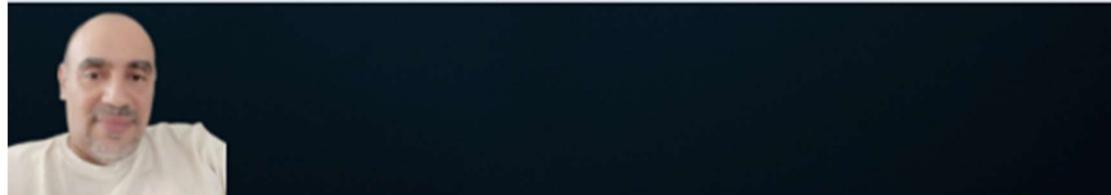




Um exemplo com
Java, Spring Boot e
Thymeleaf



Sumário

1	Introdução	5
2	Criando o Banco	7
3	Criando o aplicativo no IntelliJ Ultimate IDE	9
3.1	Configure o Arquivo de Propriedades	13
4	Definindo o Modelo	16
4.1	Validação back-end	18
5	Interface Repository	20
6	Camada Service	22
7	Controller	25
7.1	Descrição dos métodos da classe AppController	26
8	Spring Boot Application Class	29
9	Usando o Thymeleaf	30
9.1	Arquivo alert.html	31
9.2	Arquivo index.html	32
9.3	Arquivo visualização.html	35
9.4	Arquivo new_product.html	36
10	Hospedando a aplicação no Heroku	42
10.1	Criando conta no Heroku	42
10.2	Instalar o Heroku CLI	46
10.3	Testando a instalação do Heroku CLI	48
10.4	Preparando o Banco de Dados	51
10.5	Preparando o Aplicativo Java para Nuvem	53
10.6	Enviando a aplicação para a nuvem	55
10.7	Logando no heroku login	55
10.8	Passos Finais	57

Índice de Figuras

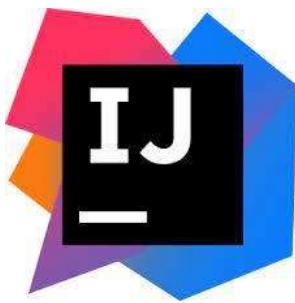
1	Introdução	5
2	Criando o Banco	7
3	Criando o aplicativo no IntelliJ Ultimate IDE	9
3.1	Configure o Arquivo de Propriedades	13
4	Definindo o Modelo	16
4.1	Validação back-end	18
5	Interface Repository	20
6	Camada Service	22
7	Controller	25
7.1	Descrição dos métodos da classe AppController	26
8	Spring Boot Application Class	29
9	Usando o Thymeleaf	30
9.1	Arquivo alert.html	31
9.2	Arquivo index.html	32
9.3	Arquivo visualização.html	35
9.4	Arquivo new_product.html	36
10	Hospedando a aplicação no Heroku	42
10.1	Criando conta no Heroku	42
10.2	Instalar o Heroku CLI	46
10.3	Testando a instalação do Heroku CLI	48
10.4	Preparando o Banco de Dados	51
10.5	Preparando o Aplicativo Java para Nuvem	53
10.6	Enviando a aplicação para a nuvem	55
10.7	Logando no heroku login	55
10.8	Passos Finais	57

1 Introdução

Neste tutorial sobre o Spring Boot, você aprenderá a desenvolver um aplicativo Java para Web que gerencia informações em um banco de dados – com operações CRUD padrão: **Criar, Recuperar, Atualizar e Deletar** dados do Banco. Utilizamos as seguintes tecnologias:

- **Spring Boot:** tecnologia que permite o desenvolvimento rápido de aplicativos com padrões de projetos para reduzir o código. O Spring Boot também nos ajuda a criar um aplicativo *Web Java* executável com facilidade.
- **Spring Web:** simplifica a codificação da camada do controlador (controller).
- **Spring Data JPA:** simplifica a codificação da camada de acesso a dados.
- **Spring Bean Validation:** é o padrão para implementar a lógica de validação no ecossistema Java. Está bem integrado com Spring e Spring Boot.
- **Hibernate:** é usado como um framework ORM – implementação de JPA. Não há mais código JDBC padrão.
- **Thymeleaf:** simplifica a codificação da camada de visualização. Não usamos tags JSP e JSTL nativas.
- **BootStrap:** framework front-end que fornece estruturas de CSS para a criação de sites e aplicações responsivas de forma rápida e simples. Usaremos o webjars (dependências adicionadas ao pom.xml) para que o bootstrap esteja local na aplicação.
- **MySQL:** Sistema Gerenciador de Banco de Dados que utilizaremos para este exemplo.
- **Heroku:** Trata-se de uma PaaS (Plataforma como um Serviço) que permite hospedagem, configuração, testagem e publicação de projetos virtuais na nuvem. Usaremos para hospedagem do site em produção.

Para o desenvolvimento do projeto, usamos IntelliJIDEA, JDK 8 e Maven e o Spring Boot.



2 Criando o Banco

Nota: Opcional a criação, se utilizar o arquivo de propriedades de configuração indicando para criar o banco, caso não exista.

Criando o Banco de Dados MySQL (banco: sales). Usando o MySQL Workbench crie o schema “sales” e a tabela “product”. A Figura 1 apresenta o banco de dados “sales”.

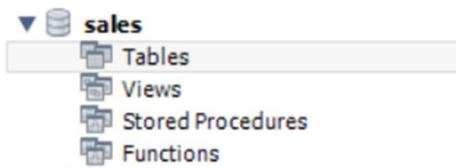


Figura 1 - Banco [Sales]

Com o banco criado, crie a tabela product como mostra a Figura 1 - Banco [Sales].

A screenshot of the MySQL Workbench 'product - Table' configuration window. At the top, it shows 'sales - Schema' and 'product - Table'. The 'Table Name' is set to 'product', 'Schema' is 'sales', and 'Engine' is 'InnoDB'. Under 'Charset/Collation', 'Default Charset' and 'Default Collation' are selected. There is a 'Comments' field below. The main area shows the table structure with five columns: 'id' (INT, PK, NN, AI), 'name' (VARCHAR(45), NN), 'brand' (VARCHAR(45), NN), 'madein' (VARCHAR(45), NN), and 'price' (FLOAT, NN). Each column has its properties listed in a grid below it, such as 'B', 'UN', 'ZF', and 'G'.

