

Spring Boot

Acelere o desenvolvimento
de microsserviços



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: eletrônicos, mecânicos, gravação ou quaisquer outros.

Adriano Almeida

Livros para o programador

Rua Vergueiro, 3185

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura www.alura.com.br que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso: 978-85-94120-00-7

EPUB: 978-85-94120-01-4

MOBI: 978-85-94120-02-1

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

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:



Figura 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.

Conheça os componentes principais dessa arquitetura revolucionária, tirando o máximo proveito dela vendo os exemplos de acesso a banco de dados, exibição de páginas web usando templates, serviços REST sendo consumidos por front-end em JQuery e AngularJS, testes unitários e de integração, deploy na nuvem e alta disponibilidade com Spring Cloud.

Com dois exemplos completos de sistema, o leitor poderá facilmente adaptar para o seu sistema e tirar proveito das vantagens do Spring Boot 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.

Sumário

1 Tendências do mercado	1
1.1 A evolução dos serviços	1
1.2 Como surgiu o Spring Boot	6
2 Conhecendo o Spring Boot	8
2.1 Sua arquitetura	10
2.2 Nossa ferramenta	11
2.3 Nosso primeiro programa	14
2.4 Ajustando os parafusos	20
2.5 Próximos passos	22
3 Primeiro sistema	23
3.1 O nosso contexto	24
3.2 Criando uma aplicação simples	23
3.3 O nosso contexto	24
3.4 Dados	27
3.5 Usando a aplicação	32
3.6 Usando o console H2	33
3.7 Próximos passos	35

4 Explorando os dados	36
4.1 O novo projeto	37
4.2 As classes de domínio	39
4.3 Repositórios	40
4.4 Carga inicial	41
4.5 Próximos passos	45
5 Explorando os templates	47
5.1 Templates naturais	47
5.2 Convenções	50
5.3 Layout padrão	51
5.4 CRUD	55
5.5 Próximos passos	62
6 Desenvolvimento produtivo	63
6.1 Devtools	63
6.2 LiveReload	65
6.3 Docker	67
6.4 Próximos passos	70
7 Customizando	71
7.1 Banner	71
7.2 Páginas de erro	73
7.3 Actuator	74
7.4 Próximos passos	79
8 Expondo a API do seu serviço	81
8.1 HATEOAS	81
8.2 Angular acessando ReST	85

8.3 Próximos passos	92
9 Testando sua app	94
9.1 Testes unitários	94
9.2 Testes de integração	96
9.3 Próximos passos	98
10 Empacotando e disponibilizando sua app	100
10.1 JAR simples	100
10.2 JAR executável	100
10.3 WAR	103
10.4 Tomcat/Jetty/Undertow	104
10.5 Spring Boot CLI	106
10.6 Próximos passos	107
11 Subindo na nuvem	108
11.1 Profiles	108
11.2 Heroku	114
11.3 Próximos passos	120
12 Alta disponibilidade em sua aplicação	122
12.1 Nosso exemplo	125
12.2 Config server	127
12.3 Eureka Server	129
12.4 Zuul Gateway	130
12.5 Ajuste no sistema atual	132
12.6 Testando nosso cluster	136
12.7 Próximos passos	140
13 Indo além	141

13.1 Referências	141
13.2 Sistema entregue	143
13.3 Considerações finais	143
13.4 Referências Bibliográficas	144
14 Apêndice A — Starters	145
15 Apêndice B — Resumo das propriedades	151

TENDÊNCIAS DO MERCADO

Quem trabalhou com Java 5 ou anterior provavelmente lembra das aplicações grandes e pesadas que engoliam o hardware do servidor e, por uma pequena falha em uma parte da aplicação, comprometia todo o sistema.

Em uma aplicação nova, não podemos mais pensar dessa maneira, mas é interessante entender como chegamos nesse ponto e aprender com os erros do passado.

1.1 A EVOLUÇÃO DOS SERVIÇOS

Era uma vez a adoção das empresas em massa de um tal de Java, que conversava com qualquer banco de dados e até mainframe, fazia transação distribuída e tornava qualquer plataforma confiável para executar os seus sistemas. O que começou a surgir nessas empresas foi o velho (e como é velho) problema do reaproveitamento de código.

Depois do décimo sistema feito por diferentes equipes, foi descoberto que todos eles tinham um mesmo cadastro de clientes. Que tal isolar essa parte em um sistema único?

Assim, na virada do século, começaram os serviços (internos e externos) usando um padrão de comunicação via XML, chamado SOAP (*Simple Object Access Protocol*). Com isso, 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, para um serviço de consulta que retorna um simples valor. 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

Muitos serviços SOAP sendo usados por vários sistemas apareciam rapidamente como a principal causa de lentidão, obrigando os programadores a procurarem por alternativas: 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 com 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.

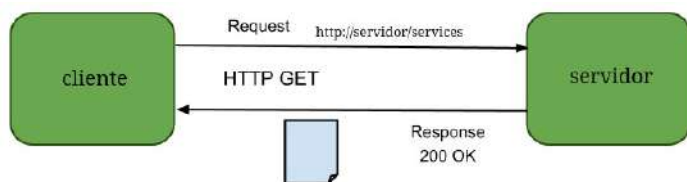


Figura 1.2: Serviços ReST

Pela simplicidade e rapidez, esse padrão foi rapidamente adotado pelo mercado. 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.



Figura 1.3: Glória do REST

Mesmo com os serviços em ReST, as aplicações continuam a empacotar todas as funcionalidades em um lugar só, sendo classificadas como aplicações monolíticas.

Fazendo uma analogia de funcionalidade como sendo um brinquedo, a nossa aplicação sempre oferece a mesma quantidade de brinquedos, independente da demanda. Em um cenário com muitas crianças que só brincam de boneca, sobrarão muitos aviões.

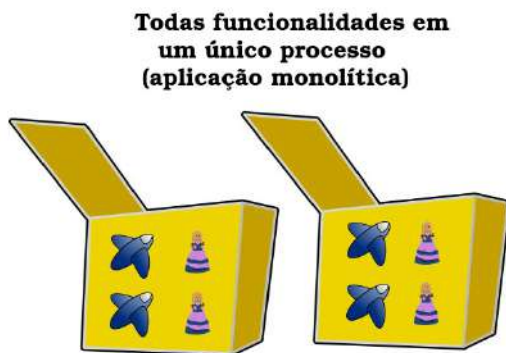


Figura 1.4: Aplicação monolítica

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.

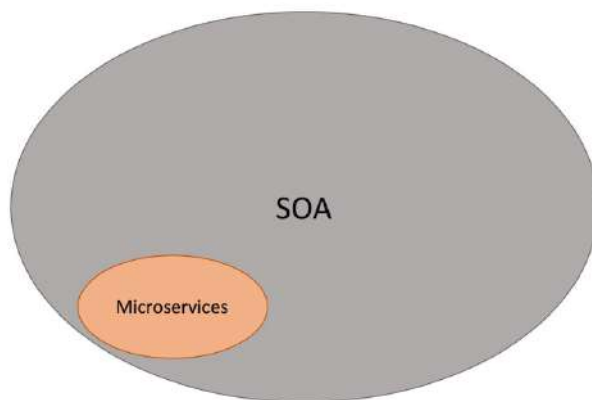


Figura 1.5: SOA

O ideal de uma aplicação é separar suas funcionalidades e se adequar conforme o 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 fique de sobra:

Funcionalidades diferentes em serviços separados (microserviços)

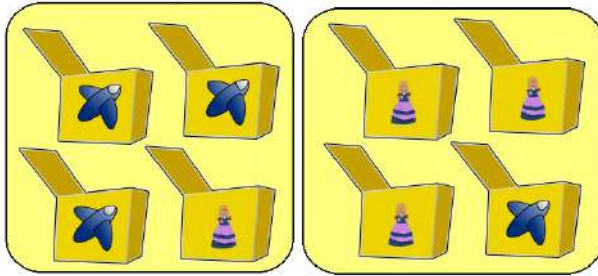


Figura 1.6: Microserviços

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

1.2 COMO SURTIU 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 outras pendências também, como fazer aplicações prontas para nuvem (*cloud-ready applications*). Mais tarde, a anotação `@Conditional` foi criada no Spring Framework 4, e isso 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 que não existe 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 revolucionária ferramenta.

CONHECENDO O SPRING BOOT

Desde 2003, o ecossistema Spring cresceu muito. Do ponto de vista do desenvolvedor, isso é bom, pois aumenta a gama de opções para usar, e ele mesmo não precisa implementar.

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.

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, ajuda na configuração também 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, e ele deverá obedecer aos padrões (servlet, filter, diretório WEB-INF etc.)



Figura 2.1: Boot

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



Figura 2.2: Boot

2.1 SUA ARQUITETURA

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



Figura 2.3: 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** — é um 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** — é uma IDE customizada para o desenvolvimento com Spring Boot;
- **Samples** — dezenas de implementações de exemplos disponíveis para uso.

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

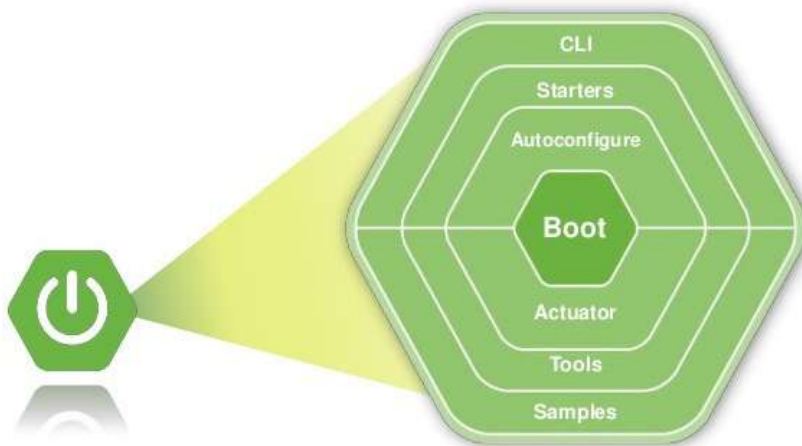


Figura 2.4: Arquitetura do Spring Boot

2.2 NOSSA FERRAMENTA

O *Spring Tools Suite* é um conjunto de ferramentas baseadas no Eclipse para desenvolvimento de aplicações Spring. Ele se encontra disponível em <https://spring.io/tools>.

Existe a opção da ferramenta completa pronta para uso, ou a instalação de um plugin para um Eclipse já instalado. Neste livro, vamos usar a primeira, por ser a mais simples. Se já possuir 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.

Apenas uma breve explicação para o conteúdo do pacote, nele temos três diretórios:

1. `legal` — Contém arquivos texto com as licenças open source;
2. `pivotal` — Contém o *Pivotal TC Server*, uma versão do Tomcat customizada para o Spring;
3. `sts` — Contém o Spring Tools Suite.

O *Pivotal TC Server* é um produto pago em sua versão corporativa, mas a versão developer instalada é gratuita.

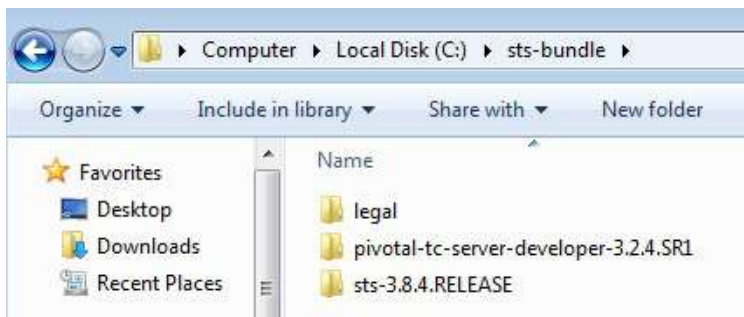


Figura 2.5: Conteúdo do Spring Tools Suite

Dentro do diretório `sts`, execute o executável chamado `STS`.



Figura 2.6: Iniciando o Spring Tools Suite

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.

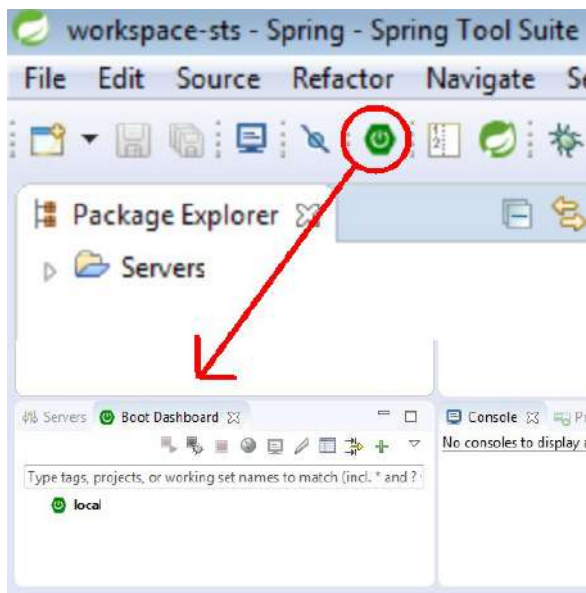


Figura 2.7: Boot Dashboard

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/sts>.

2.3 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** .

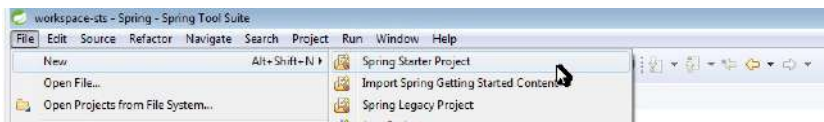


Figura 2.8: Novo projeto

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

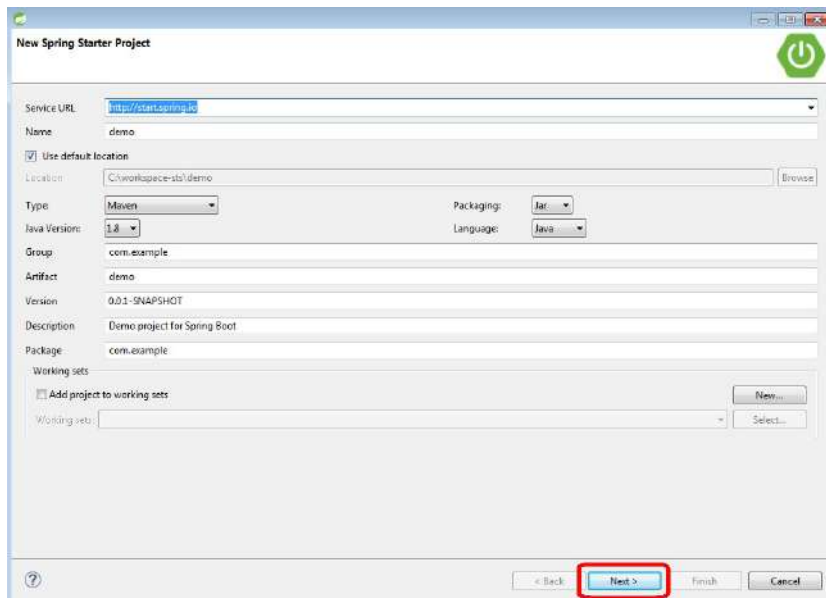


Figura 2.9: Informações padrão do novo projeto

Nessa 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 `web`; em outros exemplos usaremos outras opções.

