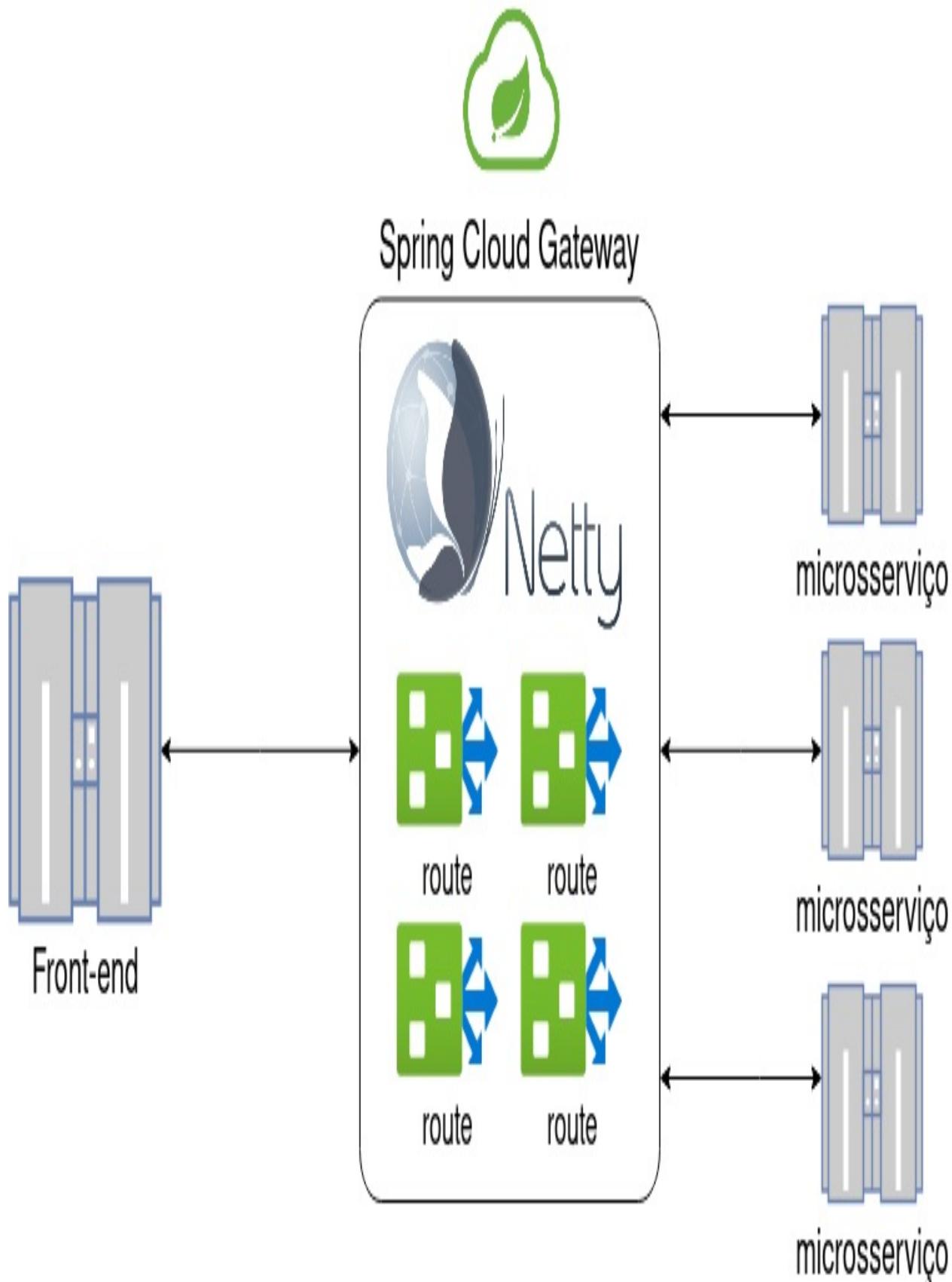


Figura 16.3: Spring Cloud Gateway

Adicionando Gateway ao sistema

Como a aplicação do sistema de pedidos está crescendo, Rodrigo precisa de uma solução de alta disponibilidade para não perder nenhum pedido.

Cenário 1 - dividindo responsabilidades

O primeiro passo para isolar as aplicações para o Spring Cloud Gateway é separar no sistema a parte de front-end da parte de back-end.

No nosso exemplo, vamos usar o mesmo pacote para representar o sistema de pedidos front-end e back-end:

<http://localhost:8080> - front-end

<http://localhost:8081> - back-end

Para subir o back-end precisamos apenas empacotar o sistema e chamar com outro arquivo application.properties:

```
mvn package
```

```
java -jar target/green-dog-delivery-
```

2.4.0

```
-SNAPSHOT.jar
```

```
--spring.config.location=src/main/resources/application-back-end1.properties
```

O front-end precisa ser alterado para acessar o novo back-end. Portanto, precisamos alterar novamente o arquivo delivery.js e mudar a chamada para o destino do novo servidor:

```
// $scope.urlPedido="/rest/pedido/novo/2/" + pedidoStr;
```

```
$scope.urlPedido=  
"http://localhost:8081/rest/pedido/novo/2/" + pedidoStr;
```

Entretanto, subindo os dois servidores e fazendo um pedido novo via browser, percebemos que o pedido não é feito e retorna um erro no console JavaScript, informando que a chamada do serviço foi bloqueada por uma política CORS.