Figura 2 - Tabela [product]

Se preferir pode executar o seguinte script MySQL para criar a tabela de produtos:

```
CREATE TABLE `product` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `brand` varchar(45) NOT NULL,
  `madein` varchar(45) NOT NULL,
```

```
    `price` float NOT NULL,  
    PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

3 Criando o aplicativo no IntelliJ Ultimate IDE

Para a criação do aplicativo, abra o IntelliJ e selecione New Project para iniciar um novo projeto. Preencha escolhendo o caminho de sua preferência e o restante como mostra a Figura 3.

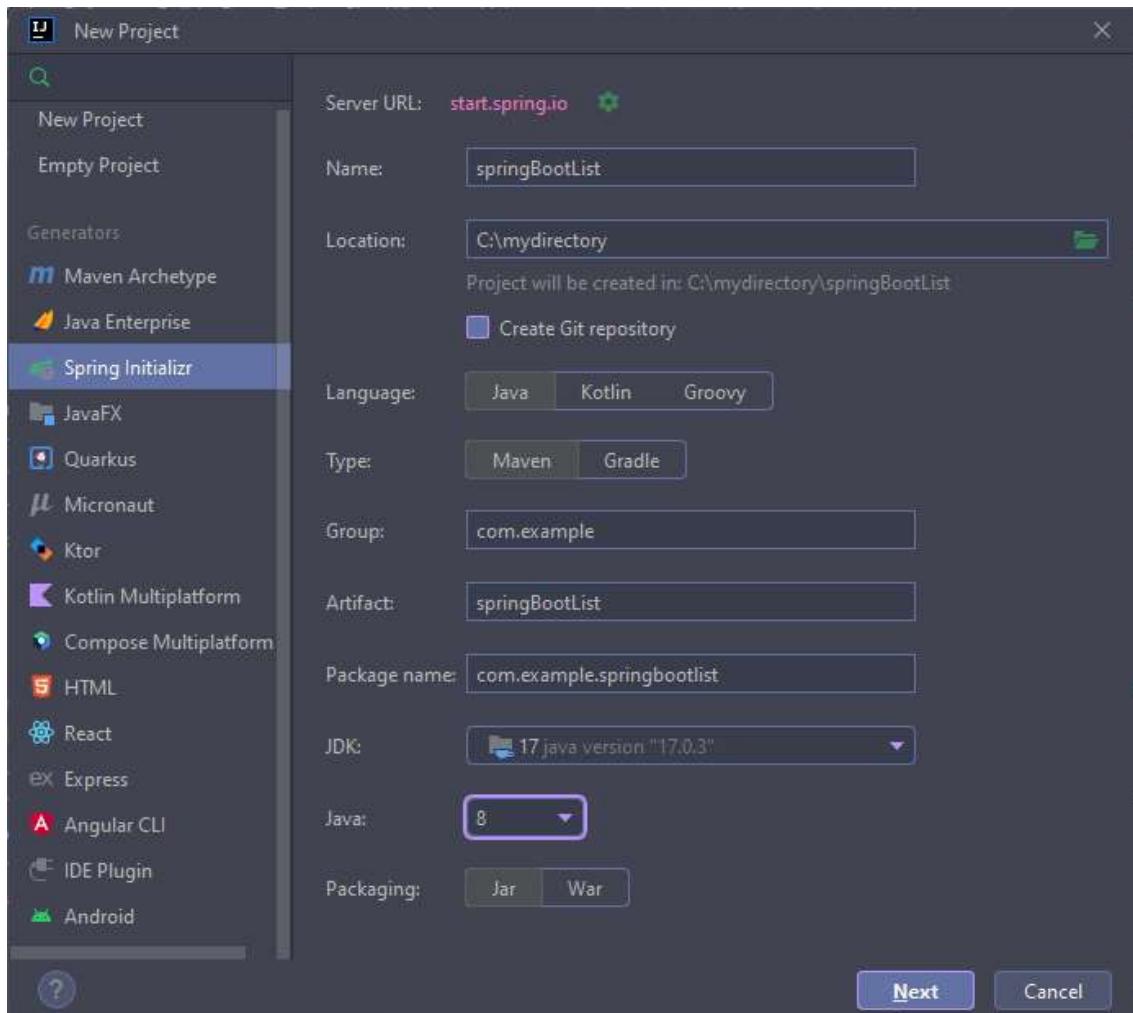


Figura 3 - Janela [New Project]

Clique sobre o botão **Next** para avançar à próxima janela. Nela você deve especificar as dependências. Observe a área marcada da Figura 4. São as dependências que devem ser adicionadas ao seu projeto. **[Atente para a versão do Spring Boot a ser utilizada].**

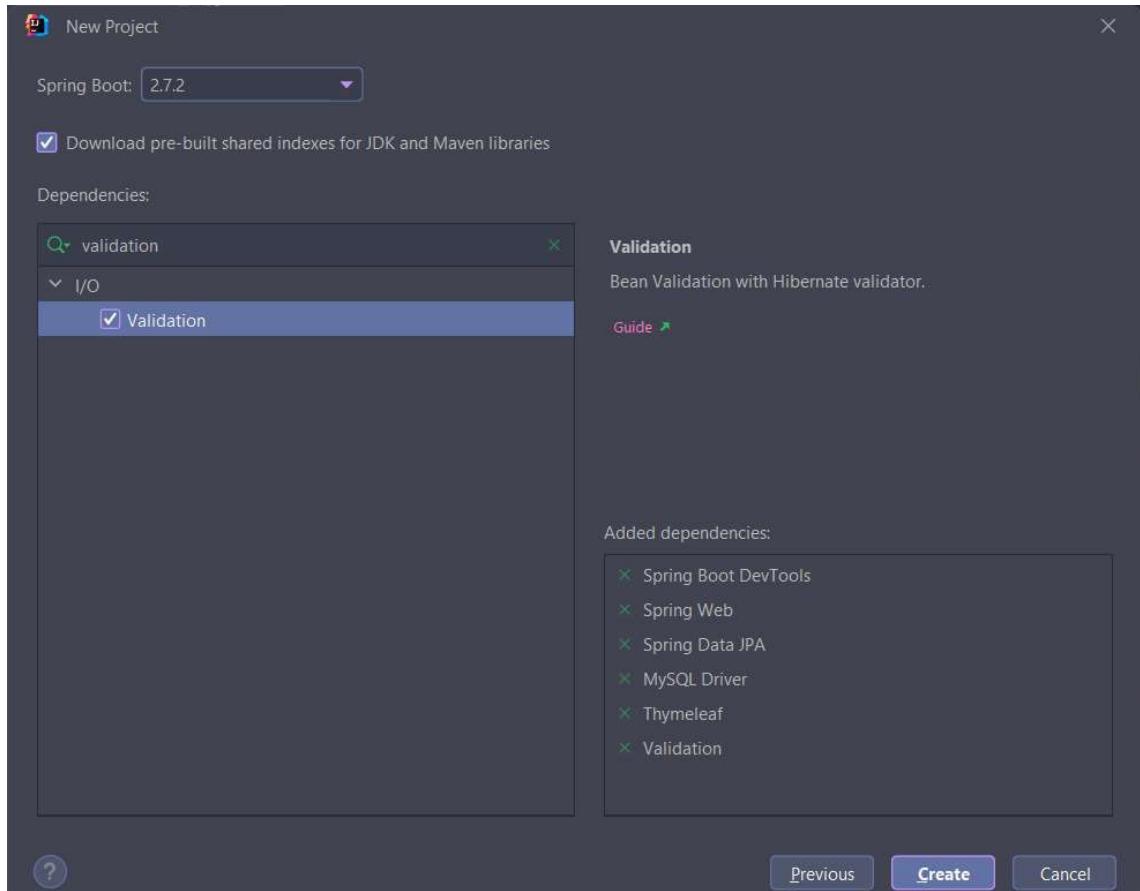


Figura 4 - Janela [Dependências]

Depois de criado o projeto, adicone as demais dependências que serão necessárias ao seu projeto. A listagem abaixo apresenta como deve ficar o seu arquivo pom.xml. Procure adicionar as versões que foram indicadas para não ter problemas de compatibilidade com este tutorial.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.2</version>
    <relativePath/> <!-- lookup parent from repository --&gt;
  &lt;/parent&gt;
  &lt;groupId&gt;com.example&lt;/groupId&gt;
  &lt;artifactId&gt;springProductList&lt;/artifactId&gt;
  &lt;version&gt;0.0.1-SNAPSHOT&lt;/version&gt;</pre>
```

```
<name>springProductList</name>
<description>springProductList</description>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</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-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!--Dependências adicionadas -->
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>bootstrap</artifactId>
        <version>5.2.0</version>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bowergithub.iconic</groupId>
        <artifactId>open-iconic</artifactId>
        <version>1.1.1</version>
    </dependency>
```

```
</dependency>
</dependencies>

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

Como você pode ver, com o Spring Boot temos que especificar apenas algumas dependências:

- Spring Boot Starter Web,
- Spring Boot Data JPA,
- Spring Boot Thymeleaf e
- MySQL JDBC driver.

Manualmente foi adicionada a dependência do Bootstrap e o webjars-locator-core, além da biblioteca de ícones open-iconic.

Nota 1

A biblioteca Bootstrap evita o uso de CDN para a distribuição do arquivo css. A biblioteca open-ionic possui uma coleção de ícones projetados para uso em aplicativos web, iOS, Android e desktop, apresenta suporte para SVG e é completamente aberto (opensource), licença MIT.

Nota 2

Ao usar o Spring Boot, ele detectará automaticamente a biblioteca webjars-locator-core no caminho de classe e a usará para resolver automaticamente a localização para você. Para habilitar esse recurso, você precisará adicionar a biblioteca webjars-locator-core como uma dependência do seu aplicativo no arquivo pom.xml.

Use o Spring Boot DevTools para reinicialização automática para que você não precise reiniciar manualmente o aplicativo durante o desenvolvimento.

No pacote `com.example.springproductlist` observe a classe **SpringProductListApplication**, ela deve conter o método main.