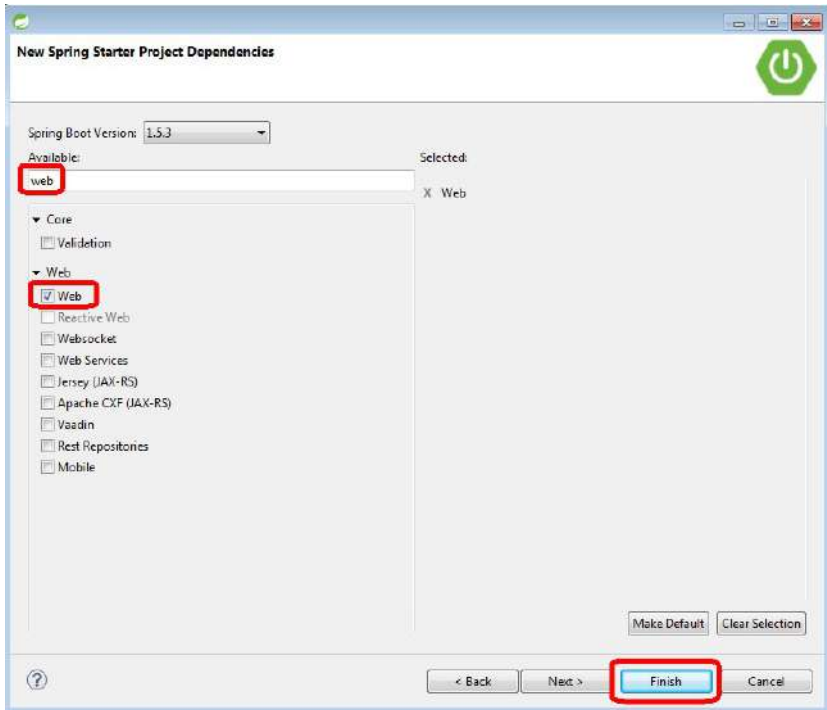


Figura 2.10: Escolhendo dependências do projeto

Clicando em **Finish** , o projeto será criado. Mas existe a opção **Next** , que exibe o link <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.

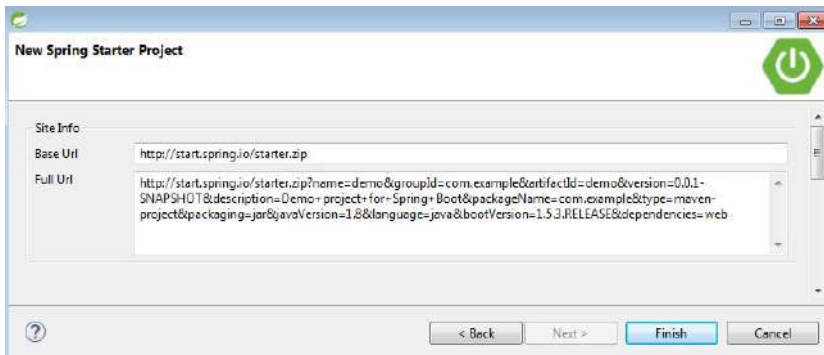


Figura 2.11: Parâmetros de URL da criação do projeto

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

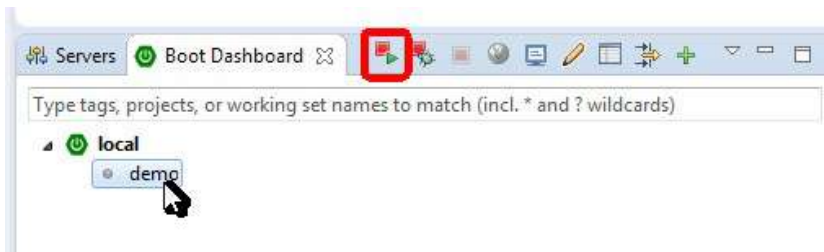


Figura 2.12: Subindo o projeto

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

```

:: Spring Boot :: (v1.5.3.RELEASE)

2017-04-21 23:19:11.929 INFO 3998 --- [main] com.example.DemoApplication : Starting DemoApplication on cecolinha with PID 3998
2017-04-21 23:19:11.935 INFO 3998 --- [main] org.springframework.boot.SpringApplication : No active profile set, falling back to default profile
2017-04-21 23:19:12.973 INFO 3998 --- [main] org.springframework.boot.SpringApplication : Loading source class DemoApplication
2017-04-21 23:19:14.366 INFO 3998 --- [main] org.springframework.boot.SpringApplication : Tomcat initialized with port(s): 8080 (http)
2017-04-21 23:19:14.197 INFO 3998 --- [main] org.apache.catalina.core.StandardService : Starting service Tomcat
2017-04-21 23:19:14.197 INFO 3998 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.14
```

Figura 2.13: Spring Boot no console do Eclipse

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

```

C:\workspace-sts\demo > mvn spring-boot:run

[INFO] Scanning for projects...
[INFO] Building demo 0.0.1-SNAPSHOT
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.3.RELEASE:run (default-cli) @ demo ---
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ demo ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ demo ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ demo ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\workspace-sts\demo\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ demo ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] <<< spring-boot-maven-plugin:1.5.3.RELEASE:run (default-cli) < test-compile @ demo <<<
[INFO] --- spring-boot-maven-plugin:1.5.3.RELEASE:run (default-cli) @ demo ---

:: Spring Boot :: (v1.5.3.RELEASE)

2017-04-21 23:24:00.229 INFO 3928 --- [main] com.example.DemoApplication : Starting DemoApplication on cecolinha with PID 3928
2017-04-21 23:24:00.235 INFO 3928 --- [main] org.springframework.boot.SpringApplication : No active profile set, falling back to default profile
2017-04-21 23:24:01.273 INFO 3928 --- [main] org.springframework.boot.SpringApplication : Loading source class DemoApplication
```

Figura 2.14: Spring Boot no console do Windows

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



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 {

    @RequestMapping("/")
    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.

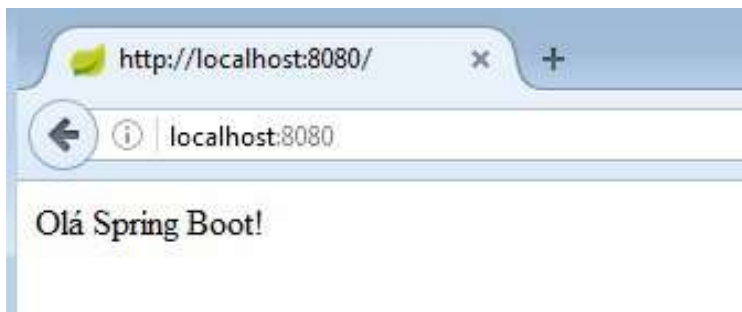


Figura 2.16: Nova página inicial

Vamos avaliar o que aconteceu aqui: escolhendo apenas que queríamos uma aplicação com dependência *web*, conseguimos ter rapidamente rodando uma página web, respondendo o conteúdo que escolhemos e 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.

2.4 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 pré-definidos, 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 possíveis de ajustar. 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.

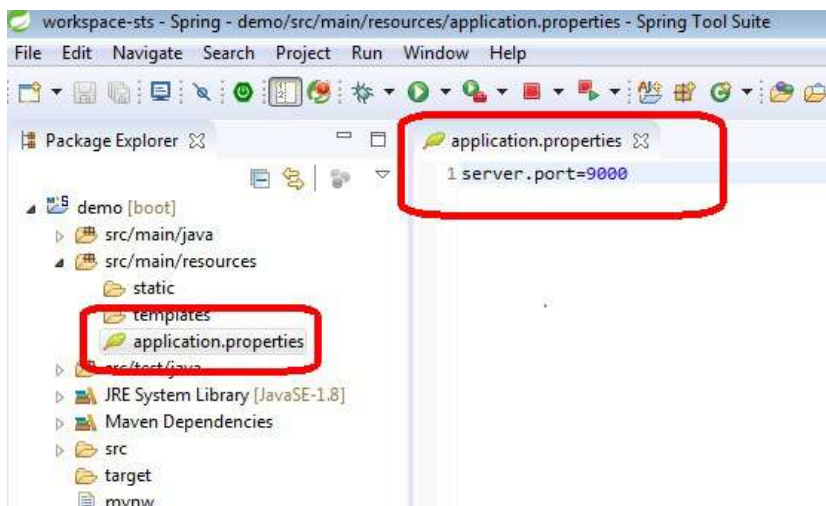


Figura 2.17: Arquivo `application.properties`

Em seguida, apenas com um restart na aplicação, verificamos a

nova porta 9000 sendo usada.

```
INFO 2420 --- [main] com.example.DemoApplication : Starting DemoApplication on ceholinha with PID 2420
INFO 2420 --- [main] com.example.DemoApplication : No active profile set, falling back to default prof
INFO 2420 --- [main] main : etionConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedde
INFO 2420 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 9000 (http)
INFO 2420 --- [main] o.apache.catalina.core.StandardService : Starting service Tomcat
INFO 2420 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.14
INFO 2420 --- [ost-startStop-1] o.s.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
```

Figura 2.18: Porta 9000 em execução

2.5 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- A instalar o Spring Suite Tools;
- 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.

PRIMEIRO SISTEMA

3.1 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 em um banco de dados. Este precisa ser substituído por uma aplicação Java o mais rápido possível.

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.

3.2 CRIANDO UMA APLICAÇÃO SIMPLES

Percebendo a grande quantidade de configurações, Rodrigo teve a ideia de fazer uma aplicação simples que ele pudesse

consultar as propriedades existentes no Spring Boot em uma tela web simples. Dessa maneira, seria possível fazer uma aplicação para validar a facilidade e 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.

Os starters

Os starters são configurações pré-definidas 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 trabalhosa tarefa.

Não devemos nos preocupar como que eles funcionam, 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, será necessário usar mais starters.

3.3 O NOSSO CONTEXTO

Com o nosso escopo definido (se fosse em uma aplicação Java web tradicional, sem nenhum framework), a implementação seria 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 Contêiner para rodar a aplicação.

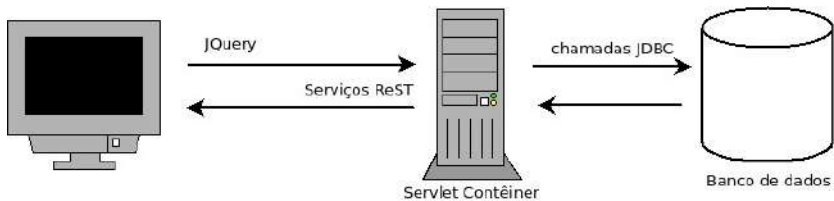


Figura 3.1: Aplicação simples sem framework Java

Utilizando Spring Framework, podemos atualizar essa lista:

- Spring MVC para os serviços ReST;
- Spring MVC para converter os dados em JSON;
- Spring Data para acessar ao 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 Contêiner 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 <http://projects.spring.io/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 <http://projects.spring.io/spring-data/>.

Com a Spring Boot, para fazermos as operações da lista anterior, vamos usar os seguintes starters:

- `spring-boot-starter-web`
- `spring-boot-starter-data-jpa`
- `spring-boot-starter-data-rest`
- `spring-boot-starter-tomcat` .

Iniciamos 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:	1.8	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:

Dependencies:

- Frequency Used
 - ☒ Web
- Cloud Routing
 - ☐ Feign
- SQL
 - ☒ JPA
- Web
 - ☒ Web
 - ☐ Rest Repositories HAL Browser
 - ☐ Jersey (JAX-RS)
 - ☐ REST Docs
 - ☒ Rest Repositories
 - ☐ HATEOAS

rest | Make Default | Clear Selection

Figura 3.3: Escolhendo as dependências do projeto

3.4 DADOS

Temos o projeto criado, agora vamos analisar 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 exemplo a propriedade `server.port` exibida anteriormente:

```
insert into propriedade
(categoria, subcategoria, nome, valor, descricao) values
('WEB PROPERTIES', 'EMBEDDED SERVER CONFIGURATION', '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;
private String subcategoria;

/* getters e setters */
```

A carga inicial será feita automaticamente pelo Spring Boot, precisamos 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, categoria ou subcategoria com Spring Data:

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



```

@Query("Select c from Propriedade c
      where c.nome like %:filtro%
      order by categoria, subcategoria, 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;

    @RequestMapping(value="/find/{filtro}", method=RequestMethod.GET)
    List<Propriedade> findByFiltro
        (@PathVariable("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 espera por convenção 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:

```

function propertiesService(filter) {
    $.ajax({
        type : "GET",
        url : "/api/find/"+filter,

```

```

data : '$format=json',
dataType : 'json',
success : function(data) {
    var total=0;
    $.each(data, function(d, results) {
        $("#propTable tbody").append(
            "<tr>"
            + "<td class=\"text-nowrap\">"
            + results.categoria
            + "</td>" + "<td class=\"text-nowrap\">"
            + results.subcategoria
            + "</td>" + "<td class=\"text-nowrap\">"
            + results.nome
            + "</td>" + "<td class=\"text-nowrap\">"
            + results.valor
            + "</td>" + "<td class=\"text-nowrap\">"
            + results.descricao
            + "</td>"
            + "</tr>")
            total++;
        })
        $("#results").text(total+" found");
    });
});
};

```

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` :

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

```
<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>3.3.7-1</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.1.1</version>
</dependency>
```

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

3.5 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.