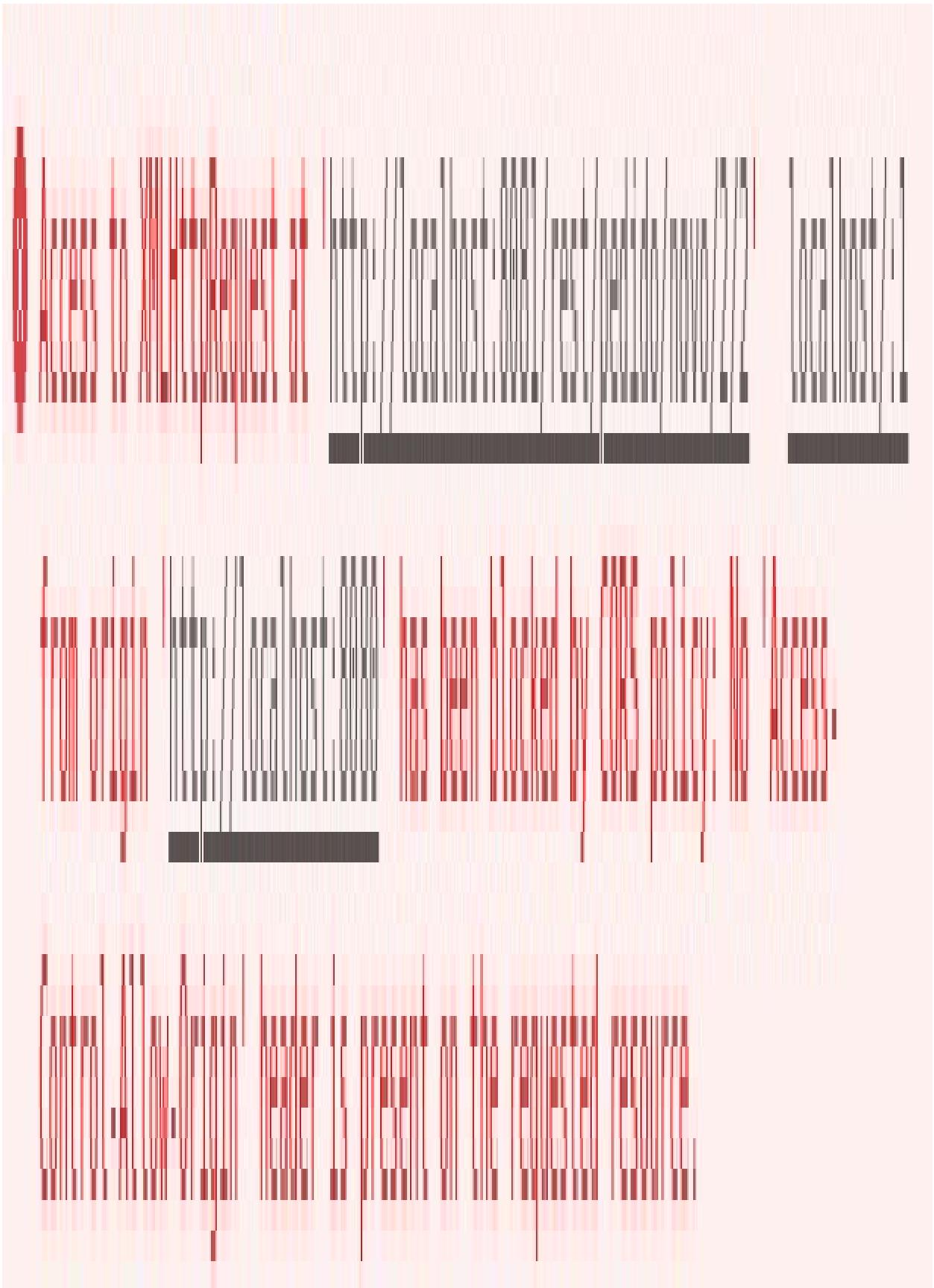


Figura 16.4: Erro CORS

CORS (Cross-Origin Resource Sharing) é um mecanismo que permite que recursos restritos em uma página da web sejam recuperados por outro domínio de fora.

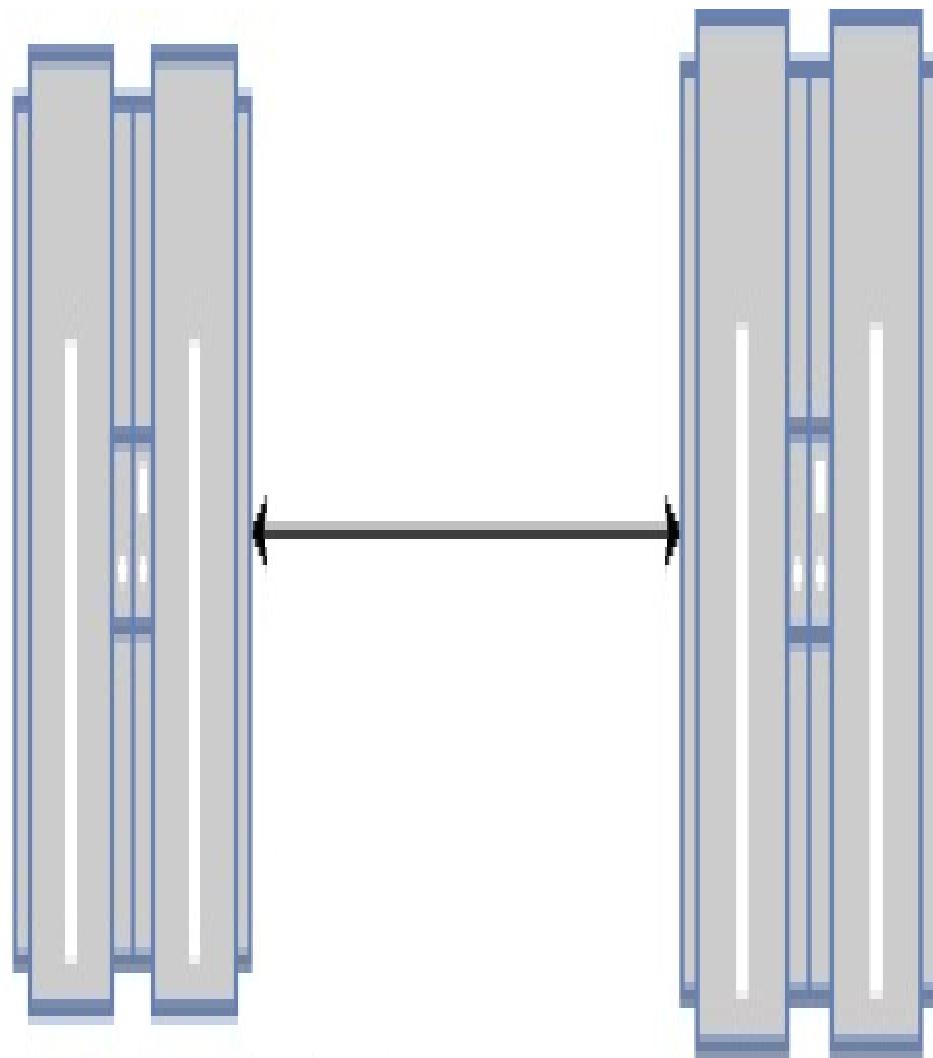
Isso serve para avisar ao Web Browser que o servidor do front-end (localhost:8080) é diferente do back-end (localhost:8081). Isso pode ser feito via um HTTP header:

Access-Control-Allow-Origin: http://localhost:8080

No Spring Boot, isso é feito com uma anotação no método chamado:

```
@CrossOrigin(origins = {"http://localhost:8080"})
@GetMapping("/rest/pedido/novo/{clienteId}/{listaDeItens}") public
RespostaDTO novo
```

Com essa alteração, o cenário está pronto. Temos os dois ambientes separados e podemos fazer pedidos com sucesso.



Sistema de Pedidos Front-end

Sistema de Pedidos Back-end (API)

Figura 16.5: Dois servidores sem gateway

Cenário 2 - Spring Cloud Gateway no meio

Agora vamos iniciar o terceiro projeto chamado de spring-cloud-greendogdelivery-casadocodigo, que usará a porta 10000.

No pom.xml colocaremos as dependências principais do Spring Cloud Gateway:

```
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-gateway</artifactId> </dependency>
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency> <dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-redis-reactive</artifactId> </dependency>
```

E toda a configuração do Spring Cloud Gateway pode ser feita no Java ou em arquivos properties/yaml.

A configuração é composta de três itens básicos:

Route: a rota é definida por um ID, uma URI de destino, uma coleção de predicates (predicados) e filters (filtros).

Predicate: é a função Predicate do Java 8 que permite comparar tudo o que vem da request HTTP, headers e parâmetros.

Filter: são filtros do tipo Spring Framework GatewayFilter que podem modificar requests e responses antes e depois de enviar.

Vamos configurar inicialmente o nosso arquivo application.yml:

server:

port: 10000

spring:

cloud:

gateway:

routes:

- id: umaMaquina

uri: http://localhost:8081

predicates:

- Path=/rest/pedido/novo/**

Nessa configuração, tudo que chegar em /rest/pedido/novo/* será redirecionado para http://localhost:8081/rest/pedido/novo/*.

E para usarmos o Spring Cloud Gateway, precisamos mudar o apontamento anterior à porta 10000:

```
// $scope.urlPedido="/rest/pedido/novo/2/"+pedidoStr; //
$scope.urlPedido="http://localhost:8081/rest/pedido/novo/2/"+pedidoStr;
```

```
$scope.urlPedido= "http://localhost:10000/rest/pedido/novo/2/" + pedidoStr;
```

Com essa alteração, temos o Spring Cloud Gateway no meio do caminho protegendo o back-end. Além de proteger, podemos evoluir o back-end sem nenhuma alteração no front-end, como dobrar o número de servidores de pedidos back-end. A responsabilidade de chamar o servidor certo não é mais do front-end, é do Spring Cloud Gateway.

As alterações do front-end encerraram, agora podemos focar apenas no Gateway.