```
package net.maromo;

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

@SpringBootApplication
public class SpringProductListApplication {

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

Neste local crie o pacote Java principal **product**. Conforme Figura 5 - Pacote [product].

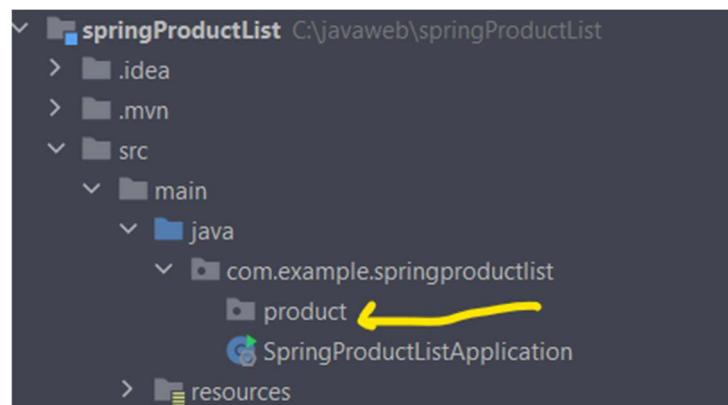


Figura 5 - Pacote [product]

3.1 Configure o Arquivo de Propriedades

Configure ou crie o arquivo `application.properties` no diretório `src/main/resources`. Veja a Figura 6 - Arquivo de Configuração [`application.properties`].

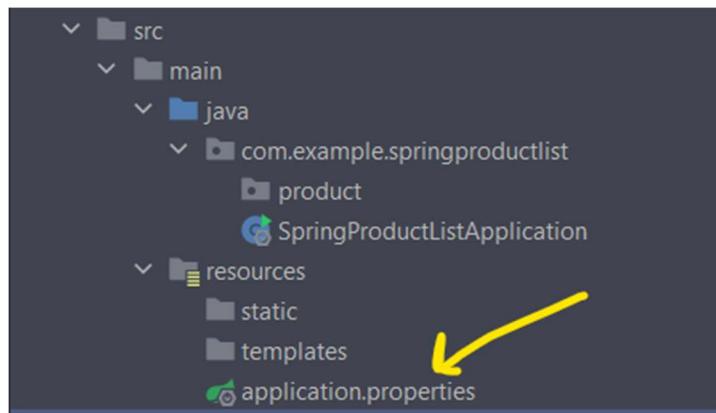


Figura 6 - Arquivo de Configuração [application.properties]

Conteúdo do arquivo:

```
#DATASOURCE
spring.datasource.url =
jdbc:mysql://localhost:3306/sales?createDatabaseIfNotExist=true
spring.datasource.username = root
spring.datasource.password = password

#JPA
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.hibernate.use-new-id-generator-mappings = false

#THYMELEAF
spring.thymeleaf.cache=false
```

Em #DATASOURCE a primeira linha diz ao spring para criar o esquema se ele não existir. Em seguida, nas próximas duas linhas, especificamos as propriedades de conexão do banco de dados (**altere os valores de acordo com suas configurações**).