```
main| j.LocalContainerEntityManagerFactoryBean : building JPA container EntityManagerFactory for persistence unit 'default'
main| o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
main| org.hibernate.Version : HHH000412: Hibernate Core (5.0.11.Final)
main| org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
main| org.hibernate.cfg.Environment : HHH000021: Bytecode provider name : javassist
main| o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations (5.0.1.Final)
main| org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
main| org.hibernate.tool.hbm2ddl.SchemaExport : HHH000227: Running hbm2ddl schema export
(255) not null, categoria varchar(255), descricao varchar(255), subcategoria varchar(255), valor varchar(255), primary key
main| org.hibernate.tool.hbm2ddl.SchemaExport : HHH000230: Schema export complete
main| j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
main| o.s.jdbc.datasource.init.ScriptUtils : Executing SQL script from URL [file:/home/fb/workspace-sts-livro/springb
main| o.s.jdbc.datasource.init.ScriptUtils : Executed SQL script from URL [file:/home/fb/workspace-sts-livro/springb
main| o.h.h.t.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
main| o.s.b.f.config.PropertiesFactoryBean : Looking for @ControllerAdvice: org.springframework.boot.context.embedded
main| s.w.s.m.a.a.RequestMappingHandlerAdapter : Mapped "[{"/api/find/{filtro}],methods={GET}]" onto java.util.List<con.b
main| s.w.s.m.a.a.RequestMappingHandlerMapping : Mapped "[{/error}],produces={text/html}]" onto public org.springframework
main| s.w.s.m.a.a.RequestMappingHandlerMapping : Mapped "[{/error}]" onto public org.springframework.http.ResponseEntity
main| o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/*] onto handler of type 'class org.springfram
```

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.

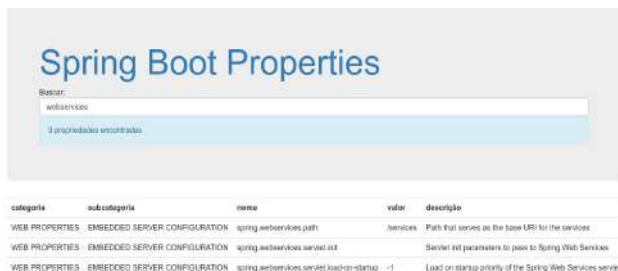


Figura 3.5: Buscando por propriedades de webservices

3.6 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:



Figura 3.6: Entrar no console do H2

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



Figura 3.7: Consultando propriedades via SQL no console do H2

3.7 PRÓXIMOS PASSOS

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

Certifique-se de que aprendeu a:

- Utilizar Spring Data e Spring MVC com Spring Boot;
- Consultar as propriedades do arquivo de configuração `application.properties` ;
- Fazer carga inicial em uma base H2;
- Usar o H2 console.

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

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 de 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;
- Alta disponibilidade para fazer novos pedidos;
- Tela para validar se o ambiente está ok;
- Uso de AngularJS na tela de novo pedido.

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

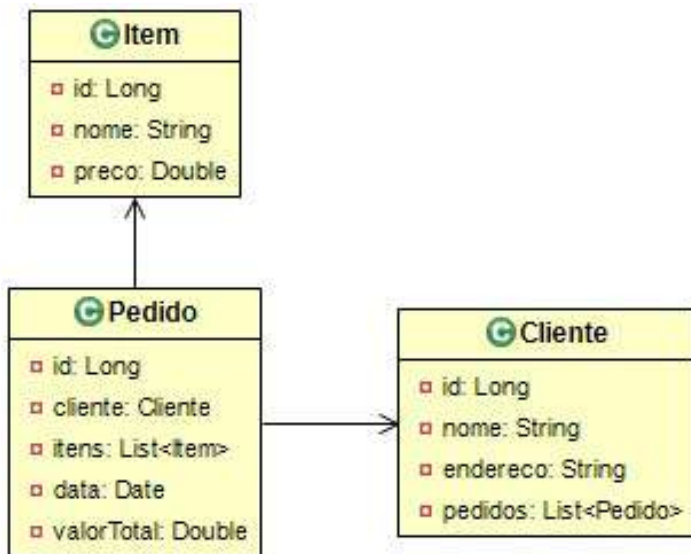


Figura 4.1: Diagrama de classes

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.

4.1 O NOVO PROJETO

Sabemos que precisamos de banco de dados, então vamos usar a opção de *JPA*. Vamos também precisar 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 <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>.

No nosso projeto do Eclipse `springboot-greendogdelivery`, vamos escolher essas duas opções e também a opção de *Rest Repositories*. Se tudo deu 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.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>
```

4.2 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;

```

4.3 REPOSITÓRIOS

Com as classes de domínio, criaremos as classes de repositório. Com a facilidade do Spring Data, precisar gerenciar conexão, chamar `PreparedStatement` e fazer um loop para atribuir o `ResultSet` para 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.

4.4 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 = 111;
private static final long ID_CLIENTE_ZE_PEQUENO = 221;

private static final long ID_ITEM1 = 1001;
private static final long ID_ITEM2 = 1011;
private static final long ID_ITEM3 = 1021;

```

```
private static final long ID_PEDID01 = 10001;
private static final long ID_PEDID02 = 10011;
private static final long ID_PEDID03 = 10021;
```

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",25d);
Item dog2=new Item(ID_ITEM2,"Green Dog tradicional picante",27d);
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_PEDID01,fernando,
    listaPedidoFernando1,dog1.getPreco());
```

```

fernando.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 - Ze 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:greenodog
```

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:

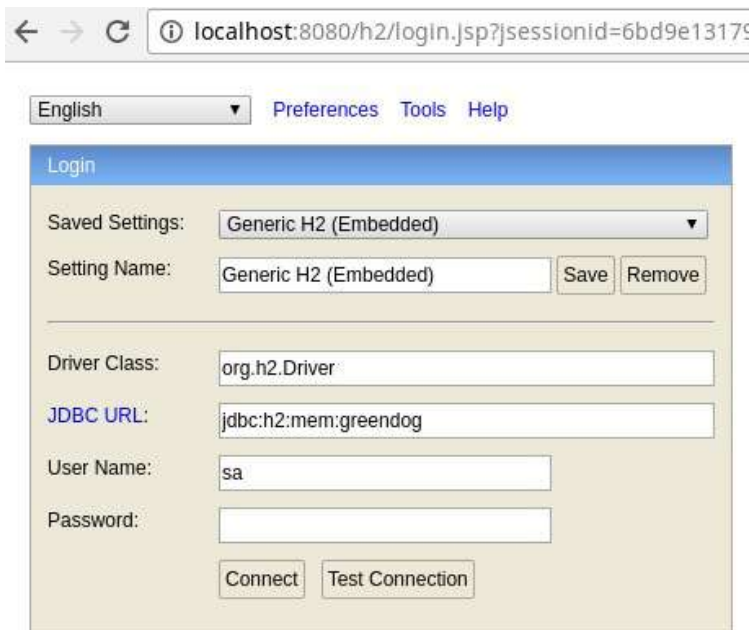


Figura 4.2: Entrar no console H2

E podemos verificar que a carga foi feita com sucesso:

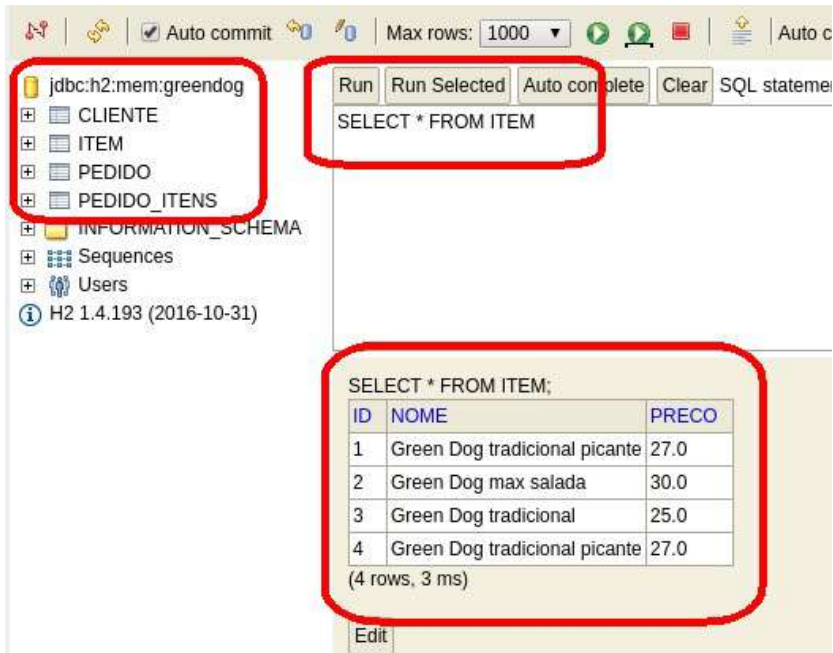


Figura 4.3: Carga inicial no H2

4.5 PRÓXIMOS PASSOS

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

Certifique-se de que aprendeu:

- Como 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.

EXPLORANDO OS TEMPLATES

Rodrigo já conseguiu fazer a carga de seu sistema com o Spring Boot. Agora 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, o JSP é uma engine complicada de trabalhar com os designers, 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

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.

Vejam esse exemplo de uma página JSP renderizada dinamicamente, tudo ok:

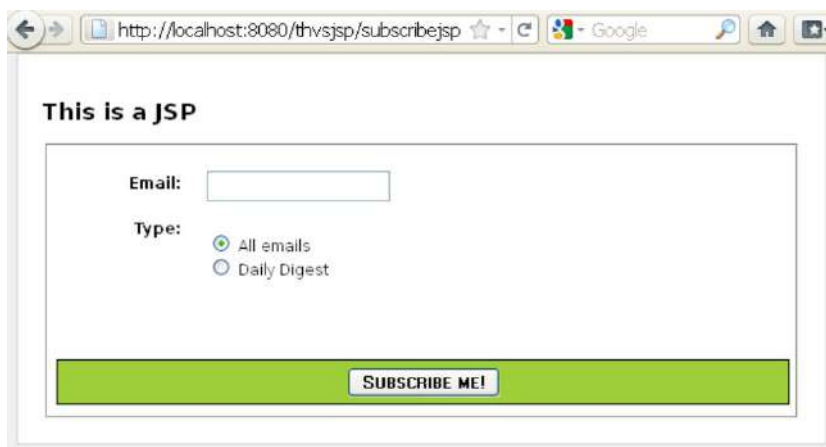


Figura 5.1: Página JSP renderizada

Mas essa mesma página, se abrir a página JSP fonte no web browser, o layout fica comprometido:

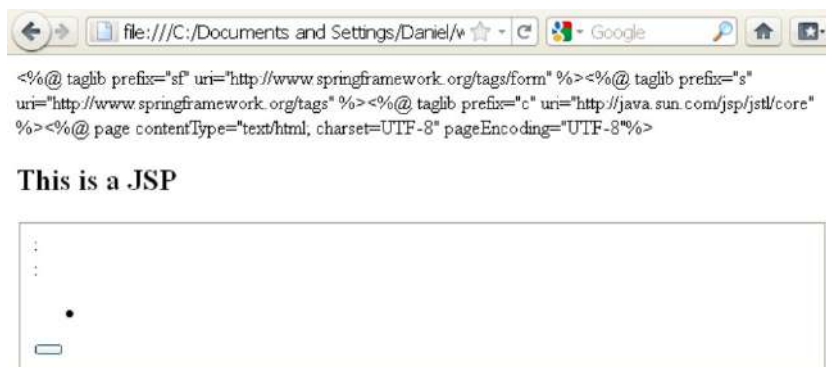


Figura 5.2: Página JSP fonte

A página renderizada no Thymeleaf se comporta igual ao JSP:



Figura 5.3: Página Thymeleaf renderizada

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



Figura 5.4: Página Thymeleaf fonte

Essa "mágica" é possível porque o Thymeleaf trabalha com os seus argumentos dentro de atributos `th`, que o browser ignora, e

mantém o layout *intacto*.

Atributos

Os principais atributos do Thymeleaf são:

- `th:text` — Exibe o conteúdo de uma variável ou expressão, como `th:text="${cliente.id}"` ;
- `th:href` — Monta um link relativo, exemplo:
`th:href="@{/clientes/}"` ;
- `th:src` — Monta link relativo para atributo `src` ,
como `th:src="@{/img/oferta.png}"` ;
- `th:action` — Monta link relativo para tag atributo
`action` de formulário, dessa forma:
`th:action="@{/pedidos/(form)}"` ;
- `th:field` e `th:object` — Relaciona um objeto a
um campo do formulário, como por exemplo:
`th:object="${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'"` .

5.2 CONVENÇÕES

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.

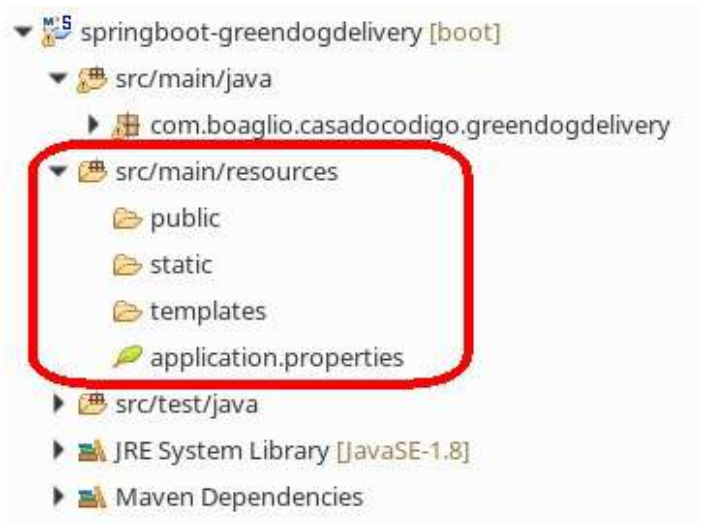


Figura 5.5: Convenções web do Spring Boot

5.3 LAYOUT PADRÃO

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.

Seguindo a convenção do Spring Boot, dentro de `src/main/resources`, 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, onde 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:

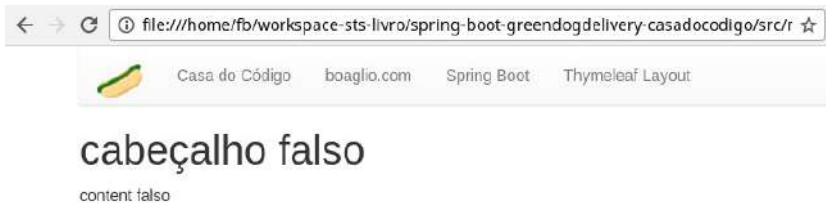


Figura 5.6: Fonte do layout no browser

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>
```

Entretanto, ao subirmos a nova alteração, verificamos que o layout não é aplicado corretamente:



Figura 5.7: Fonte do layout incorreto no browser

Isso acontece porque usamos uma opção do Thymeleaf 3, e o

Spring Boot carrega a versão 2. Para especificarmos uma versão do Thymeleaf, adicionamos nas propriedades do `pom.xml` :

```
<properties>
<thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>
<thymeleaf-layout-dialect.version>2.1.1
</thymeleaf-layout-dialect.version>
</properties>
```

Depois de um restart, o layout funciona corretamente:



Figura 5.8: Fonte do layout correto no browser

5.4 CRUD

As rotinas de CRUD (*Create, Retrieve, Update, Delete*) 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.remove` .