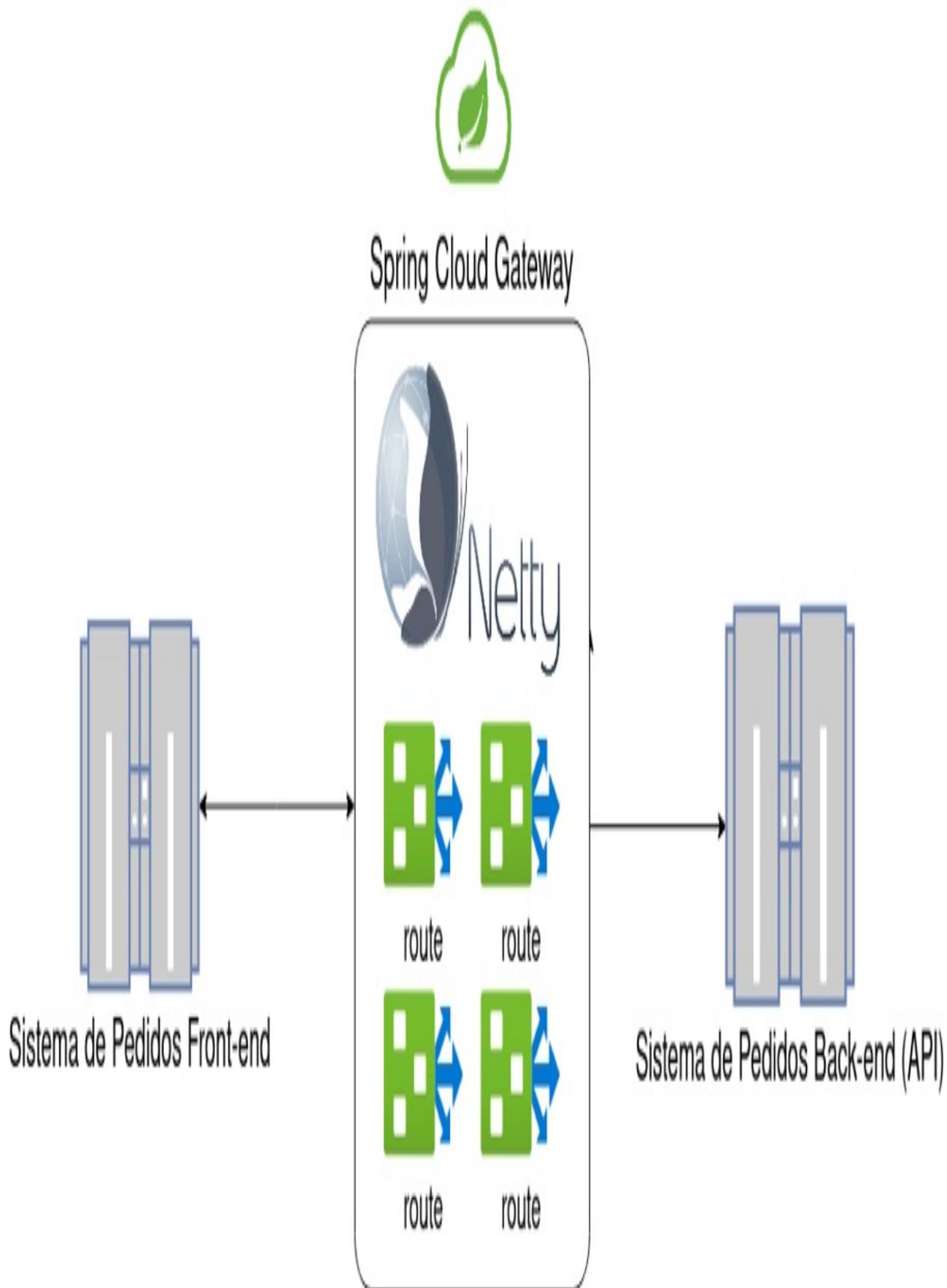


Figura 16.6: Cenário com um back-end

Além do browser, podemos acessar diretamente o Gateway para simular um novo pedido:

```
$ http :10000/rest/pedido/novo/2/2
```

HTTP/

1.1 200

OK

{

"mensagem": "Pedido efetuado com sucesso"

,

"pedido": 8

,

"valorTotal": 30.0

```
}
```

Cenário 3 - Spring Cloud Gateway distribuindo carga

Vamos adicionar mais uma máquina de back-end:

<http://localhost:8080> - front-end

<http://localhost:8081> - back-end

<http://localhost:8082> - back-end

Entretanto, o servidor da porta 8081 é muito mais robusto de CPU e memória RAM que o da porta 8082. Com o Spring Cloud Gateway, podemos, além de distribuir a carga, dar um peso maior, ou seja, 80% das requisições serão redirecionadas para a máquina boa.

```
spring:
```

```
  cloud:
```

```
    gateway:
```

```
      routes:
```

```
        - id: maquina_boa
```

```
          uri: http://localhost:8081
```

predicates:

- Weight=servidoresDePedidos, 8

- Path=/rest/pedido/novo/**

- id: maquina_ruim

uri: http://localhost:8082

predicates:

- Weight=servidoresDePedidos, 2

- Path=/rest/pedido/novo/**

Com o predicado Weight (peso), definimos que do grupo de servidores servidoresDePedidos uma máquina tem o peso 8 e a outra o peso 2.