Em #JPA informamos que o hibernate atualizará as tabelas existentes (criando, se não existirem).

Na linha `spring.jpa.show-sql` marcamos a propriedade como verdadeira, para que os comando das queries sejam apresentadas no console quando executamos a aplicação.

Em #THYMELEAF a propriedade `spring.thymeleaf.cache` definida como false, você desabilita o cache do Thymeleaf, assim os modelos serão recarregados automaticamente quando precisarem ser analisados, isso

tem a ver com a troca rápida dos modelos do lado do servidor (aumenta velocidade da resposta).

4 Definindo o Modelo

Definindo a classe **Modelo**, no pacote `product`, criado anteriormente, crie uma classe Java chamada **Product**. Veja a Figura 7 - Classe Modelo [Product].

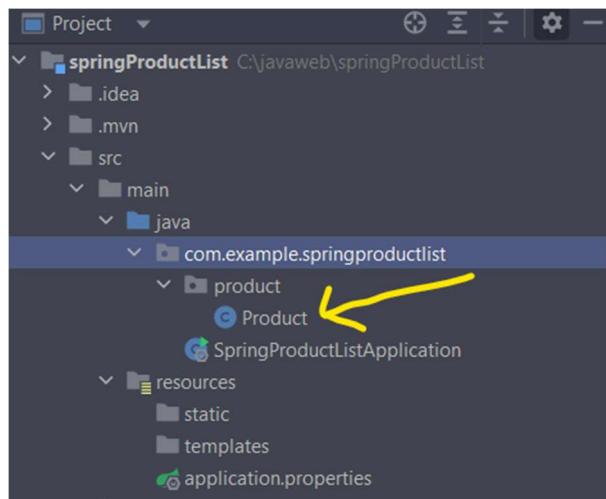


Figura 7 - Classe Modelo [Product]

Esta classe de modelo de domínio da aplicação é a nossa Entidade, usamos para mapear com a tabela de produtos (product) no banco de dados (sales) da seguinte forma:

```
package com.example.springproductlist.product;

import org.springframework.format.annotation.NumberFormat;
import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

@Entity
public class Product {
    private Long id;
    @NotBlank(message = "Product name is required.")
    @Size(max = 60, message = "the name must contain a maximum of 60 characters.")
    @Column(name = "name", nullable = false, length = 60)
    private String name;
    @NotBlank(message = "Brand is required.")
    @Size(min = 2, message = "the name must contain a minimum of 2 characters.")
    @Column(name = "brand", nullable = false, length = 45)
    private String brand;
    @NotBlank(message = "Made in is required.")
    @Size(min = 2, message = "the name must contain a minimum of 2 characters.")
    @Column(name = "madein", nullable = false, length = 45)
    private String madein;
    @NumberFormat(style = NumberFormat.Style.CURRENCY, pattern = "#,##0.00")
    @Column(name="price", nullable = false, columnDefinition = "DECIMAL(7,2) DEFAULT 0.00")
    private float price;
}
```

```

protected Product() {
}

protected Product(Long id, String name, String brand, String madein, float price) {
    super();
    this.id = id;
    this.name = name;
    this.brand = brand;
    this.madein = madein;
    this.price = price;
}

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public String getMadein() {
    return madein;
}

public void setMadein(String madein) {
    this.madein = madein;
}

public float getPrice() {
    return price;
}

public void setPrice(float price) {
    this.price = price;
}
}

```

Esta é uma classe de entidade JPA simples com o nome da classe e os nomes dos campos são idênticos aos nomes das colunas do produto da

tabela no banco de dados, para minimizar as anotações usadas. As anotações de validação são de responsabilidade do **Bean Validation**.

4.1 Validação back-end

A validação back-end é a validação de dados no lado servidor da aplicação. Esse é considerado o meio mais seguro de validação, já que, diferente de validação front-end, baseada em JavaScript que pode ser burlada.

No Java existe uma especificação para validação back-end a qual tem o nome de Bean Validation. Esse tipo de validação é baseado em anotação incluídas nas classes de entidades. Os atributos dessas classes são marcados com anotações do tipo `@NotNull`, `@NotBlank`, `@Size`, entre outras. Para que Bean Validation trabalhe, é necessário incluir uma biblioteca que represente a implementação dessa especificação, como exemplo, temos a dependencia `IO.Validation`. Dependência **já inserida** no nosso arquivo pom.xml. Abaixo:

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

Quando usamos o Spring MVC a validação back-end pode ser facilmente integrada ao Bean Validation como vimos na classe de domínio, a entidade **Product**. Exemplo de recorte, veja a Figura 8 - Recorte Anotações [Bean Validation]



```
8  @Entity
9  public class Product {
10     private Long id;
11     @NotBlank(message = "Product name is required.")
12     @Size(max = 60, message = "the name must contain a maximum of 60 characters.")
13     @Column(name = "name", nullable = false, length = 60)
14     private String name;
```

Figura 8 - Recorte Anotações [Bean Validation]

Na linha onde 11 a anotação @NotBlank indica que o nome do produto é obrigatório, o tamanho em caracteres é de no máximo 60, o campo tem o nome “name” não pode ser não preenchido (na tabela) e o tamanho definido para o campo no banco também é de 60 caracteres. Os demais campos possuem validações parecidas. Observe todas.

5 Interface Repository

Crie uma interface java **Repository** com o nome **ProductRepository**, também no pacote product. Veja a Figura 9 - Interface [ProductRepository].