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;

@OneToMany(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 de `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 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 esse 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="endereço" th:text="${cliente.endereço}">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` , iniciado 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 o formulário, `th:object` para relacionar 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 aos campos dos objetos definidos 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 para a 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 esse método:

```

@PostMapping(params = "form")
public ModelAndView create(@Valid Cliente cliente,
BindingResult result, RedirectAttributes redirect) {
  if (result.hasErrors()) { return new ModelAndView
  (CLIENTE_URI + "form", "formErrors", result.getAllErrors()); }
  cliente = this.clienteRepository.save(cliente);
  redirect.addFlashAttribute("globalMessage",
    "Cliente gravado com sucesso");
  return new ModelAndView("redirect:/" + CLIENTE_URI +
    "{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.

5.5 PRÓXIMOS PASSOS

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 aprendeu:

- O conceito de template natural;
- Os principais atributos do Thymeleaf;
- Como o Thymeleaf integra-se aos formulários.

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

DESENVOLVIMENTO PRODUTIVO

Quando chega a hora da entrega, 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, em que o browser carrega automaticamente a página toda vez que o seu código-fonte for alterado.

Para ativar em um projeto Spring Boot, o primeiro passo é adicionar 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.

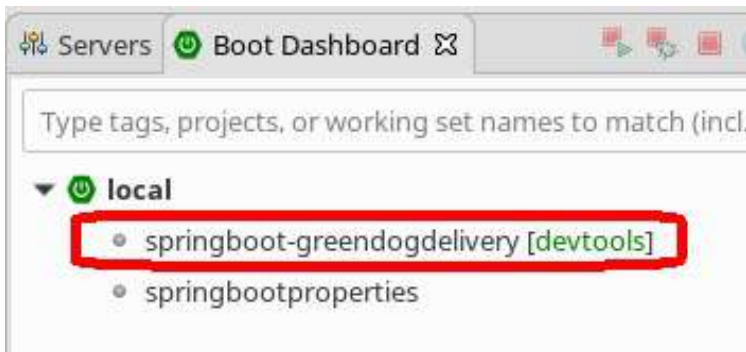


Figura 6.1: Devtools no Boot Dashboard

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:

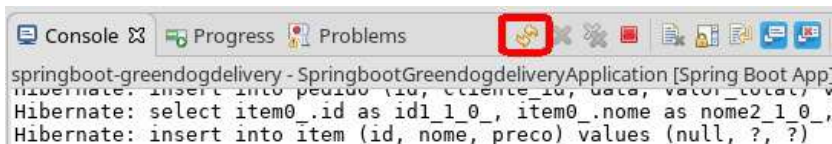


Figura 6.2: Restart forçado do devtools

Com esse módulo ativo, não precisamos nos preocupar em parar e subir a aplicação, o restart será automático. A produtividade aumenta muito e 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 fonte for alterado. Isso é muito útil no desenvolvimento de telas web.

A sua instalação é muito simples. Acesse <http://livereload.com/extensions/>, e instale em seu web browser.

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 ele estiver *preto*, *ligado*.

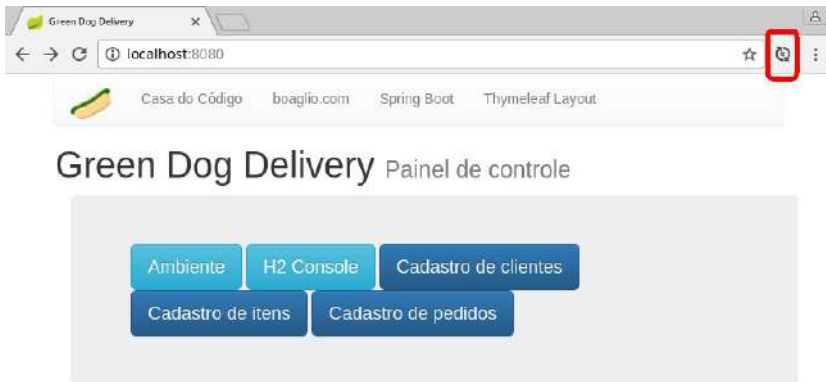


Figura 6.3: LiveReload desligado

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 automaticamente, o browser atualiza a página via LiveReload, mostrando a alteração feita:

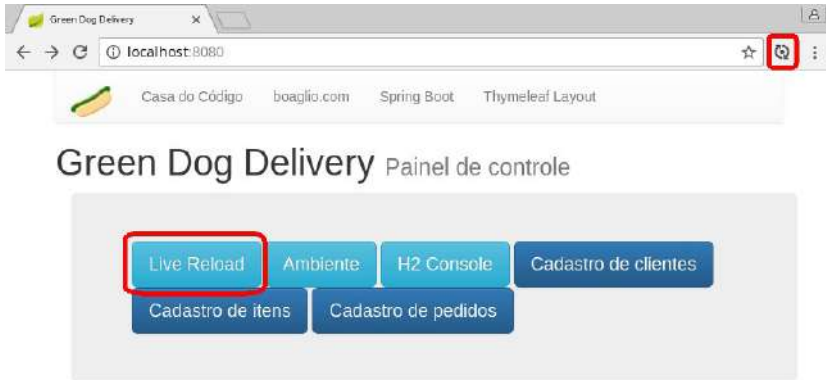


Figura 6.4: LiveReload ativo

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

```
<plugin>
<groupId>com.spotify</groupId>
<artifactId>docker-maven-plugin</artifactId>
```

```

<version>0.4.13</version>
<configuration>
<imageName>greendogdelivery</imageName>
<baseImage>frolvlad/alpine-oraclejdk8:slim</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 docker:build
```

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

```

[INFO] Building image greendogdelivery
Step 1/4 : FROM frolvlad/alpine-oraclejdk8:slim
---> 83b387a3b515
Step 2/4 : ADD /springboot-greendogdelivery-0.0.1-SNAPSHOT.jar //
---> 763c605745fe
Removing intermediate container 96d187e07bca
Step 3/4 : EXPOSE 8080
---> Running in 1a87584c1100
---> f4f26af3d475
Removing intermediate container 1a87584c1100
Step 4/4 : ENTRYPOINT java -jar /springboot-greendogdelivery.jar
---> Running in d00411831f0a
---> 1f51c621c55e
Removing intermediate container d00411831f0a
Successfully built 1f51c621c55e
[INFO] Built greendogdelivery
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----

```


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

```
docker run -p 8080:8080 greendogdelivery:latest
```

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.

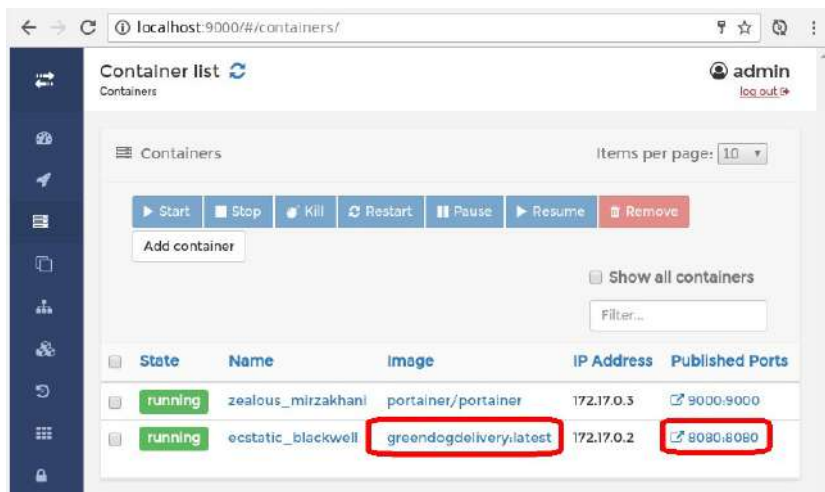


Figura 6.5: Portainer

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

6.4 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- Ativar e usar o devtools em projetos Spring Boot;
- Configurar e usar o LiveReload;
- Gerar uma imagem Docker da aplicação.

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

CUSTOMIZANDO

Como toda aplicação web que Rodrigo fez, ele precisa customizar algumas coisas. As páginas de erro do Spring Boot 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 na identificação no meio de tanto texto do Spring Boot.

Existe uma opção em que, adicionando uma imagem `banner.png` no diretório `src/main/resources`, o Spring Boot automaticamente transforma em ASCII colorido:

7.2 PÁGINAS DE ERRO

A página de erro 404 (de página não encontrada) dá uma mensagem um pouco confusa ao usuário: *Whitelabel Error Page*. O termo 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.



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>
  </div>
</div>
<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:



Figura 7.4: Página não encontrada padrão

A página de erro 500 (500.html) para exibir uma mensagem amigável de sistema indisponível pode ser feita do mesmo jeito, e com o mesmo código. 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 de Spring do inglês 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 gerenciar a sua aplicação quando ela for publicada (estiver em execução).

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

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

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

- `actuator` - Lista todos os links disponíveis;
- `dump` - 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 ;
- `trace` - 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="@{/dump}">dump</a></p>
</td><td><p>Faz um thread dump.</p></td></tr>
<tr><td><p><a href="#" th:href="@{/env}">env</a></p>
</td><td><p>Mostra properties do CE do Spring.</p></td></tr>
<tr><td><p><a href="#" th:href="@{/health}">health</a></p>
</td><td><p>Mostra informações do status da aplicação.</p>
</td></tr>
<tr><td><p><a href="#" th:href="@{/metrics}">metrics</a></p>
</td><td><p>Mostra métricas da aplicação.</p></td></tr>
<tr><td><p><a href="#" th:href="@{/mappings}">mappings</a></p>
</td><td><p>Exibe os caminhos dos @RequestMapping.</p>
</td></tr>
<tr><td><p><a href="#" th:href="@{/trace}">trace</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.



Figura 7.5: Links do Actuator

A opção `health` mostra informações da saúde do sistema, como a situação do banco de dados e do espaço em disco:

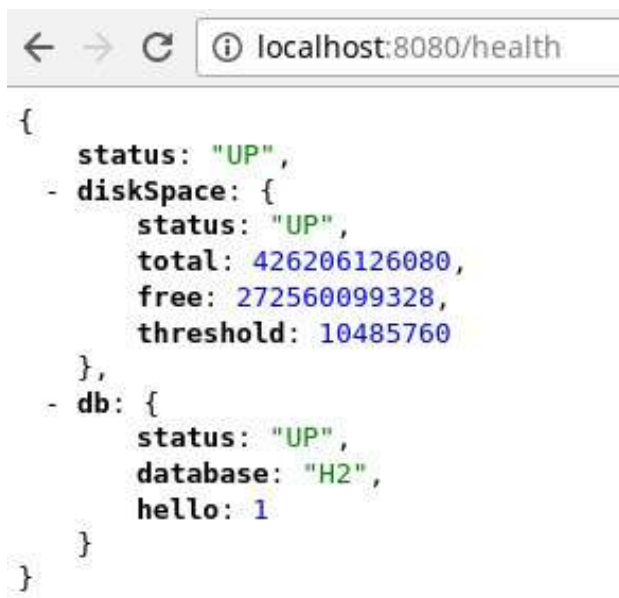
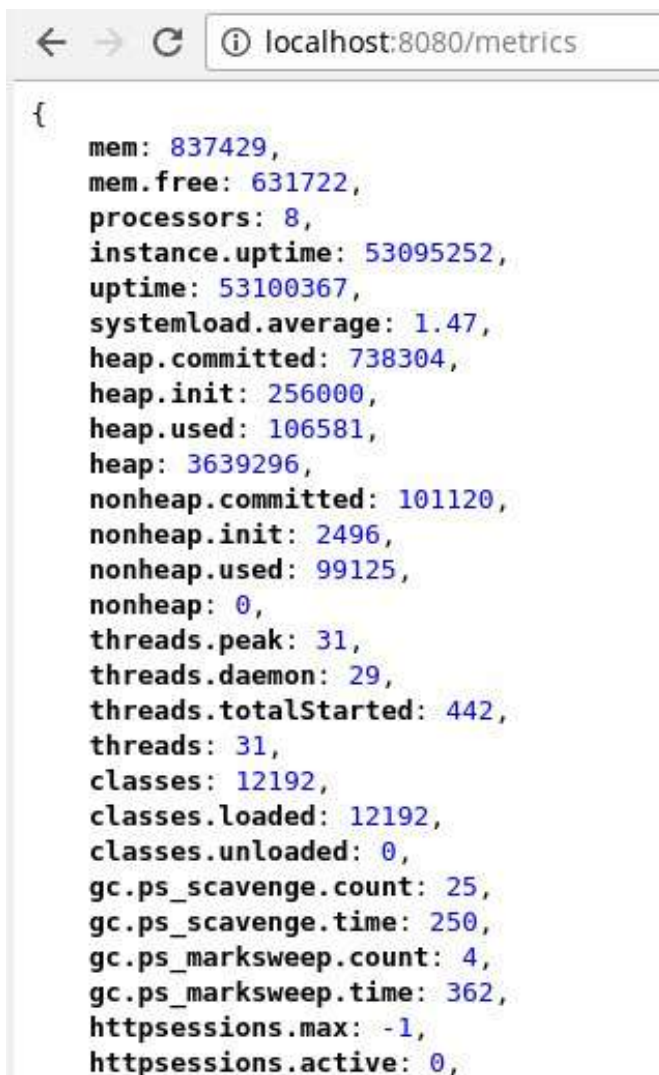


Figura 7.6: Opção health do Actuator

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

Vale a pena destacar os counters e gaugers. Os counters sempre somam alguma coisa, como por exemplo, resposta de status HTTP 200 na raiz (`counter.status.200.root: 2`); já os gaugers medem o tempo de resposta da última chamada, que pode aumentar ou diminuir, como a chamada de `/metrics` que demora 1 ms (`gauge.response.metrics: 1`).



```
{
  mem: 837429,
  mem.free: 631722,
  processors: 8,
  instance.uptime: 53095252,
  uptime: 53100367,
  systemload.average: 1.47,
  heap.committed: 738304,
  heap.init: 256000,
  heap.used: 106581,
  heap: 3639296,
  nonheap.committed: 101120,
  nonheap.init: 2496,
  nonheap.used: 99125,
  nonheap: 0,
  threads.peak: 31,
  threads.daemon: 29,
  threads.totalStarted: 442,
  threads: 31,
  classes: 12192,
  classes.loaded: 12192,
  classes.unloaded: 0,
  gc.ps_scavenge.count: 25,
  gc.ps_scavenge.time: 250,
  gc.ps_marksweep.count: 4,
  gc.ps_marksweep.time: 362,
  httpsessions.max: -1,
  httpsessions.active: 0,
```

Figura 7.7: Opção health do Actuator

7.4 PRÓXIMOS PASSOS

Os fontes do projeto estão em

<https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/customizando>.

Certifique-se de que aprendeu a:

- Customizar um banner na saída de console do Spring Boot;
- Criar páginas customizadas de erros HTTP;
- Ativar e usar as páginas do Spring Actuator.

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

EXPONDO A API DO SEU SERVIÇO

O sistema do Rodrigo já tem um cadastro de clientes, itens e pedidos, que podem ser usados pelo administrador. O que resta agora é a tela de novos pedidos, que será feita 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 é muito usada pela comunidade. Para uma boa performance, foi escolhido o framework AngularJS 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, servindo como um agente que mapeia e limita a arquitetura ReST. De acordo com o modelo de maturidade ReST

de Richardson, o HATEOAS é o nível máximo que introduz a descoberta de serviços, fornecendo uma maneira de fazer um protocolo autodocumentado.

Resumindo, o HATEOAS é uma arquitetura mais completa do que o 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 online.

Vamos ver um exemplo do HATEOAS começando com alteração no nosso repositório `ItemRepository`. Adicionaremos a anotação `RepositoryRestController`, que define o acesso aos serviços na URI `/itens`.