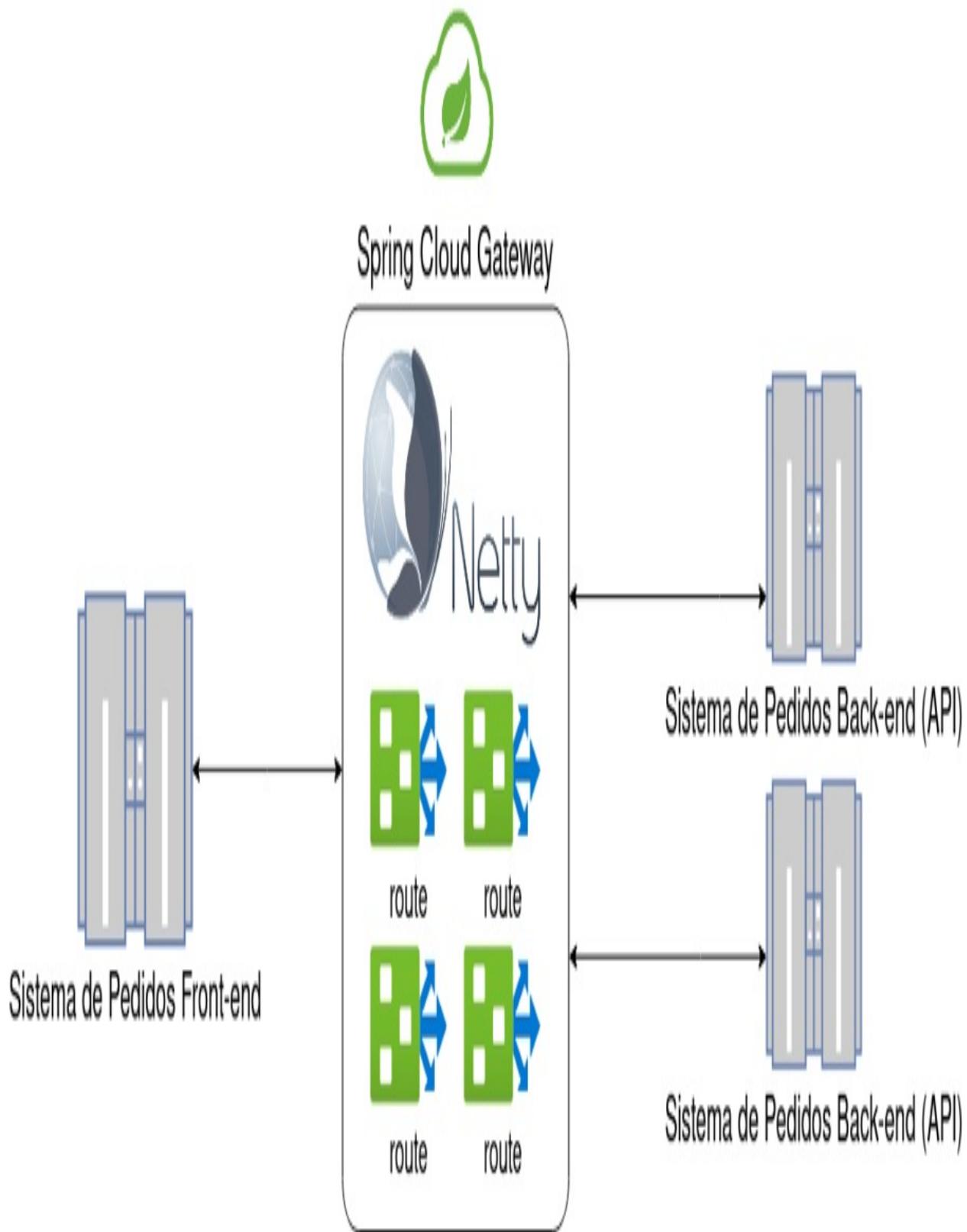


Figura 16.7: Cenário com dois back-ends

Vamos adicionar também a chamada ao end-point servidor, ainda usando o predicado Weight com um grupo diferente chamado servidores:

spring:

cloud:

gateway:

routes:

- id: servidor_maquina_boa

uri: http://localhost:8081

predicates:

- Weight=servidores, 8

- Path=/servidor

- id: maquina_boa

uri: http://localhost:8081

predicates:

- Weight=servidoresDePedidos, 8

- Path=/rest/pedido/novo/**

- id: servidor_maquina_ruim

uri: http://localhost:8082

predicates:

- Path=/servidor

- Weight=servidores, 2

- id: maquina_ruim

uri: http://localhost:8082

predicates:

- Weight=servidoresDePedidos, 2

- Path=/rest/pedido/novo/**

Testando o end-point servidor várias vezes, podemos conferir como o Gateway faz o balanceamento:

```
$ http :10000
```

```
/servidor
```

localhost:

8081

Cenário 4 - Spring Cloud Gateway com filtro de tamanho

Uma das vantagens do Spring Cloud Gateway é a possibilidade de colocar proteções para o back-end. Uma delas é o tamanho da requisição. Isso é feito com o filtro RequestSize.

spring:

cloud:

gateway:

routes:

- id: servidor_maquina_boa

- uri: http://localhost:8081

- predicates:

- Weight=servidores, 8

- Path=/servidor

- id: maquina_boa

- uri: http://localhost:8081

- predicates:

- Weight=servidoresDePedidos, 8

- Path=/rest/pedido/novo/**

- filters:

- name: RequestSize

args:

maxSize: 5

- id: servidor_maquina_ruim

uri: http://localhost:8082

predicates:

- Path=/servidor

- Weight=servidores, 2

- id: maquina_ruim

uri: http://localhost:8082

predicates:

- Weight=servidoresDePedidos, 2

- Path=/rest/pedido/novo/**

filters:

- name: RequestSize

args:

maxSize: 5

Com o filtro RequestSize definido em ambos os servidores, o tamanho máximo da requisição está com 5 bytes. Definimos um número baixo porque, no nosso caso, a API aceita apenas GET e não necessita de body.

Vamos repetir o teste de pedido com uma request usando apenas "{}":

```
curl -X GET -d '{}'  
http://localhost:  
10000/rest/pedido/novo/2/2  
  
{  
  "valorTotal":30.0,"pedido":9,"mensagem":"Pedido efetuado com sucesso"}
```

Agora com uma request um pouco maior, a conexão é bloqueada:

```
curl -v -X GET -d '{"ataque":"request grande"}'
```

```
http://localhost:  
10000/rest/pedido/novo/2/2
```

```
> GET /rest/pedido/novo/
```

```
2/2 HTTP/1.1
```

```
> Host: localhost:
```

```
10000
```

* upload completely sent off:

27 out of 27

bytes

< HTTP/

1.1 413

Request Entity Too Large

< errorMessage: Request size is larger than permissible limit.

Request size is

27 B where permissible limit is 5 B

Com esse novo cenário, em um evento como a Black Friday, por exemplo, novas máquinas podem ser adicionadas ao Gateway, mantendo a alta disponibilidade e sem alteração no front-end.