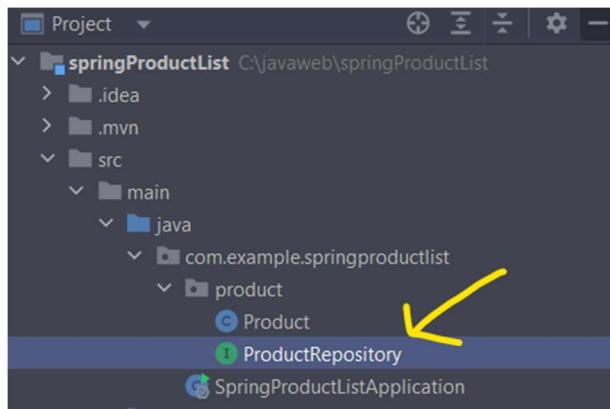


Figura 9 - Interface [ProductRepository]

Deixe o Código como mostra a próxima listagem:

```
package com.example.springproductlist.product;

import org.springframework.data.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, Long> {
}
```

Como você pode ver, essa interface estende a interface `CrudRepository` do Spring Data JPA. `CrudRepository` define métodos CRUD padrão, além de operações específicas de JPA. Não precisamos escrever código de implementação porque o Spring Data JPA gerará o código necessário em tempo de execução, na forma de instâncias de proxy.

Portanto, o objetivo de escrever a interface do repositório é informar ao Spring Data JPA sobre o tipo de domínio (Product) e o tipo de ID (Long) com os quais vai trabalhar. A Figura 10 - Interface implementada [`CrudRepository`] apresenta os métodos CRUD padrão da Interface `CrudRepository`.

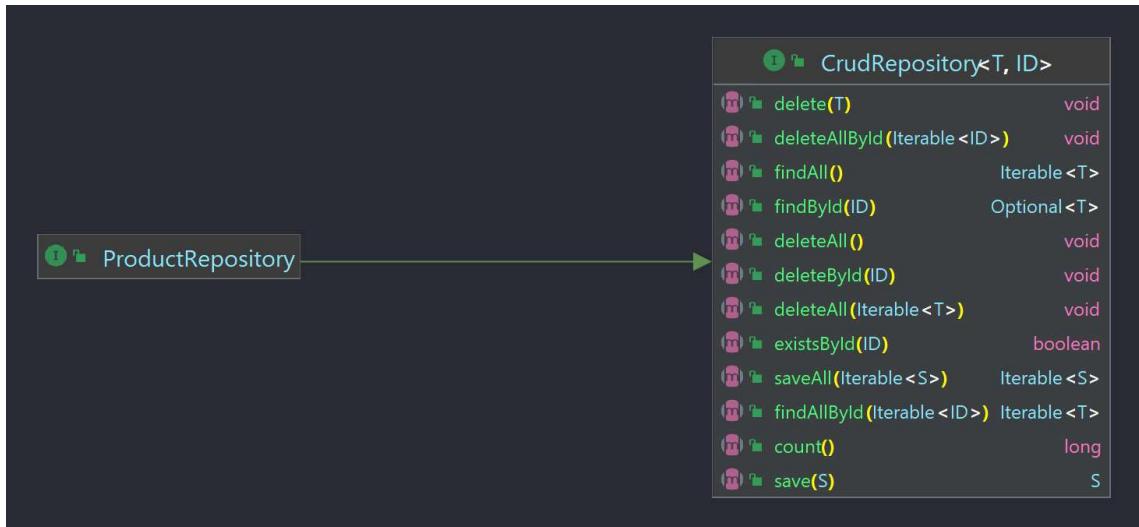


Figura 10 - Interface implementada [CrudRepository]

6 Camada Service

Definindo a Classe Service

Crie uma classe Java chamada **ProductService** no pacote product.

Veja a Figura 11 - Camada Service [ProductService].

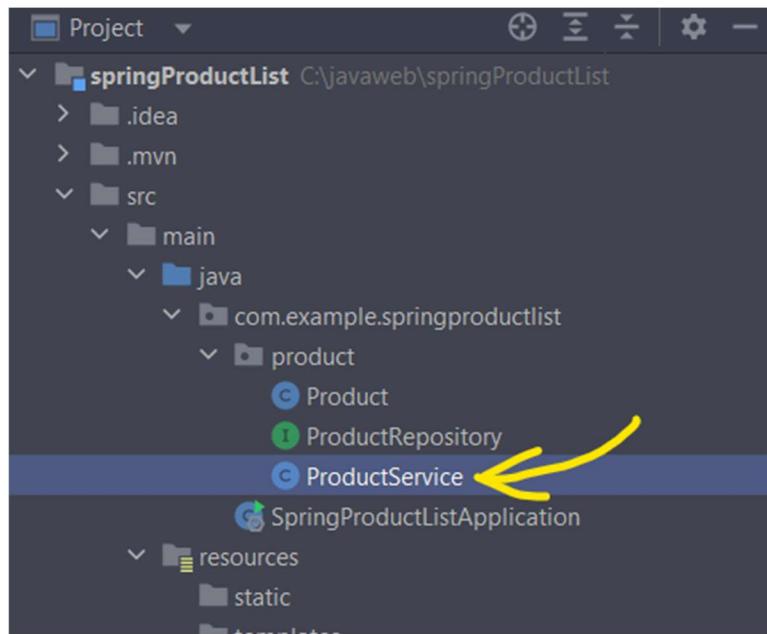


Figura 11 - Camada Service [ProductService]

Em seguida precisamos codificar a classe **ProductService** na camada de serviço/negócio com o seguinte código:

```
package com.example.springproductlist.product;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
@Transactional
public class ProductService {

    private final ProductRepository repo;

    public ProductService(ProductRepository repo) {
        this.repo = repo;
    }

    public List<Product> listAll() {
        return (List<Product>) repo.findAll();
    }

    public void save(Product product) {
```

```
    repo.save(product);
}

public Product get(long id) {
    return repo.findById(id).get();
}

public void delete(long id) {
    repo.deleteById(id);
}
}
```

A anotação `@Service` indica que a classe faz anotações na camada de serviço, já a anotação `@Transactional` trabalha dentro do escopo de uma transação no banco de dados, a transação do banco de dados ocorre dentro do PersistenceContext, que por sua vez, está dentro do EntityManager que é implementado usando Hibernate Session.

Nesta classe a injeção de dependência será realizada normalmente. Mas esse processo só vai ocorrer com sucesso **se existir um único método construtor na classe**. Assim, você não precisa usar nas novas versões do Spring (4.3 para frente) a anotação `@Autowired`, deixando seu código mais limpo.