```
@RepositoryRestController
(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>
</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:

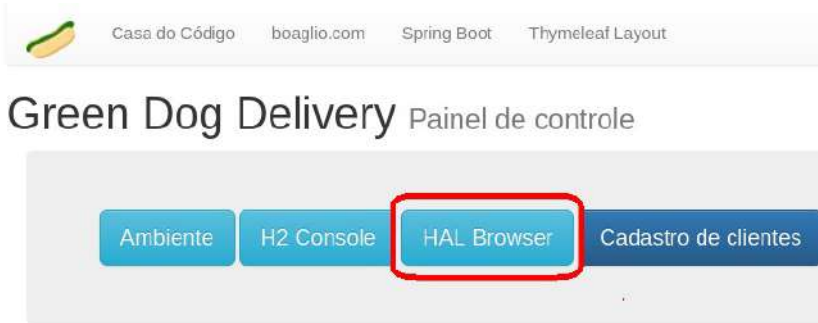


Figura 8.1: Link do HAL Browser

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

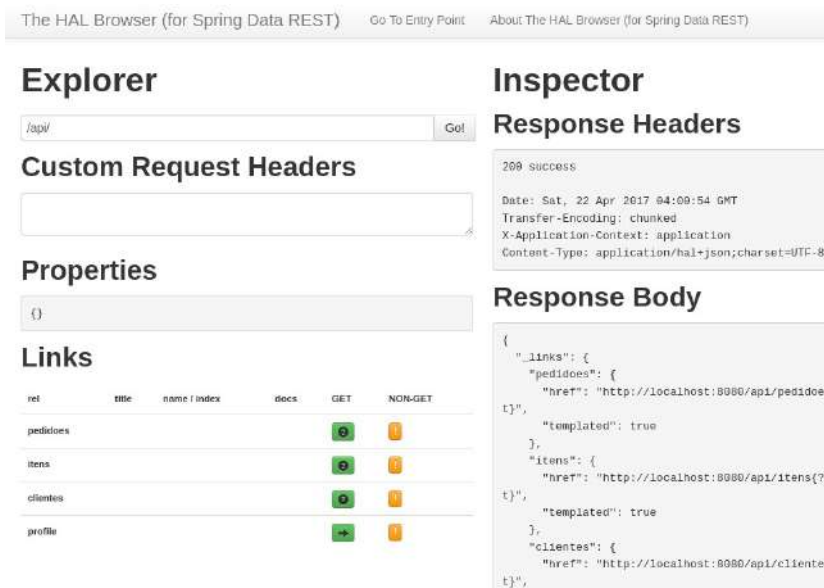


Figura 8.2: Serviços do HAL Browser

No HATEOAS, cada serviço sempre retorna além de suas informações, os endereços para o próprio serviço (*self*) e mais serviços.

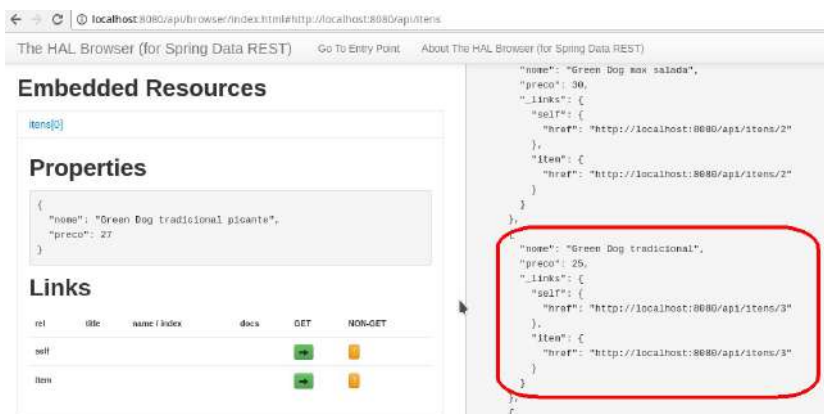


Figura 8.3: Serviço de itens no HAL Browser

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 o parâmetro para `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
extends RepositoryRestConfigurerAdapter
{

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

8.2 ANGULAR ACESSANDO REST

Rodrigo ficou empolgado depois que descobriu que também não precisava escrever os serviços ReST, agora só precisava consumir de um novo sistema de fazer 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>
```

E 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 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">
    
    Cardápio
</label>
</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="isItemSelecioneado(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}}</strong>  {{mensagem}}
</div>
<div class="alert alert-warning" ng-show="idPedido!=null">
Valor do pedido: <strong>{{valorTotal}}</strong> 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` , inicialmente declarando no formato do AngularJS e registrando a função `carregarItens` :

```

var app = angular.module("delivery",["checkboxlist-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:

```

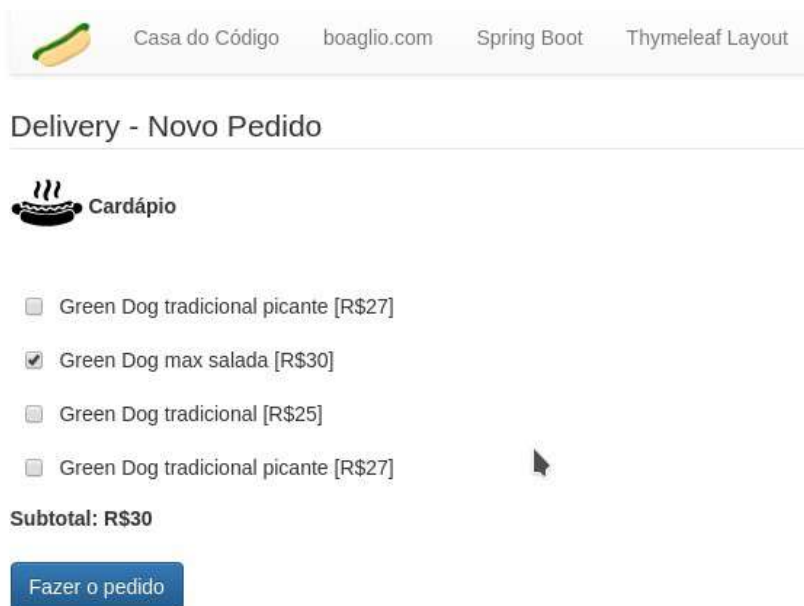
$scope.fazerPedido = function(pedidoItens) {
    $scope.message ="";
    var pedidoStr="";
    var prefixo="";
    for (var i=0; i< $scope.pedidoItens.length; i++) {
        pedidoStr+=prefixo+$scope.pedidoItens[i].id;
        prefixo=",";
    }
    $scope.urlPedido="/rest/pedido/novo/2/"+pedidoStr;
    $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.isItemSelecionado = 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]
- ☐ Green Dog tradicional picante [R\$27]

Subtotal: R\$30

[Fazer o pedido](#)

Figura 8.4: 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]
- ☐ Green Dog tradicional picante [R\$27]

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

Pronto, o nosso sistema de delivery está pronto para receber pedidos online.

8.3 PRÓXIMOS PASSOS

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

Certifique-se de que aprendeu:

- Uso e aplicação do HATEOAS;
- Uso e ativação do HAL Browser;
- Integração do Spring Data HATEOAS com o sistema

em AngularJS.

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

TESTANDO SUA APP

O sistema do Rodrigo tem seus serviços ReST, e ele tem uma equipe pronta para trabalhar com eles. Mas como garantir que eles estão funcionando corretamente? Como garantir que, depois de uma alteração, algum serviço não alterou algum resultado de algum outro serviço? 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 por `SpringRunner`. Com ela, conseguimos subir o *applicattion context* do Spring e fazer funcionar todas as injeções de dependência de seu teste. Sem ela, será necessário instanciar manualmente cada objeto envolvido no teste.

Podemos usar o `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 um.

```
@Test
```

```
public void buscaClientesCadastrados() {
```

```
Page<Cliente> clientes =
```

```
    this.repository.findAll(new PageRequest(0, 10));
```

```
assertThat(clientes.getTotalElements()).isGreaterThan(1L);
```

```
}
```

Rodar o teste inteiro demorou pouco mais de seis segundos, mas 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.

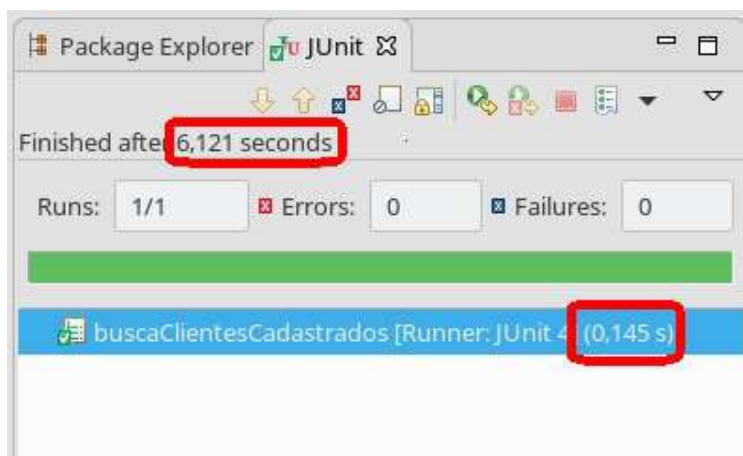


Figura 9.1: Rodando o teste de busca

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.findByNome("Fernando");
    assertThat(clienteNaoEncontrado).isNull();

    Cliente cliente =
        this.repository.findByNome("Fernando Boaglio");
    assertThat(cliente).isNotNull();
    assertThat(cliente.getNome()).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 opção 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 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 cadastraNovoPedido() 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"))));
}

```

Com esses testes, conseguimos testar integralmente o serviço de pedidos e a busca de itens.

9.3 PRÓXIMOS PASSOS

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

Certifique-se de que aprendeu a:

- Rodar testes unitários com Spring Boot;
- Rodar testes de integração com Spring Boot;

No próximo capítulo, veremos como disponibilizar a nossa aplicação em diferentes ambientes, com diferentes servidores.

EMPACOTANDO E DISPONIBILIZANDO SUA APP

Depois de validar todos os testes, Rodrigo está confiante em subir sua aplicação para o servidor e começar a usar em produção. Neste cenário, existe a dúvida de qual opção usar. Vamos entender as opções existentes.

10.1 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-1.0.0-SNAPSHOT.jar
```

E o seu JAR está pronto para uso, e pode ser publicado no servidor de produção.

10.2 JAR EXECUTÁVEL

Usamos manualmente o comando `java -jar` para subir o sistema. Temos a desvantagem de, se acontecer algum reboot do servidor, o sistema ficará fora do ar.

Para solucionar esse problema, é sugerido que use 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` essa 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 copiar o JAR `gdd.jar` 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=143

[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 exibir o status do serviço:

```
# systemctl status gdd.service
● gdd.service - gdd
   Loaded: loaded (/etc/systemd/system/gdd.service;
   Active: active (running) since Sun 2017-04-09
 Main PID: 11880 (gdd.jar)
    Tasks: 34 (limit: 4915)
   Memory: 141.4M
      CPU: 44.370s
   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, baseados 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`.

Editamos o arquivo `/var/run/springboot/gdd.conf` , adicionando a seguinte linha:

```
JAVA_OPTS=-Xmx1024M
```

Ao reiniciar o serviço, as novas configurações serão aplicadas.

Outros sistemas operacionais

Não existe um suporte oficial para Windows, mas existe uma alternativa de usar o *Windows Service Wrapper* (<https://github.com/kohsuke/winsw>). Veja mais em: <https://github.com/snicoll-scratches/spring-boot-daemon>.

Para Mac OS X, também não existe suporte. Entretanto, há uma alternativa interessante que é o *Launchd* (<http://launchd.info>).

10.3 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
35M    target/green-dog-delivery-1.0.0-SNAPSHOT.war
```

Com isso, o pacote WAR pode ser publicado em um Tomcat, Jetty, JBoss ou WildFly, sem nenhum problema.

10.4 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 entender de que maneira podemos alterar o contêiner usado.

A implementação de contêiner web padrão é *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*:



Figura 10.1: 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>
```

```
[spring-boot-gradle-plugin> GradleDogDeliveryApplication (Spring Boot App) /usr/lib/jvm/java-8-jdk/bin/java (9 de abr de 2017 11:09:14)
INFO 364 --- restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "[/dump][ /dump.json], methods=[GET], produces=[application/vnd
INFO 364 --- restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "[/autoconfig][ /autoconfig.json], methods=[GET], produces=[app
INFO 364 --- restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "[/loggers[name=*]], methods=[GET], produces=[application/vnd
INFO 364 --- restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "[/loggers[name=*]], methods=[POST], consumes=[application/vnd
INFO 364 --- restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "[/loggers][ /loggers.json], methods=[GET], produces=[applicat
INFO 364 --- restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "[/health][ /health.json], methods=[GET], produces=[applicat
WARN 364 --- restartedMain] o.s.b.d.s.OptionalLiveReloadServer : Mapped "[/mappings][ /mappings.json], methods=[GET], produces=[applicat
INFO 364 --- restartedMain] o.s.c.support.DefaultLifecycleProcessor : Unable to start LiveReload server
INFO 364 --- restartedMain] o.s.c.e.a.AnnotationMethodHandlerAdapter : Registering beans for JMX exposure on startup
INFO 364 --- restartedMain] o.s.c.e.u.UndertowEmbeddedServletContainer : Starting Undertow on port(s) 8080 (http)
INFO 364 --- restartedMain] o.s.c.e.u.UndertowEmbeddedServletContainer : Undertow started on port(s) 8080 (http)
Jados...
```

Figura 10.2: Subindo com o servidor Undertow

Existem dezenas de configurações de contêiner (todas começam com `server.`) para colocar no arquivo `application.properties` : algumas específicas para Tomcat, Jetty ou Undertow, e outras 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.

10.5 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.

Instalação

A instalação dele é bem simples. É só baixar o arquivo `bin.zip` do site <http://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. 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
```

Temos facilmente a página pronta para testes no browser.