16.1 O que aprendemos

Os fontes estão divididos em dois projetos:

Projetos do Spring Cloud — <https://github.com/boaglio/spring-cloud-greendogdelivery-casadocodigo>

Green Dog adaptado ao cloud — <https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/cloud>

Certifique-se de que você aprendeu:

os princípios básicos do Spring Cloud e Spring Cloud Gateway;

a configurar e a usar o Spring Cloud Gateway.

No próximo capítulo, vamos analisar algumas ferramentas muito úteis não oficiais do Spring Boot.

Capítulo 17

Ferramentas não oficiais

Toda a plataforma do Spring Boot oferece diversos recursos para manter o sistema, entretanto existem algumas ferramentas não oficiais que são um bom complemento.

17.1 Spring Boot Admin

Spring Boot Admin é uma interface web de administração do Spring Boot. A sua instalação é bem simples, e precisa de um servidor dedicado.

Vamos fazer um servidor Spring Boot Admin na porta 8888.

Em um novo projeto, adicionamos a dependência da parte Server no arquivo pom.xml:

```
<dependency>
```

```
<groupId>de.codecentric</groupId>
```

```
<artifactId>spring-boot-admin-starter-server</artifactId>
```

```
<version>2.3.1</version> </dependency>
```

E depois, na classe inicial do Spring Boot, a anotação `EnableAdminServer`:

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

E finalmente `application.properties` para mudar a porta para 8888:

```
server.port=8888
```

Depois de subir o servidor Spring Boot Admin, ele está pronto para receber o registro de aplicações novas e informações já registradas, acessível em `http://localhost:8888`.

O registro de uma aplicação também é simples, primeiro adicionamos a dependência da parte Client no arquivo pom.xml:

```
<dependency>
```

```
<groupId>de.codecentric</groupId>
```

```
<artifactId>spring-boot-admin-starter-client</artifactId>
```

```
<version>2.3.1</version> </dependency>
```

Em seguida, informamos no application.yaml da aplicação para ser administrada, onde está o servidor do Spring Boot Admin:

```
spring:
```

```
boot:
```

```
admin:
```

```
client:
```

url: 'http://localhost:8888'

No próximo restart da aplicação, o registro ao Spring Boot Admin será automático, e poderemos administrá-lo de uma maneira mais amigável:



spring-boot-
application
1ee091aaaa8a

Insights

Details

Metrics

Environment

Beans

Configuration

Properties

Scheduled Tasks

Loggers

spring-boot-application

Id: 1ee091aaaa8a

<http://cascao:8080/> <http://cascao:8080/actuator> <http://cascao:8080/actuator/health>

Info

app
name: greendogdelivery
description: Greendog Delivery
version: 2.4.0-SNAPSHOT

fernando Boaglio

Health

Instance

db

database MySQL

validationQuery isValid()

Figura 17.1: Spring Boot Admin

Os códigos-fontes do projeto cliente estão em <https://github.com/boaglio/spring-boot-greendogdelivery-estoque-casadocodigo/tree/melhorias>.

Mais informações na documentação oficial:
<https://github.com/codecentric/spring-boot-admin>.

17.2 JavaMelody

Diferente do Spring Boot Admin, o JavaMelody é um famoso monitor do sistema que não precisa de servidor dedicado; ele funciona incorporado ao sistema do Spring Boot.

Sua instalação também é simples, primeiro adicionamos a dependência no arquivo pom.xml:

```
<dependency>
```

```
    <groupId>net.bull.javamelody</groupId>
```

```
    <artifactId>javamelody-spring-boot-starter</artifactId>
```

```
    <version>1.86.0</version> </dependency>
```

E depois suas configurações no arquivo application.yaml:

```
server:
```

```
  port: 9000
```

```
javamelody:
```

```
  # Enable JavaMelody auto-configuration (optional, default: true)
```

```
  enabled: true
```

```
  # Enable monitoring of Spring services and controllers (optional, default: true)
```

```
  spring-monitoring-enabled: true
```

Depois de reiniciar a aplicação, o JavaMelody está disponível em <http://localhost:9000/monitoring>.

System information



Host:	cascao@192.168.0.20
Java memory used:	161 Mb / 8,014 Mb 
Nb of http sessions:	0
Nb of active threads (current http requests):	0
Nb of active jdbc connections:	0
Nb of used jdbc connections (opened if no datasource):	0
System load	2.57
% System CPU	20.61  Details

OS:	 Linux, 5.10.7-arch1-1 , amd64/64 (8 cores)
Java:	Java(TM) SE Runtime Environment, 11+28
JVM:	Java HotSpot(TM) 64-Bit Server VM, 11+28, mixed mode
PID of process:	1090388
Nb of opened files	162 / 40,960 
Server:	 Apache Tomcat/9.0.41

Figura 17.2: JavaMelody

Os códigos-fontes do projeto estão em <https://github.com/boaglio/spring-boot-greendogdelivery-estoque-casadocodigo/tree/melhorias2>.

Mais informações na documentação oficial:

<https://github.com/javamelody/javamelody/wiki/SpringBootStarter>

17.3 JHipster

O JHipster é uma plataforma que gera uma aplicação completa baseada em Spring Boot, na qual rapidamente conseguimos configurar um microsserviço que já se integra ao banco de dados.