Em tempo de execução, o Spring Data JPA gerará uma instância de proxy de **ProductRepository** e a injetará na instância da classe **ProductService**.

Você pode ver que essa classe de serviço é redundante, pois delega todas as chamadas para **ProductRepository**. Na verdade, a lógica de negócios seria mais complexa ao longo do tempo, por exemplo, chamando duas ou mais instâncias de repositório. Então, criamos essa classe para fins de extensibilidade no futuro. Nossa Diagrama de Classes no momento, veja a Figura 12 - Diagrama de Classes [Pacote Product]

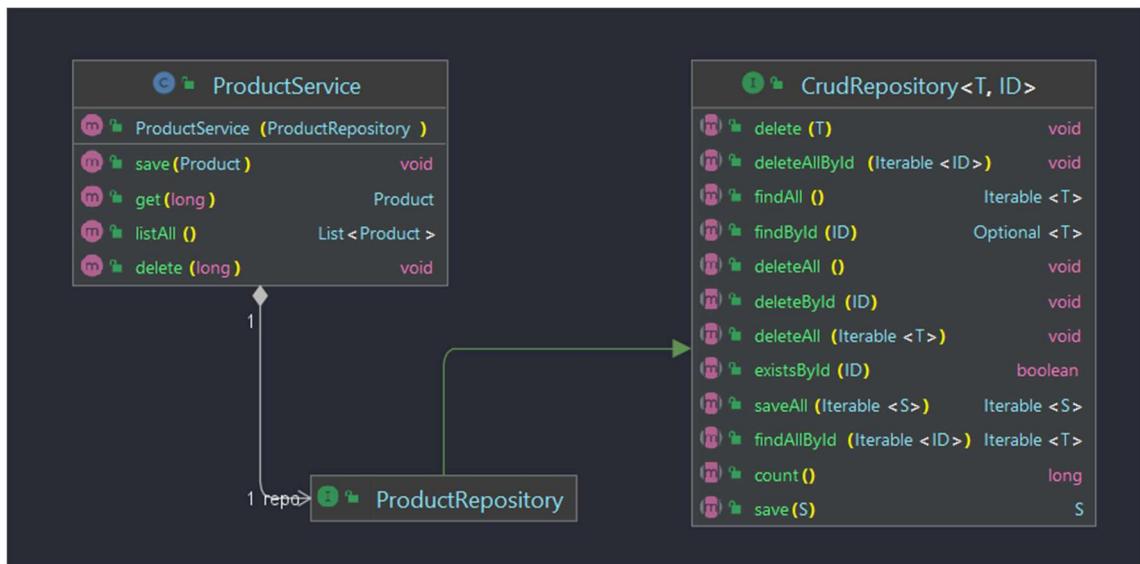


Figura 12 - Diagrama de Classes [Pacote Product]

7 Controller

Podemos anotar um **Controller** (uma classe que contém métodos para estrutura Spring MVC) com a anotação `@Controller`. Isso é uma especialização da classe `@Component`, que nos permite detectar automaticamente as classes de implementação através da verificação do caminho de classe.

Crie no pacote product a classe **AppController** que atuará como um controlador Spring MVC para lidar com solicitações dos clientes.

```
package com.example.springproductlist.product;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.validation.Valid;
import java.util.List;

@Controller
public class AppController {

    private final ProductService service;

    public AppController(ProductService service) {
        this.service = service;
    }

    @RequestMapping("/")
    public String viewHomePage(Model model) {
        List<Product> listProducts = service.listAll();
        model.addAttribute("listProducts", listProducts);
        return "index";
    }

    @RequestMapping("/new")
    public String showNewProductPage(Model model) {
        Product product = new Product();
        model.addAttribute("product", product);

        return "new_product";
    }
}
```

```

@RequestMapping(value = "/save", method = RequestMethod.POST)
public String saveProduct(@Valid @ModelAttribute("product") Product product,
                           BindingResult result, RedirectAttributes attr) {

    if (result.hasErrors()) {
        if(product.getId()==null){
            return "new_product";
        }
        return "edit_product";
    }
    service.save(product);
    attr.addFlashAttribute("success", "Record saved successfully");

    return "redirect:/";
}

@RequestMapping("/edit/{id}")
public ModelAndView showEditProductPage(@PathVariable(name = "id") int id) {
    ModelAndView mav = new ModelAndView("edit_product");
    Product product = service.get(id);
    mav.addObject("product", product);
    return mav;
}

@RequestMapping("/delete/{id}")
public String deleteProduct(@PathVariable(name = "id") int id) {
    service.delete(id);
    return "redirect:/";
}

```

Como você pode ver, preparamos para injeção de uma instância da classe `ProductService` neste controlador (utilizando o método construtor) – o Spring criará uma injeção de dependência automaticamente em tempo de execução. Foram descrito o código para os métodos do manipulador ao implementar cada operação CRUD.

7.1 Descrição dos métodos da classe AppController

A página inicial do site exibe uma lista de todos os produtos, para isso foi adicionado o seguinte método de manipulador na classe do controlador Spring MVC.

```

@RequestMapping("/")
public String viewHomePage(Model model) {
    List<Product> listProducts = service.listAll();
    model.addAttribute("listProducts", listProducts);
    return "index";
}

```

A URL new que será adicionada no html quando o usuário clicar sobre o link **New Product** é tratada pelo seguinte método na classe AppController:

```
@RequestMapping("/new")
public String showNewProductPage(Model model) {
    Product product = new Product();
    model.addAttribute("product", product);

    return "new_product";
}
```

Precisamos codificar outro método de manipulador para salvar as informações do produto no banco de dados, que será solicitado pelo usuário ao clicar no botão **Save** do seu formulário web.

```
@RequestMapping(value = "/save", method = RequestMethod.POST)
public String saveProduct(@Valid @ModelAttribute("product") Product product,
                           BindingResult result, RedirectAttributes attr) {

    if (result.hasErrors()) {
        if(product.getId()==null){
            return "new_product";
        }
        return "edit_product";
    }
    service.save(product);
    attr.addFlashAttribute("success", "Record saved successfully");

    return "redirect:/";
}
```

Depois que o produto é inserido no banco de dados, ele redireciona para a página inicial para atualizar a lista de produtos.