```
{
  "java.runtime.name": "Java(TM) SE Runtime Environment",
  "java.protocol.handler.pkgs": "org.springframework.boot.loader",
  "sun.boot.library.path": "/usr/lib/jvm/java-8-jdk/jre/lib/amd64",
  "java.vm.version": "25.121-b13",
  "java.vm.vendor": "Oracle Corporation",
  "java.vendor.url": "http://java.oracle.com/",
  "path.separator": ":",
  "java.vm.name": "Java HotSpot(TM) 64-Bit Server VM",
  "file.encoding.pkg": "sun.io",
  "user.country": "BR",
  "sun.java.launcher": "SUN_STANDARD"
}
```

Figura 10.3: Teste com Spring CLI

10.6 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- Deploy da aplicação como pacote JAR;
- Deploy da aplicação como serviço;
- Deploy da aplicação como pacote WAR;
- Deploy de protótipos simples com Spring CLI.

No próximo capítulo, vamos ver como usar o Spring Boot na nuvem.

SUBINDO NA NUVEM

Depois 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.

11.1 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 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 para cada pedido fará um teste com o método `envioAtivo` 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);
```

Vamos criar 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 será instanciada apenas quando esse perfil estiver ativo. Já a classe `DevNotificacaoConfig` possui a anotação `@Profile("!prod")`, indicando que 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 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:

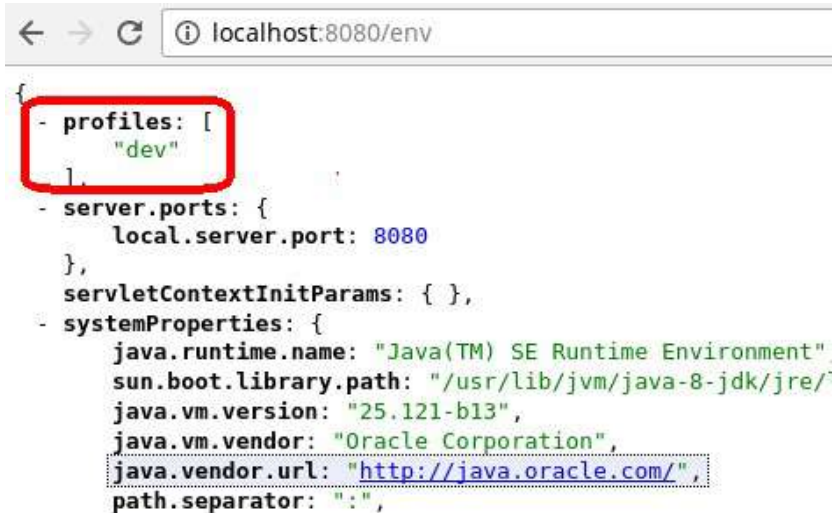


Figura 11.1: Ambiente utilizado profile de dev

Para testar o perfil `prod` e sobrescrever o valor definido em `application.properties`, use a opção de linha de comando:

```
mvn spring-boot:run
-Drun.arguments="--spring.profiles.active=prod"
```

Agora o log de pedido mudou. Veja:

```
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 usado:



```
{
  - profiles: [
    "prod"
  ],
  - server.ports: {
    local.server.port: 8080
  },
  - commandLineArgs: {
    spring.profiles.active: "prod"
  },
  servletContextInitParams: { },
  - systemProperties: {
    java.runtime.name: "Java(TM) SE Runtime Environment",
    sun.boot.library.path: "/usr/lib/jvm/java-8-jdk/jre/lib/amd64",
    java.vm.version: "25.121-b13",
    java.vm.vendor: "Oracle Corporation",
    java.vendor.url: "http://java.oracle.com/",
```

Figura 11.2: 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 por exemplo, profile de diferentes bancos de dados ou diferentes servidores.

Usando profiles para o Heroku

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 informações de conexão certamente não serão iguais. Portanto, o profile ajuda a aplicação 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
# permite acesso ao Actuator
management.security.enabled=false
```

Criaremos também o arquivo para trabalhar na nuvem `application-heroku.properties` :

```
# jpa
spring.jpa.show-sql=true
spring.datasource.url=${CLEARDB_DATABASE_URL}
spring.datasource.maxActive=10
spring.datasource.maxIdle=5
spring.datasource.minIdle=2
spring.datasource.initialSize=5
spring.datasource.removeAbandoned=true
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
# permite acesso ao Actuador
management.security.enabled=false
```

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` :

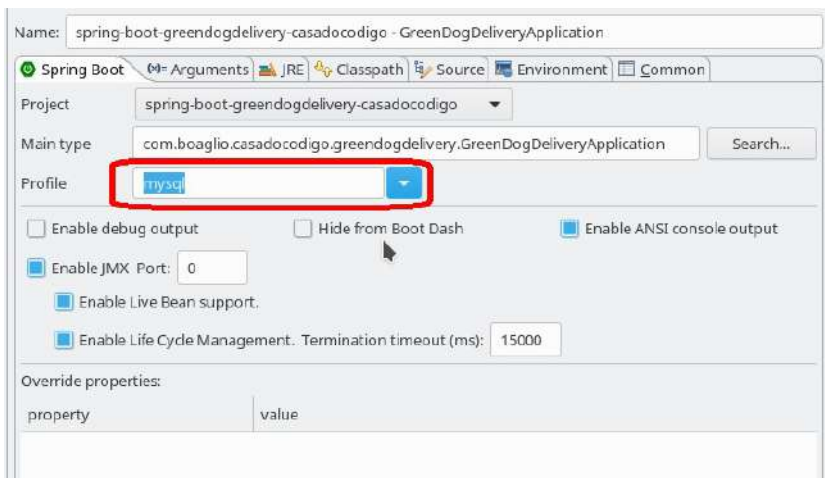


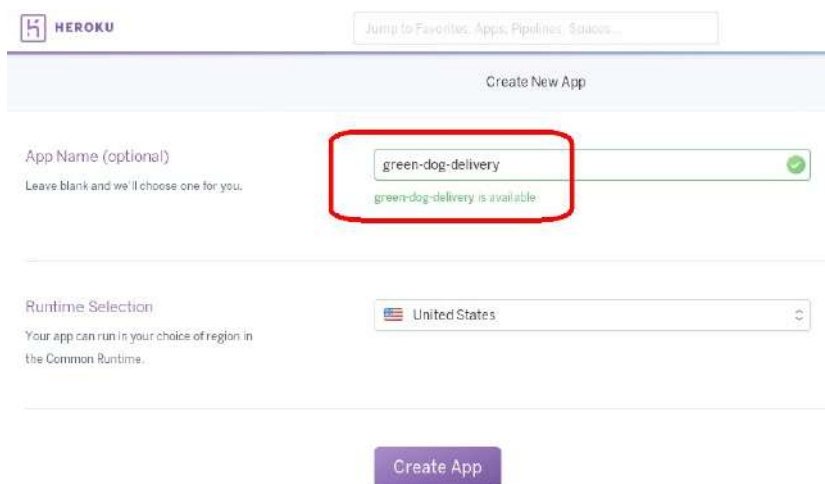
Figura 11.3: Escolher perfil no Eclipse

11.2 HEROKU

Heroku é uma empresa que oferece serviço de hospedagem na nuvem (*Platform-as-a-Service*, ou PaaS) suportando várias

linguagens de programação. Existem outras opções no mercado, mas a maioria delas é paga. Então, vamos usar o Heroku que oferece uma opção gratuita bem simples de usar.

Depois de criamos uma conta gratuita no Heroku (<https://www.heroku.com>), podemos criar uma nova aplicação `green-dog-delivery`, ou outro nome que desejarmos.



The screenshot shows the Heroku 'Create New App' interface. At the top, there's a navigation bar with the Heroku logo and a search bar. Below that, a light blue banner says 'Create New App'. The main form has two sections. The first section is 'App Name (optional)' with a subtext 'Leave blank and we'll choose one for you.' The input field contains 'green-dog-delivery' and is highlighted with a red rectangle. Below the input field, a green message says 'green-dog-delivery is available'. The second section is 'Runtime Selection' with a subtext 'Your app can run in your choice of region in the Common Runtime.' The dropdown menu is set to 'United States'. At the bottom, there is a purple 'Create App' button.

Figura 11.4: Nova aplicação

Em seguida, adicionamos um recurso (`resource`) de banco de dados MySQL, pois o nosso sistema precisa armazenar os dados em algum lugar.

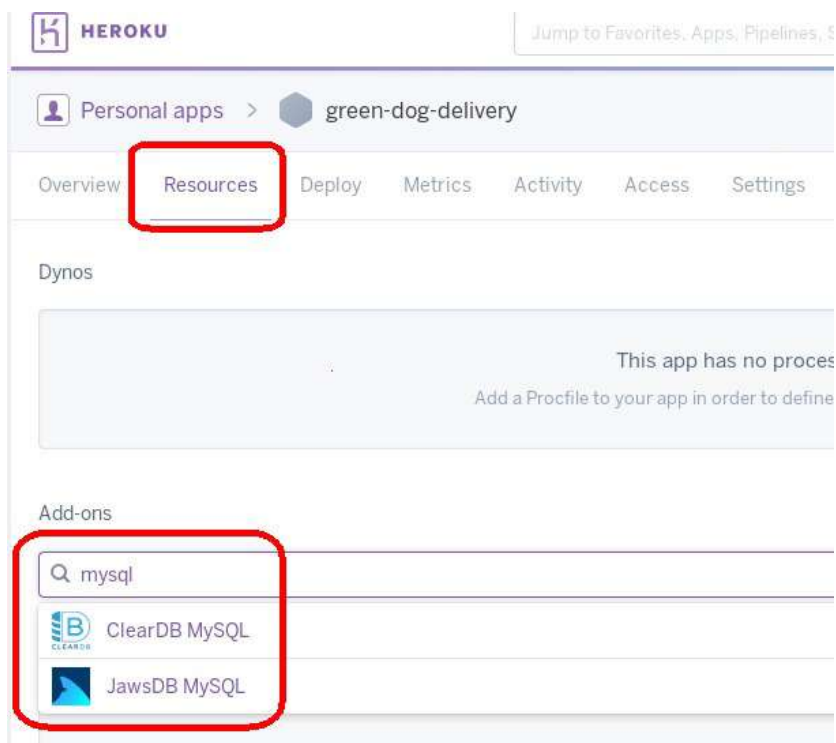


Figura 11.5: Novo recurso de banco de dados

E escolhemos a opção gratuita:

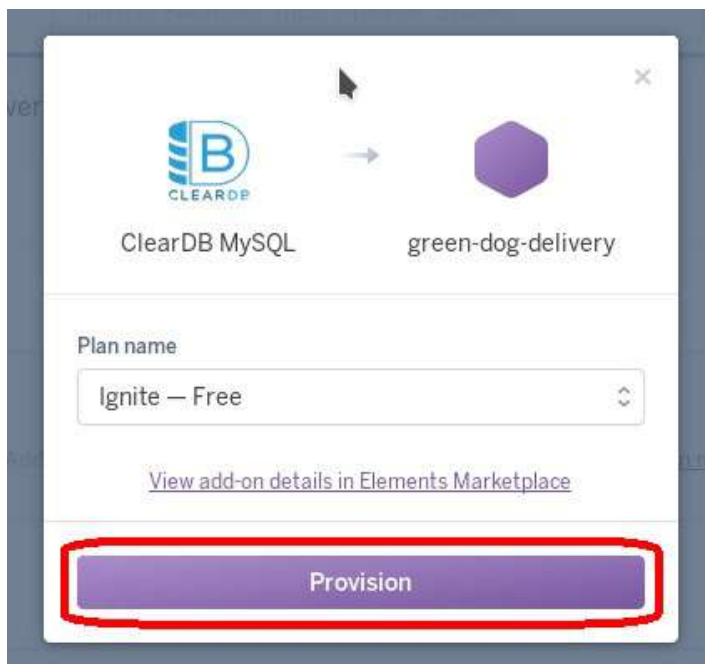


Figura 11.6: Opção gratuita do MySQL

Em seguida, obtemos as informações de acesso remoto ao banco de dados na opção `Settings` , dentro da variável `CLEARDB_DATABASE_URL` .

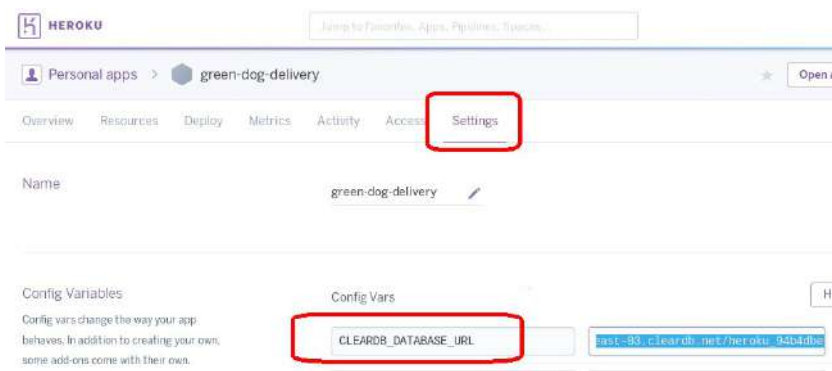


Figura 11.7: Configurações da aplicação

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, vamos usar as fontes de nenhum repositório remoto, e sim do nosso diretório local da máquina.

Vamos fazer 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 `Procfile` 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-1.0.0-SNAPSHOT.jar
-Dspring.profiles.active=heroku
```

Em seguida, vamos adicionar ao `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
Free dyno hours quota remaining this month: 1728h 18m (88%)
For more information on dyno sleeping and how to upgrade, see:
https://devcenter.heroku.com/articles/dyno-sleeping
=== web (Free): java -Dserver.port=$PORT $JAVA_OPTS -jar
target/green-dog-delivery-1.0.0-SNAPSHOT.jar
-Dspring.profiles.active=heroku (1)
web.1: up 2017/04/10 17:03:22 +0000 (~ -10340s ago)
# heroku logs
```

Com o deploy no ar, podemos testar no browser:

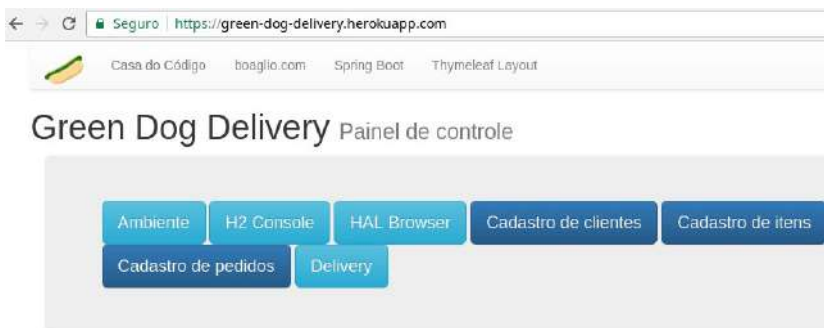


Figura 11.8: Sistema rodando na nuvem

Podemos acompanhar também pelo site da Heroku como está o nosso sistema. É só logar e entrar nos detalhes da aplicação criada:

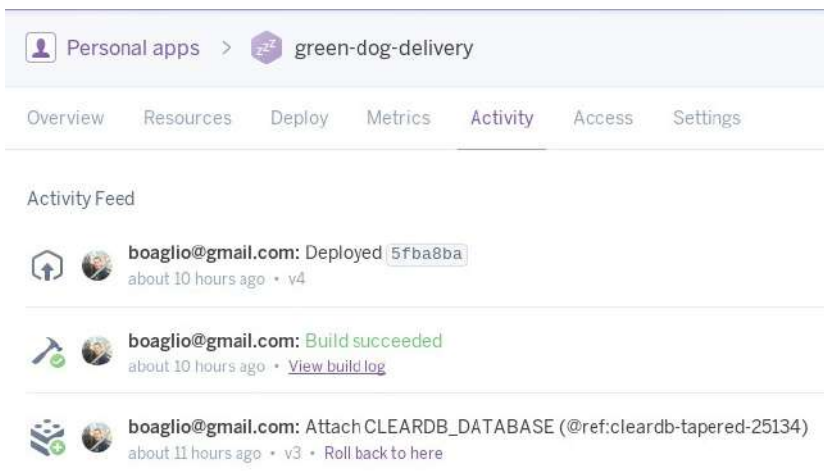


Figura 11.9: Atividades do sistema

11.3 PRÓXIMOS PASSOS

Os fontes do projeto estão em

<https://github.com/boaglio/spring-boot-greendogdelivery-casadocodigo/tree/heroku>.

Certifique-se de que aprendeu a:

- Trabalhar com profiles (perfis) no Spring Boot;
- Subir aplicação na nuvem da Heroku.

No próximo capítulo, veremos um pouco sobre microsserviços e Spring Cloud.

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 aumentar.

O cenário atual de Rodrigo tem uma alta demanda de um lado, com os pedidos online, e uma baixa de outro, com o controle de estoque nos cadastros.