Ela pode ser usada com Docker, mas sua instalação clássica é via NPM:

```
$ npm install -g generator-jhipster
```

Para criar uma aplicação, crie um diretório, entre nele e chame o jhipster:

```
$ mkdir app
```

```
$
```

```
cd
```

```
app
```

```
$ jhispter
```

Depois de responder ao questionário, todo o código-fonte será gerado nesse

diretório.

```
INFO! Using JHipster version installed globally
```

```
INFO! Executing jhipster:app
```

JHIPSTER

<https://www.jhipster.tech>

Welcome to JHipster v6.10.5

Application files will be generated in folder: /app

Documentation for creating an application is at <https://www.jhipster.tech/creating-an-app/>

If you find JHipster useful, consider sponsoring the project at <https://opencollective.com/generator-jhipster>

? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)

? [Beta] Do you want to make it reactive with Spring WebFlux? No

? What is the base name of your application? casadocodigo

? What is your default Java package name? com.boaglio

? Do you want to use the JHipster Registry to configure, monitor and scale your application? No

? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)

? Which *type* of database would you like to use?

SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)

> MongoDB

Cassandra

Couchbase

[BETA] Neo4j

No database

Figura 17.3: JHipster console

Depois do código gerado, podemos subir como uma aplicação Spring Boot:

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

Depois de subir a aplicação, podemos navegar pela interface web:

 Casadocodigo 0.0.1-SNAPSHOT

Inicio Entidades Administração Português (Brasil) Conta

Bem vindo, Java Hipster!

Esta é a página principal

Você está logado como "admin".

Em caso de dúvida sobre o JHipster:

- Página principal JHipster
- JHipster no Stack Overflow
- JHipster bug tracker
- Sala de bate-papo pública JHipster
- siga @jhipster no Twitter

Se você gosta do JHipster, não se esqueça de avaliar no [Github](#)!



Este é o rodapé da página

- Gerenciamento de usuário
- Métricas
- Estado do Sistema
- Configuração
- Auditorias
- Logs
- API

Figura 17.4: JHipster dashboard

Mais informações na documentação oficial: <https://www.jhipster.tech>.

17.4 O que aprendemos

Certifique-se de que você aprendeu:

- a instalar e a configurar Spring Boot Admin;
- a instalar e a configurar JavaMelody;
- a instalar e a configurar JHipster.

No próximo capítulo, vamos descobrir como saber mais sobre o Spring Boot.

Capítulo 18

Indo além

18.1 Sistema entregue

No final do livro, Rodrigo conseguiu criar um sistema de cadastro de itens, pedidos e clientes. Depois disso, criou facilmente um esquema para monitorar a aplicação em produção, seguido de testes unitários e de integração.

Em seguida, ele colocou o sistema na nuvem e aprendeu a quebrar o sistema em microsserviços, separando apenas a parte de pedidos para alta disponibilidade. O próximo passo de Rodrigo é utilizar as referências a seguir para se aprofundar no Spring Boot e acompanhar as novidades do framework.

18.2 Sugestões complementares

O ecossistema do Spring é enorme, e o Spring Boot é apenas parte dele.

A principal referência é a documentação oficial: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>.

Outras referências são:

Spring Cloud — <https://spring.io/projects/spring-cloud>.

Exemplos e tutoriais de Spring — <https://www.baeldung.com>.

Apresentações

Dezenas de apresentações de Spring com profissionais da Pivotal, committers dos projetos e cases de sucesso — <https://springone.io/>.

Microsserviços - Dan Woods — <https://www.infoq.com.br/articles/boot-microservices>.

Podcast

Pivotal Conversations - Podcast feito pela equipe da Pivotal sobre notícias de tecnologia, algumas vezes relacionadas ao Spring —
<https://content.pivotal.io/podcasts>.

Blogs

Blog oficial do Spring — <https://spring.io/blog>.

Dicas e tutoriais — <https://domineospring.wordpress.com>.

Tutoriais de um ex-funcionário da Pivotal —
<https://springframework.guru/blog/>.

Twitter

Twitter oficial do Spring — <https://twitter.com/springcentral>.

Pivotal — Empresa que mantém o Spring financeiramente —
<https://twitter.com/pivotal>.

Rod Johnson — Criador do Spring — <https://twitter.com/springrod>.

Josh Long — Desenvolvedor Spring — <https://twitter.com/starbuxman>.

Engine Thymeleaf — <https://twitter.com/thymeleaf>.

Livros

Vire o jogo com Spring Framework, de Henrique Lobo Weissmann — <https://www.casadocodigo.com.br/products/livro-spring-framework>.

OAuth 2.0: Proteja suas aplicações com o Spring Security OAuth2, de Adolfo Eloy - <https://www.casadocodigo.com.br/products/livro-oauth>.

Considerações finais

Espero que tenha gostado do livro. Acompanhe o fórum da Casa do Código para dúvidas e sugestões (<https://forum.casadocodigo.com.br>).

Obrigado pela leitura!

Siga-me no Twitter

<https://twitter.com/boaglio>

18.3 Referências bibliográficas

FOWLER, Martin. Microservice Prerequisites. Ago. 2014. Disponível em:
<https://martinfowler.com/bliki/MicroservicePrerequisites.html>.

FOWLER, Martin. Richardson Maturity Model steps toward the glory of REST. Mar. 2010. Disponível em:
<https://martinfowler.com/articles/richardsonMaturityModel.html>.

WEBB, Phillip; SYER, Dave; LONG, Josh; NICOLL, Stéphane; WINCH, Rob; WILKINSON, Andy; OVERDIJK, Marcel; DUPUIS, Christian; DELEUZE, Sébastien; SIMONS, Michael. Spring Boot Reference Guide. 2012. Disponível em: <https://docs.spring.io/spring-boot/docs/current/reference/pdf/spring-boot-reference.pdf>.

Capítulo 19

Apêndice A — Starters

Os starters são configurações predefinidas da tecnologia desejada para usar em seu projeto. O uso do starter facilita muito o desenvolvimento, pois ajusta automaticamente todas as bibliotecas e versões, livrando o desenvolvedor dessa árdua tarefa.