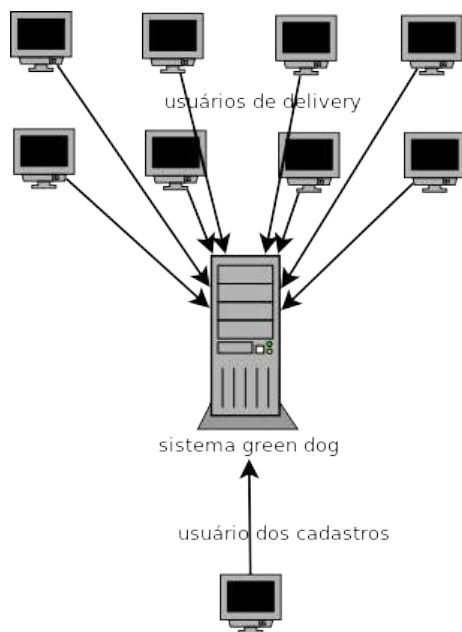


Figura 12.1: Standalone

Hoje temos apenas um sistema que cuida dos cadastros e faz os pedidos também. Com o conceito do *microservices*, podemos separar os serviços existentes em dois e crescer apenas os que realmente necessitam. Portanto, no atual cenário 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 diferenciar também no pacote de deploy para reduzir o consumo de recursos.

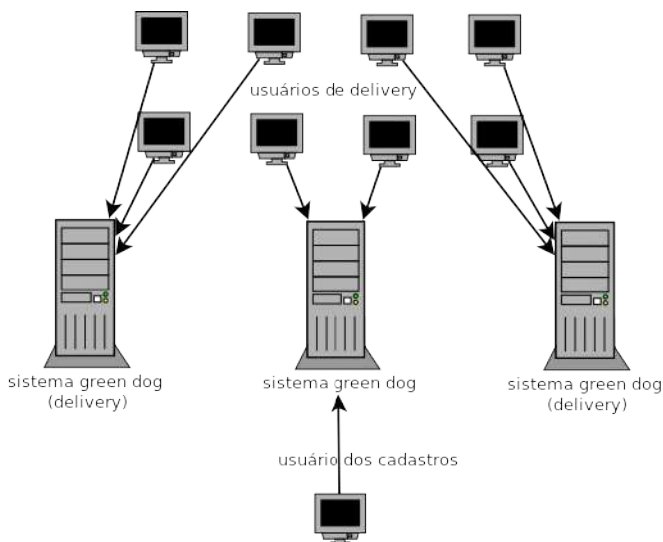


Figura 12.2: Microservices

Spring Cloud com Netflix OSS

Netflix é empresa líder mundial de streaming de séries e filmes pela internet, e também uma das maiores usuárias das facilidades que o Spring Framework oferece. Ela também gerencia os seus microsserviços usando soluções que ela mesma desenvolveu e abriu os fontes, chamando de Netflix Open Source Software (<https://netflix.github.io>).

Existem diversos projetos, e cada um tem uma parte na responsabilidade de manter os serviços no ar com alta disponibilidade e qualidade. O nosso exemplo exhibe uma pequena parte desses projetos, e está um pouco longe de ser um exemplo ideal de implementação do Spring Cloud.

12.1 NOSSO EXEMPLO

Vamos criar um exemplo bem simples usando poucas opções do Spring Cloud, mas suficientes para garantir uma alta disponibilidade do sistema do Rodrigo. Além do projeto atual, usaremos outros três projetos:

1. *Config Server* — Responsável por distribuir configurações entre os nós do cluster;
2. *Eureka* — Responsável por registrar os serviços do cluster;
3. *Zuul Gateway* — Faz o roteamento entre os serviços disponíveis.

Para facilitar o entendimento, vamos rodar tudo localmente na mesma máquina.

- <http://localhost:8080> — Zuul Gateway
- <http://localhost:8888> — Config Server
- <http://localhost:8761> — Eureka Server
- <http://localhost:8081> — Green Dog Delivery (cluster)
- <http://localhost:8082> — Green Dog Delivery (cluster)
- <http://localhost:8083> — Green Dog Delivery (cluster)

O nosso ambiente ficará assim:

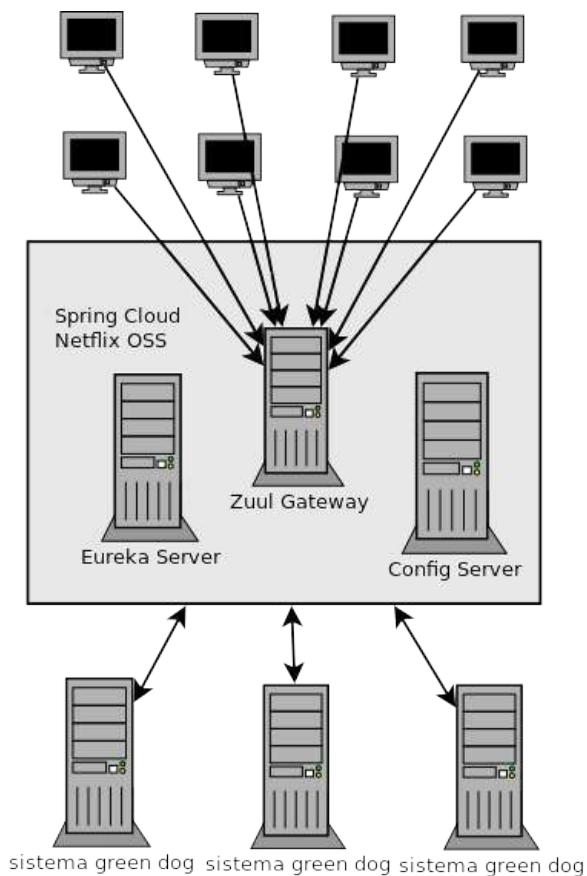


Figura 12.3: Green Dog Cloud

Com o nosso exemplo funcionando, poderemos:

- Derrubar e subir qualquer nó do cluster, sem afetar a aplicação;
- Monitorar os serviços no ar;
- Publicar novas configurações online, sem nenhum restart.

No universo do Rodrigo, isso faz muito sentido. Colocar uma oferta relâmpago no seu site tem de ser algo muito rápido e não deve derrubar nenhum serviço para isso acontecer. No nosso exemplo, essa mudança é orquestrada pelo Spring Cloud e tudo estará pronto, alterando um arquivo texto fora da aplicação e fazendo apenas um commit.

Parece mágica? Quase. O que precisamos fazer é deixar a aplicação pronta para receber isso, adaptando-a para o Spring Cloud.

Os outros serviços (Zuul, Config e Eureka) estão prontos e serão apenas projetos separados com apenas uma classe e um arquivo de configuração. Vamos detalhar cada um dos três serviços, em seguida mostrar as alterações em nosso sistema, e finalmente mostrar o nosso cluster em ação!

12.2 CONFIG SERVER

As configurações sempre são versionadas em algum lugar (local ou remoto), e publicadas aos nós do cluster pelo *Config Server*. No nosso exemplo, são propriedades do arquivo `greendogdelivery.properties`, localizadas dentro do projeto `config-repo`.

Vamos copiar o diretório `config-repo` para uma pasta qualquer para o nosso teste, e depois vamos iniciar um versionamento do Git:

```
# cp config-repo /home/fb
# cd /home/fb/config-repo
# git init
# git add .
```

```
# git commit -m "oferta hot dog"
```

O projeto config server precisa ter as dependências no pom.xml :

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Brixton.SR1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

A classe inicial do Spring Boot para iniciar o Config Server:

```
@EnableDiscoveryClient
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

O arquivo `bootstrap.yml` dentro da pasta `src/main/resources`:

```
server:
  port: 8888
spring:
  application:
    name: config
  cloud:
    config:
      server:
        git:
          uri: /home/fb/config-repo
```

Com isso, temos o projeto rodando na porta 8888 e apontando para o repositório git local. Este é ideal para os nossos testes em um repositório local, pois, em um ambiente de produção, o Config Server estaria em um repositório remoto.

12.3 EUREKA SERVER

A palavra *eureka* foi supostamente pronunciada pelo cientista grego Arquimedes (287 a.C. – 212 a.C.), quando descobriu como resolver um complexo dilema. No nosso contexto, ela procura todos os serviços do Spring Cloud e permite que se registrem nela para tornarem-se disponíveis para uso.

O projeto Eureka Server tem as dependências no `pom.xml` semelhantes ao Config Server:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

A classe inicial do Spring Boot para iniciar o Eureka Server precisa da anotação `EnableEurekaServer` :

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

O arquivo `bootstrap.yml` dentro da pasta `src/main/resources` :

```
server:
  port: 8761
spring:
  application:
    name: server
  cloud:
    config:
      uri: http://localhost:8888
```

Com isso, temos o projeto rodando na porta 8761 e apontando para o Config Server.

12.4 ZUUL GATEWAY

No filme *Caça Fantasmas*, de 1984, Zuul era o porteiro do portal de Gozer, um deus antigo que queria trazer todos os demônios de seu universo para a cidade de New York. No nosso contexto, ela que receberá todas as requisições e enviará aos servidores disponíveis.

O projeto Zuul Gateway tem as dependências no pom.xml semelhantes ao Config Server:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

A classe inicial do Spring Boot para iniciar o Zuul Gateway com a anotação `EnableZuulProxy` :

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

O arquivo `bootstrap.yml` dentro da pasta `src/main/resources` :

```
server:
  port: 8080
spring:
  application:
    name: gateway
  cloud:
    config:
      uri: http://localhost:8888
zuul:
  routes:
    greendogdelivery:
      path: /**
```

```
stripPrefix: false
```

Com isso, temos o projeto rodando na porta 8080 e apontando para o Config Server. Além disso, ele configura a rota dos serviços do greendogdelivery para o contexto /* .

No geral, o que foi feito até aqui foi apenas criar classes iniciais do Spring Boot, cada uma com uma anotação diferente (EnableConfigServer , EnableZuulProxy e EnableEurekaServer), suas configurações, para então subirmos tudo. A mágica está exatamente nessa facilidade de apenas colocar anotações e configurações, e os servidores ficam prontos funcionando e se comunicando automaticamente.

12.5 AJUSTE NO SISTEMA ATUAL

Precisamos adequar o sistema ao Spring Cloud, colocar os serviços para ajudar a publicar a nossa oferta, e uma opção de debug para visualizarmos o servidor de origem.

Começamos adicionando as dependências do pom.xml :

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

A classe inicial do Spring Boot para iniciar o Zuul Gateway:

```
@EnableEurekaClient
@SpringBootApplication
@ComponentScan(basePackages = "com.boaglio")
public class GreenDogApplication {
```



```

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

```

A anotação `EnableEurekaClient` define esse sistema como um candidato para registrar o Eureka Server, e o `ComponentScan` reduz a busca dos beans da aplicação toda para apenas a package `com.boaglio`. 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;

```

O Config Server enviará novos valores para esses dois parâmetros dinamicamente. Para essa classe suportar isso, é preciso adicionar a anotação `RefreshScope`.

Para ajudar no debug de nossos testes, criaremos um serviço para exibir de qual servidor vem a solicitação:

```
@GetMapping("/servidor")
@ResponseBody
public String server(HttpServletRequest request) {
    return request.getServerName()+":"+request.getServerPort();
}
```

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;
    });
};
```

Finalmente, teremos 3 arquivos `properties` distintos para subir a aplicação em portas diferentes. O primeiro deles é:

```
# 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=none
#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
# permite acesso ao Actuator
management.security.enabled=false
# cloud
spring.application.name=greendogdelivery
spring.cloud.config.uri=http://localhost:8888
server.port=8081
```

Os outros dois arquivos restantes têm apenas a propriedade da porta diferente: 8082 e 8083 .

O arquivo `greendogdelivery.properties` existente no repositório local, e inicialmente não tem nenhuma oferta publicada:

```
#
# sem oferta
#
debug=0
mensagem=nenhuma

#
# com oferta
#
#mensagem=Compre 1 hot dog e ganhe 1 suco de laranja !
#debug=1
```

12.6 TESTANDO NOSSO CLUSTER

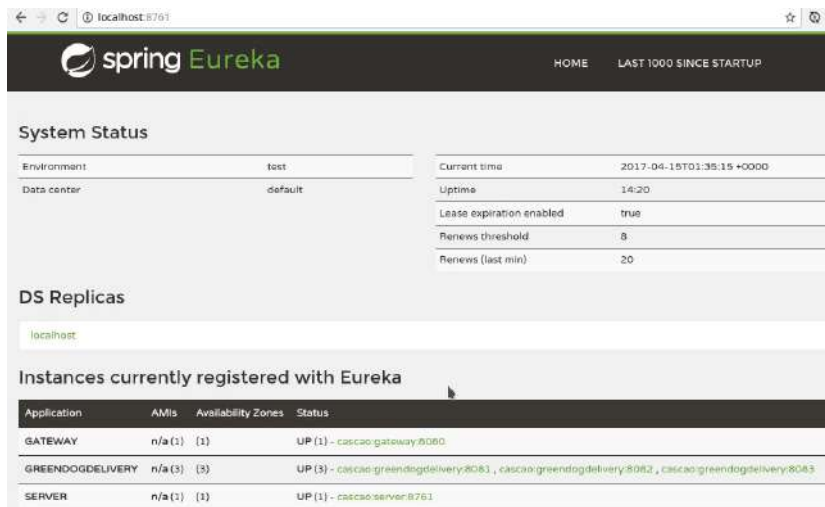
Vamos subir todos os processos no ar. Subimos o Zuul Gateway, o Config Server e o Eureka Server com:

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

Já a nossa aplicação subiremos indicando um arquivo properties diferente:

```
#mvn spring-boot:run
-Drun.arguments="--spring.profiles.active=cliente1"
#mvn spring-boot:run
-Drun.arguments="--spring.profiles.active=cliente2"
#mvn spring-boot:run
-Drun.arguments="--spring.profiles.active=cliente3"
```

Com tudo no ar, podemos acessar o Eureka no browser e ver os serviços online, em <http://localhost:8761>.



The screenshot shows the Spring Eureka dashboard in a web browser. The page has a dark header with the Spring Eureka logo and navigation links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into sections: System Status, DS Replicas, and Instances currently registered with Eureka.

System Status

Environment	test
Data center	default

Current time	2017-04-15T01:35:15 +0000
Uptime	14:20
Lease expiration enabled	true
Renews threshold	8
Renews (last min)	20

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
GATEWAY	n/a (1)	(1)	UP (1) - cascao.gateway:8080
GREENDOGDELIVERY	n/a (3)	(3)	UP (3) - cascao.greendogdelivery:8081, cascao.greendogdelivery:8082, cascao.greendogdelivery:8083
SERVER	n/a (1)	(1)	UP (1) - cascao.server:8761

Figura 12.4: Eureka dashboard

Veja o serviço que mostra o servidor de origem:

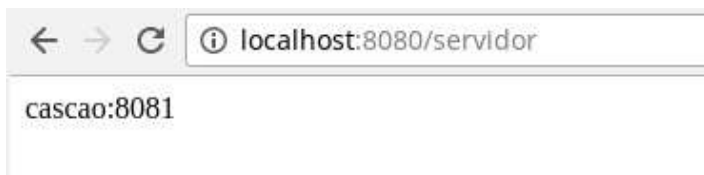


Figura 12.5: Servidor do sistema

Note que, ao fazermos o reload de página, sempre aparece outro servidor. Pode existir um servidor no ar ou três, o serviço sempre volta algum valor.

Ao chamarmos o serviço em <http://localhost:8080/oferta>, são exibidos valores vazios:



Figura 12.6: Servidor sem oferta

Em seguida, alteramos o arquivo `greendogdelivery.properties` para ativarmos a oferta:

```
#
# sem oferta
#
#debug=0
#mensagem=nenhuma
```

```
#
# com oferta
#
mensagem=Compre 1 hot dog e ganhe 1 suco de laranja !
debug=1
```

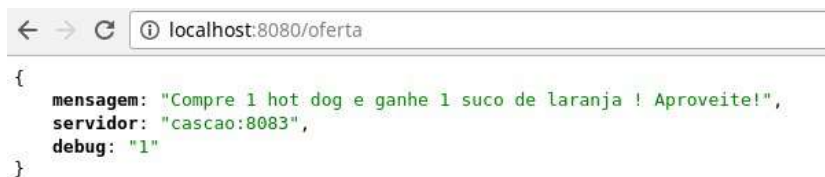
E depois, commitamos:

```
# cd /home/fb/config-repo
# git add .
# git commit -m "oferta hot dog"
```

Para finalmente publicar esses novos valores no cluster, precisamos solicitar um refresh das novas configurações, fazendo uma chamada POST em <http://localhost:8080/refresh>. Isso pode ser feito com um aplicativo cliente ReST, ou via linha de comando usando o curl :

```
# curl -X POST http://localhost:8080/refresh
["debug", "mensagem"]
```

Agora o serviço de oferta retorna a seguinte mensagem:



```
{
  mensagem: "Compre 1 hot dog e ganhe 1 suco de laranja ! Aproveite!",
  servidor: "cascao:8083",
  debug: "1"
}
```

Figura 12.7: Servidor com oferta

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

Delivery - Novo Pedido



Cardápio

☐ 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 12.8: Servidor com oferta

Pronto! Rodrigo pode comemorar! Facilmente ele conseguirá publicar novas ofertas.

E se uma nova máquina entrar no cluster, ela será atualizada também automaticamente. Então, em um evento como Black Friday, ele pode subir dez máquinas, ou mais, da mesma maneira que subiu essas três, sem nenhuma configuração adicional.

12.7 PRÓXIMOS PASSOS

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 aprendeu:

- Princípios básicos do Spring Cloud e Netflix OSS;
- Como configurar e usar Config Server;
- Como configurar e usar Eureka Server;
- Como configurar e usar Zuul Gateway;
- Como configurar e usar a opção de RefreshScope para obter parâmetros dinâmicos.

No próximo capítulo, vamos descobrir onde saber mais sobre o Spring Boot.

INDO ALÉM

O ecossistema do Spring é enorme, e o Spring Boot é apenas parte dele.

13.1 REFERÊNCIAS

A principal referência é a documentação oficial:
<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>.

Outras referências são:

- Projetos da Netflix (diversos projetos usando Spring Boot e Spring Cloud para soluções de persistência, monitoração, big data e segurança) — <https://netflix.github.io>
- Exemplos e tutoriais de Spring — <http://www.baeldung.com>

Apresentações

- Dezenas de apresentações de Spring com profissionais da Pivotal, committers dos projetos e cases de sucesso — <https://www.infoq.com/springone>

- Microserviços - Dan Woods — <https://www.infoq.com/br/articles/boot-microservices>

Podcasts

- Pivotal Conversations, feita 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://domineiospring.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 - Spring Developer — <https://twitter.com/starbuxman>
- Engine Thymeleaf — <https://twitter.com/thymeleaf>

Livros

- *Spring MVC: Domine o principal framework web Java*, de Alberto Souza — <https://www.casadocodigo.com.br/products/livro-spring-mvc>
- *Vire o jogo com Spring Framework*, de Henrique Lobo Weissmann — <https://www.casadocodigo.com.br/products/livro-spring-framework>

13.2 SISTEMA ENTREGUE

No final do livro, Rodrigo conseguiu criar um sistema de cadastro de itens, pedidos e clientes. Depois disso, criou facilmente um esquema de monitorar a aplicação em produção, seguido de testes unitários e de integração.

Em seguida, ele colocou o sistema na nuvem e aprendeu a como quebrar o sistema em microsserviços, separando apenas a parte de pedidos para alta disponibilidade. O próximo passo de Rodrigo é seguir essas referências para se aprofundar no Spring Boot e acompanhar as novidades do framework.

13.3 CONSIDERAÇÕES FINAIS

Espero que tenha gostado do livro, acompanhe o fórum da Casa do Código para dúvidas e sugestões (<http://forum.casadocodigo.com.br/>).

Obrigado pela leitura!

13.4 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>.

APÊNDICE A — STARTERS

Os starters são configurações pré-definidas 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 trabalhosa tarefa.

A documentação oficial está em <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#usando-boot-starter>.

Starters web

Nome	Descrição
<i>spring-boot-starter-web</i>	Starter para construir aplicações web (incluindo RESTful) usando Spring MVC. Usa Tomcat como contêiner padrão.
<i>spring-boot-starter-web-services</i>	Starter para usar com Spring Web Services.
<i>spring-boot-starter-jersey</i>	Starter para construir aplicações web RESTful usando JAX-RS e Jersey. Uma alternativa para <i>spring-boot-starter-web</i> .
<i>spring-boot-starter-hateoas</i>	Starter para construir aplicações web hypermedia RESTful, com Spring MVC e Spring HATEOAS.
<i>spring-boot-starter-data-rest</i>	Starter para expor repositórios Spring Data via REST, usando Spring Data REST.

Starters auxiliares

Nome	Descrição
<i>spring-boot-starter</i>	Core starter, incluindo configuração automática, log e YAML.
<i>spring-boot-starter-test</i>	Starter para testar as aplicações do Spring Boot com JUnit, Hamcrest e Mockito.
<i>spring-boot-starter-integration</i>	Starter para usar com Spring Integration.
<i>spring-boot-starter-mail</i>	Starter para usar com Java Mail e Spring Framework's email sending support.
<i>spring-boot-starter-mobile</i>	Starter para construir aplicações web usando Spring Mobile.
<i>spring-boot-starter-validation</i>	Starter para usar com Java Bean Validation com Hibernate Validator.
<i>spring-boot-starter-websocket</i>	Starter para construir aplicações WebSocket usando Spring Framework WebSocket.
<i>*spring-boot-starter-aop</i>	Starter para Spring AOP e AspectJ.
<i>spring-boot-starter-security</i>	Starter para usar com Spring Security.
<i>spring-boot-starter-batch</i>	Starter para usar com Spring Batch.
<i>spring-boot-starter-cache</i>	Starter para usar com o suporte de cache do Spring Framework.
<i>spring-boot-starter-cloud-connectors</i>	Starter para usar com Spring Cloud Connectors que simplifica conectar a serviços na nuvem, como Cloud Foundry e Heroku.

Starters de template

Nome	Descrição
<i>spring-boot-starter-thymeleaf</i>	Starter para construir aplicações web MVC usando Thymeleaf views.
<i>spring-boot-starter-</i>	Starter para construir aplicações web MVC usando

<i>groovy-templates</i>	Groovy Templates views.
<i>spring-boot-starter-mustache</i>	Starter para construir aplicações web MVC usando Mustache views.
<i>spring-boot-starter-freemarker</i>	Starter para construir aplicações web MVC usando FreeMarker views.

Starters de mensageria

Nome	Descrição
<i>spring-boot-starter-artemis</i>	Starter para JMS usando Apache Artemis.
<i>spring-boot-starter-activemq</i>	Starter para JMS messaging usando Apache ActiveMQ.
<i>spring-boot-starter-amqp</i>	Starter para usar com Spring AMQP e Rabbit MQ.

Starters de gerenciamento de dados

Nome	Descrição
<i>spring-boot-starter-jdbc</i>	Starter para usar com JDBC, com o Tomcat JDBC connection pool.
<i>spring-boot-starter-data-jpa</i>	Starter para usar com Spring Data JPA, com Hibernate.
<i>spring-boot-starter-jooq</i>	Starter para usar com jOOQ para acessar banco de dados SQL. É uma alternativa para o <i>spring-boot-starter-data-jpa</i> ou <i>spring-boot-starter-jdbc</i> .
<i>spring-boot-starter-data-couchbase</i>	Starter para usar com Couchbase e Spring Data Couchbase.
<i>spring-boot-starter-data-solr</i>	Starter para usar com Apache Solr e Spring Data Solr.
<i>spring-boot-starter-data-mongodb</i>	Starter para usar com MongoDB e Spring Data MongoDB.
<i>spring-boot-starter-data-</i>	Starter para usar com Elasticsearch e Spring Data Elasticsearch.

<i>elasticsearch</i>	
<i>spring-boot-starter-data-redis</i>	Starter para usar com Redis, e Spring Data Redis com Jedis client.
<i>spring-boot-starter-data-gemfire</i>	Starter para usar com GemFire e Spring Data GemFire.
<i>spring-boot-starter-data-cassandra</i>	Starter para usar com Cassandra e Spring Data Cassandra.
<i>spring-boot-starter-data-neo4j</i>	Starter para usar com Neo4j e Spring Data Neo4j.
<i>spring-boot-starter-jta-atomikos</i>	Starter para JTA usando Atomikos.
<i>spring-boot-starter-jta-bitronix</i>	Starter para JTA transactions usando Bitronix.
<i>spring-boot-starter-jta-narayana</i>	Starter para usar com Narayana JTA.

Starters de redes sociais

Nome	Descrição
<i>spring-boot-starter-social-facebook</i>	Starter para usar com Spring Social Facebook.
<i>spring-boot-starter-social-linkedin</i>	Starter para usar com Spring Social LinkedIn.
<i>spring-boot-starter-social-twitter</i>	Starter para usar com Spring Social Twitter.

Starters de produção

Nome	Descrição
<i>spring-boot-starter-actuator</i>	Starter para usar com Spring Boot Actuator, que disponibiliza opções de monitoração e gerenciamento da sua aplicação.

<i>spring-boot-starter-remote-shell</i>	Starter para usar com o CRaSH remote shell, para monitorar e gerenciar sua aplicação sobre SSH.
---	---

Starters de servidor de aplicação

Nome	Descrição
<i>spring-boot-starter-undertow</i>	Starter para usar com Undertow como contêiner servlet embutido. Uma alternativa para <i>spring-boot-starter-tomcat</i> .
<i>spring-boot-starter-jetty</i>	Starter para usar com Jetty como contêiner servlet embutido. Uma alternativa para <i>spring-boot-starter-tomcat</i> .

Nome	Descrição
<i>spring-boot-starter-tomcat</i>	Starter para usar com Tomcat como contêiner servlet embutido (padrão).

Starters de log

Nome	Descrição
<i>spring-boot-starter-logging</i>	Starter para logging usando Logback (padrão).
<i>spring-boot-starter-log4j2</i>	Starter para usar com Log4j2. Uma alternativa para <i>spring-boot-starter-logging</i> .

Starters descontinuados

Nome	Descrição	Observação
<i>spring-boot-starter-ws</i>	Starter para usar com Spring Web Services.	Descontinuado na versão 1.4 no lugar de <i>spring-boot-starter-web-services</i> .
<i>spring-boot-starter-hornetq</i>	Starter para JMS usando HornetQ.	Descontinuado na versão 1.4 no lugar de <i>spring-boot-starter-artemis</i> .

<i>spring-boot-starter-redis</i>	Starter para usar com Redis, com Spring Data Redis e Jedis client.	Descontinuado na versão 1.4 no lugar de <i>spring-boot-starter-data-redis</i> .
<i>spring-boot-starter-velocity</i>	Starter para construir aplicações web MVC usando Velocity views.	Descontinuado na versão 1.4.

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/common-application-properties.html>.

Categoria	Subcategoria	Total
ACTUATOR PROPERTIES	ENDPOINTS	77
ACTUATOR PROPERTIES	ENDPOINTS CORS CONFIGURATION	6
ACTUATOR PROPERTIES	HEALTH INDICATORS	15
ACTUATOR PROPERTIES	INFO CONTRIBUTORS	5
ACTUATOR PROPERTIES	JMX ENDPOINT	4
ACTUATOR PROPERTIES	JOLOKIA	1
ACTUATOR PROPERTIES	MANAGEMENT HTTP SERVER	22
ACTUATOR PROPERTIES	METRICS EXPORT	13
ACTUATOR PROPERTIES	REMOTE SHELL	18
ACTUATOR PROPERTIES	TRACING	1

CORE PROPERTIES	ADMIN	2
CORE PROPERTIES	AOP	2
CORE PROPERTIES	APPLICATION SETTINGS	3
CORE PROPERTIES	AUTO-CONFIGURATION	1
CORE PROPERTIES	BANNER	8
CORE PROPERTIES	Email	9
CORE PROPERTIES	HAZELCAST	1
CORE PROPERTIES	IDENTITY	2
CORE PROPERTIES	INTERNATIONALIZATION	5
CORE PROPERTIES	JMX	3
CORE PROPERTIES	LOGGING	9
CORE PROPERTIES	OUTPUT	1
CORE PROPERTIES	PID FILE	2
CORE PROPERTIES	PROFILES	2
CORE PROPERTIES	PROJECT INFORMATION	2
CORE PROPERTIES	SENDGRID	5
CORE PROPERTIES	SPRING CACHE	10
CORE PROPERTIES	SPRING CONFIG	2
CORE PROPERTIES	SPRING CORE	1
DATA PROPERTIES	ATOMIKOS	38
DATA PROPERTIES	BITRONIX	63
DATA PROPERTIES	CASSANDRA	17
DATA PROPERTIES	COUCHBASE	14
DATA PROPERTIES	DAO	1
DATA PROPERTIES	DATA COUCHBASE	3
DATA PROPERTIES	DATA REDIS	1

DATA PROPERTIES	DATA REST	9
DATA PROPERTIES	DATASOURCE	24
DATA PROPERTIES	ELASTICSEARCH	4
DATA PROPERTIES	EMBEDDED MONGODB	5
DATA PROPERTIES	FLYWAY	24
DATA PROPERTIES	H2 Web Console	4
DATA PROPERTIES	JEST	7
DATA PROPERTIES	JOOQ	1
DATA PROPERTIES	JPA	12
DATA PROPERTIES	JTA	3
DATA PROPERTIES	LIQUIBASE	12
DATA PROPERTIES	MONGODB	10
DATA PROPERTIES	NARAYANA	13
DATA PROPERTIES	NEO4J	7
DATA PROPERTIES	REDIS	13
DATA PROPERTIES	SOLR	3
DEVTOOLS PROPERTIES	DEVTOOLS	9
DEVTOOLS PROPERTIES	REMOTE DEVTOOLS	8
INTEGRATION PROPERTIES	ACTIVEMQ	11
INTEGRATION PROPERTIES	ARTEMIS	12
INTEGRATION PROPERTIES	HORNETQ	12
INTEGRATION PROPERTIES	JMS	6
INTEGRATION PROPERTIES	RABBIT	42

INTEGRATION PROPERTIES	SPRING BATCH	5
SECURITY PROPERTIES	SECURITY	30
WEB PROPERTIES	EMBEDDED SERVER CONFIGURATION	242