A documentação oficial está em <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter>.

Starters

Starters de produção

Starters de servidor de aplicação

|

Capítulo 20

Apêndice B — Resumo das propriedades

Existem diversas propriedades no Spring Boot separadas por categorias e subcategorias.

Consulte a documentação oficial em <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>.



Extraindo configurações

O Spring Boot consegue ler as configurações externas de diferentes maneiras, entretanto é usada a seguinte ordem:

Argumento de linha de comando.

Java System properties (usando `System.getProperties()`).

Variáveis do sistema operacional.

Anotação `@PropertySource` dentro das classes `@Configuration`.

Arquivo `application.properties` fora do JAR da aplicação.

Arquivo `application.properties` dentro do JAR da aplicação.

Propriedade usando `SpringApplication.setDefaultProperties`.

Usando linha de comando

Frequentemente, precisamos alterar o comportamento da aplicação Spring Boot empacotada no JAR.

Sabemos que, por padrão, a aplicação sobe na porta 8080. Para simular um cluster com dois sistemas rodando na mesma máquina, precisamos subir o mesmo sistema em uma porta HTTP diferente, por exemplo, na 8081.

Se fosse uma alteração no arquivo `application.properties` do sistema, a alteração seria na propriedade:

```
server.port=8081
```

Com o pacote JAR da aplicação Spring Boot criado, podemos alterar o valor existente com o comando:

```
$ java -jar app.jar --server.port=8081
```

Ou assim:

```
$ java -jar -Dserver.port=8081 app.jar
```

Se o arquivo application.properties estiver fora do pacote JAR, podemos usar assim:

```
$ java -jar app.jar --spring.config.location=outras-prop.properties
```

Capítulo 21

Apêndice C — Guia de atualização

Vamos mostrar um guia rápido de atualização de aplicações Spring Boot.

A maneira mais segura de migrar aplicações é usando esta ordem:

Subir versão para 1.5.

Subir versão para 2.0.

Subir versão para 2.2.

Subir versão para 2.4.

Apesar de parecer mais trabalhosa, o ajuste no projeto é gradual. Quando muitas versões são modificadas de uma vez, corremos o risco de não encontrarmos o ajuste correto e pegarmos um erro estranho para o qual não acharemos muita informação na internet.

Versão menor que 1.5 para 1.5.x

O primeiro passo é atualizar para a versão 1.5.x, alterando o pom.xml e corrigindo manualmente algumas dependências e alguns imports de algumas classes do projeto.

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-parent</artifactId>
```

```
<version>1.5.22.RELEASE</version>
```

```
<type>pom</type> </dependency>
```

Versão 1.5.x para 2.0.x

No branch 2 ocorreram várias mudanças, portanto a melhor estratégia é migrar primeiro as dependências e depois as propriedades.

Migrando dependências

A migração não deve ser feita para a primeira versão (2.0.0) e sim para a mais recente, que contém correções e melhorias.

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-parent</artifactId>
```

```
<version>2.0.9.RELEASE</version>
```

```
<type>pom</type> </dependency>
```

No caso do Thymeleaf, é previsto migrar para a versão 3 usando as duas dependências:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-thymeleaf</artifactId> </dependency>
<dependency>
```

```
<groupId>nz.net.ultraq.thymeleaf</groupId>
```

```
<artifactId>thymeleaf-layout-dialect</artifactId> </dependency>
```

É preciso remover qualquer referência direta da versão, como:

```
<thymeleaf.version>3.0.2.RELEASE</thymeleaf.version> <thymeleaf-layout-
dialect.version>2.1.1</thymeleaf-layout-dialect>
```

Na documentação oficial temos mais informações na migração dos templates:

<https://www.thymeleaf.org/doc/articles/thymeleaf3migration.html>.

Migrando propriedades

Existe uma ferramenta que auxilia na migração das propriedades, basta adicionar a dependência e subir a aplicação:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-properties-migrator</artifactId>
```

```
<scope>runtime</scope> </dependency>
```

Se a ferramenta encontrar algum problema nas propriedades, ela exibirá um relatório do que precisa ser alterado.

Exemplo:

Key: management.security.enabled

Line:

18

Reason: A global security auto-configuration is now provided.

Na documentação oficial temos mais informações na migração do Spring Boot 2.0: <https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.0-Migration-Guide>.

Versão 2.0.x para 2.2.x

Aqui foram feitas poucas mudanças. As mais importantes são:

O spring-boot-starter-test agora traz o JUnit 5 como padrão.

Mudança na API do Spring HATEOAS. Por exemplo, classes que faziam extends do RepositoryRestConfigurerAdapter agora precisam remover o extends e implementar a interface RepositoryRestConfigurer.

Na documentação oficial temos mais informações na migração do Spring Boot 2.2: <https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.2>

Release-Notes.

Versão 2.2.x para 2.4.x

Para a versão 2.4, poucas mudanças foram feitas. A mais importante foi no spring-boot-starter-test, que removeu a biblioteca Vintage Engine (JUnit 5), mecanismo que permite que testes escritos com JUnit 4 sejam executados pelo JUnit 5.

Se for necessário manter os testes em JUnit 4, precisamos alterar as dependências desta forma:

<dependency>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

<scope>test</scope>

```
<exclusions>
```

```
<exclusion>
```

```
<groupId>org.hamcrest</groupId>
```

```
<artifactId>hamcrest-core</artifactId>
```

```
</exclusion>
```

```
</exclusions> </dependency>
```

Na documentação oficial temos mais informações na migração do Spring Boot 2.4: <https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.4-Release-Notes>.