O recurso de edição do produto é referenciado na página inicial, que terá um hiperlink que permite aos usuários editar um produto.

Codifique o método do manipulador na classe do controlador da seguinte maneira:

```
@RequestMapping("/edit/{id}")
public ModelAndView showEditProductPage(@PathVariable(name = "id") int id) {
    ModelAndView mav = new ModelAndView("edit_product");
    Product product = service.get(id);
    mav.addObject("product", product);
```

```
    return mav;
}
```

O recurso da exclusão de produto será por meio de um hiperlink para excluir um produto na página inicial.

```
<a th:href="@{'/delete/' + ${product.id}}>Delete</a>
```

Para isso, codifique o método manipulador na classe do controlador da seguinte maneira:

```
@RequestMapping("/delete/{id}")
public String deleteProduct(@PathVariable(name = "id") int id) {
    service.delete(id);
    return "redirect:/";
}
```

Quando o usuário clicar no hiperlink excluir, as informações do produto correspondente, são removidas do banco de dados e a página inicial é atualizada.

8 Spring Boot Application Class

A classe Spring Boot Spring Application é usada para inicializar um aplicativo Spring a partir de um método principal Java (main). Esta classe cria automaticamente o ApplicationContext a partir do classpath, verifica as classes de configuração e inicia o aplicativo. Você deve verificar se o método main() na classe SpringProductListApplication está preparado para inicializar nosso aplicativo Spring Boot. (como mostrado na sequência):

```
package com.example.springproductlist;

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

@SpringBootApplication
public class SpringProductListApplication {

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

Aqui, a anotação `@SpringBootApplication` faz todas as “coisas mágicas”, como criar a instância do servidor web e o servlet despacher do Spring MVC.

9 Usando o Thymeleaf

O Thymeleaf é uma template engine para projetos Java que facilita a criação de páginas HTML. Para que possamos enviar dados de uma aplicação Spring Boot para o Thymeleaf, antes, é necessário criarmos um controller que será executado quando uma determinada rota for chamada pelo navegador, dessa maneira esse controller será responsável por enviar esses dados e definir qual o template do Thymeleaf deve ser exibido para o usuário. Ou seja, o controller tem por principal objetivo, direcionar o fluxo da aplicação mapeando e direcionando as ações recebidas (request) pela camada da apresentação para os respectivos serviços da aplicação.

Neste exemplo usamos Thymeleaf em vez de JSP, para isso no diretório de **templates** em `src/main/resources` (usado para armazenar arquivos de templates HTML) - Figura 13 - templates - crie um arquivo chamado **alert.html** e outro chamado **index.html**. Veja a Figura 14 - Arquivos alert.html e index.html.

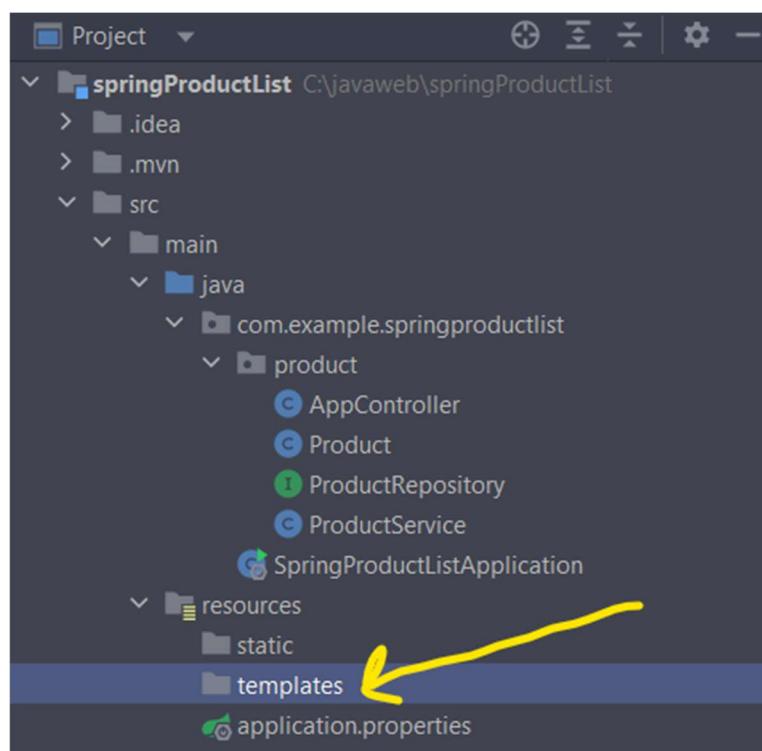


Figura 13 - templates

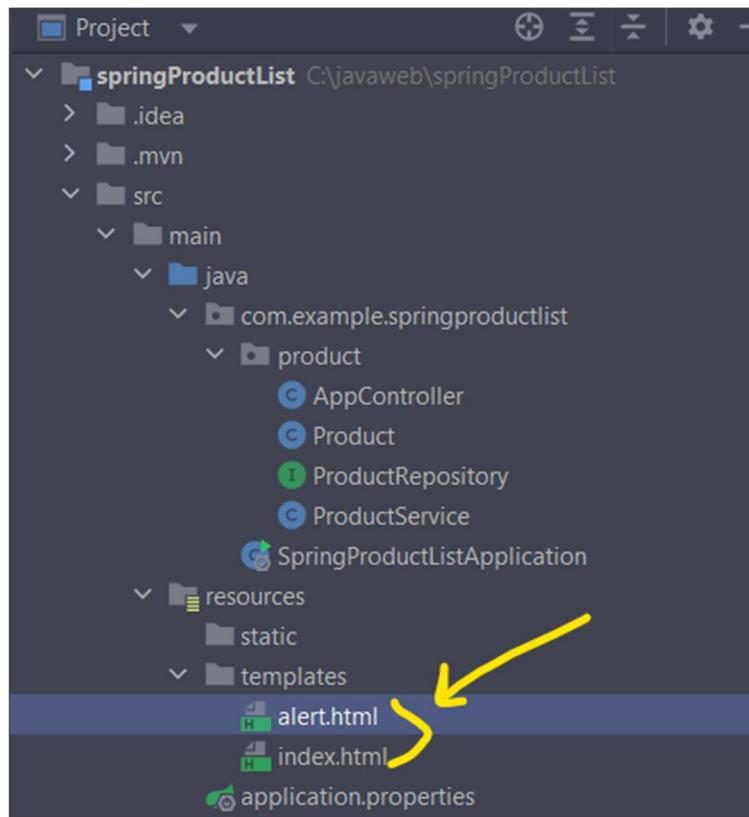


Figura 14 - Arquivos alert.html e index.html

O arquivo **alert.html** será um arquivo fragmento do Thymeleaf. Os fragments são uma forma de conseguirmos isolar um trecho de código dos nossos templates em um arquivo separado e então inclui-los em outros templates de acordo com a nossa necessidade. O código abaixo servirá para emitir alertas em mais de uma janela de código.

9.1 Arquivo alert.html

Código do arquivo alert.html:

```
<div th:if="${success} != null">
    <div class="alert alert-success alert-dismissible fade show" role="alert">
        <i class="oi oi-check"></i>
        <span>
            <strong th:text="${success}"></strong>
        </span>
        <button type="button" class="btn-close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
    </div>
</div>

<div th:if="${fail} != null">
```

```

<div class="alert alert-danger alert-dismissible fade show" role="alert">
    <i class="oi oi-check"></i>
    <span>
        <strong th:text="${fail}"></strong>
    </span>
    <button type="button" class="btn-close" data-dismiss="alert" aria-label="Close">
        <span aria-hidden="true">&times;</span>
    </button>
</div>
</div>

```

Observe a propriedade especial do Thymeleaf chamada th:text e nela como valor da propriedade colocamos o nome do atributo que queremos exibir entre \${}, no nosso caso temos dois, o de sucesso \${sucess} e o de falha \${fail}.

9.2 Arquivo index.html

Código do arquivo index.html:

```

<!DOCTYPE html>
<html lang="pt-br" xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8"/>
    <title>Product Manager</title>
    <link rel="icon" href="/image/favicon.png"/>
    <link th:rel="stylesheet" th:href="@{/webjars/bootstrap/css/bootstrap.min.css} "/>
    <link th:rel="stylesheet" th:href="@{/webjars/open-iconic/font/css/open-iconic-bootstrap.min.css} "/>
</head>
<body>
<div class="container">
    <div class="container-fluid py-5">
        <h1 class="display-5 fw-bold">Product List</h1>
        <p class="col-md-8 fs-4">
            List of products available today for sale
        </p>
        <a class="btn btn-primary btn-lg" href="/new">Create New Product</a>
    </div>

    <div class="container-fluid">
        <div th:replace="alert"></div>
        <table class="table table-bordered">
            <thead>
                <tr>
                    <th>Product ID</th>
                    <th>Name</th>
                    <th>Brand</th>
                    <th>Made In</th>
                    <th>Price</th>
                    <th>Actions</th>
                </tr>
            <tbody>

```

```

</thead>
<tbody>

<tr th:each="product : ${listProducts}">
    <td th:text="${product.id}">Product ID</td>
    <td th:text="${product.name}">Name</td>
    <td th:text="${product.brand}">Brand</td>
    <td th:text="${product.madein}">Made in</td>
    <td class="text-end" th:text="${{product.price}}">Price</td>
    <td>
        <a class="btn btn-primary oi oi-brush"
            title="Edit Record!"
            th:href="@{'/edit/' + ${product.id}}" role="button">
        </a>
        <a class="btn btn-danger oi oi-circle-x"
            title="Delete Record?"
            th:href="@{'/delete/' + ${product.id}}">
        </a>
    </td>
</tr>
</tbody>
</table>
</div>
</div>
</body>
</html>

```

Agora podemos executar a classe **SpringProductListApplication** para testar nosso aplicativo Web Spring Boot.

Abra seu navegador e digite a URL `http://localhost:8080` para ver a página inicial do site:

Product ID	Name	Brand	Made In	Price	Actions

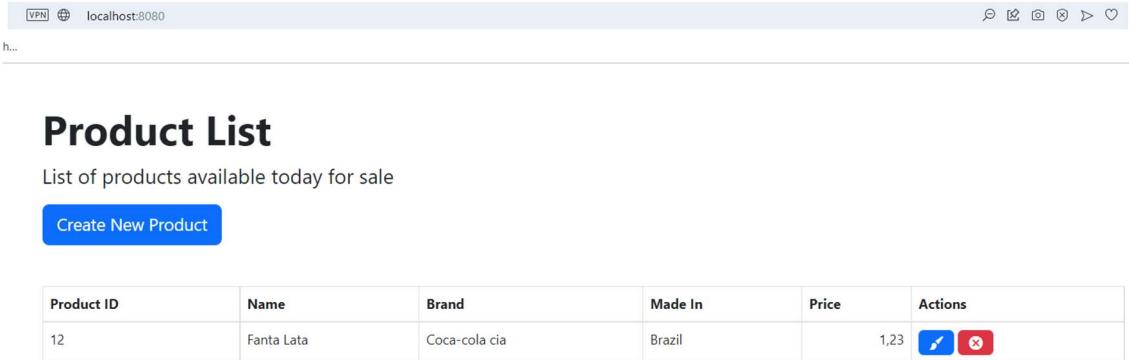
Figura 15 - Resultado da execução do index.html

Veja, a lista de produtos é exibida. Supondo que você tenha inserida algumas linhas na tabela de produtos antes. No nosso caso, está vazia. Use o Workbench insira um novo registro e atualize em seguida o navegador.

Script para inserção de um registro no Workbench:

```
insert into sales.product(name, brand, madein, price) values ('Fanta Lata', 'Coca-cola cia', 'Brazil', 1.23);
```

Atualizando o navegador:



The screenshot shows a web browser window with the URL 'localhost:8080'. The page title is 'Product List' and the subtitle is 'List of products available today for sale'. A blue button labeled 'Create New Product' is visible. Below it is a table with the following data:

Product ID	Name	Brand	Made In	Price	Actions
12	Fanta Lata	Coca-cola cia	Brazil	1,23	 

Figura 16 - Nova execução

Observe que no arquivo **index.html**, temos um hiperlink que permite ao usuário criar um produto:

```
<a href="/new">Create New Product</a>
```

Se clicarmos sobre o item, ainda não possuímos o destino do recurso. Para que isso ocorra é necessário que você crie dois arquivos, primeiro o fragmento **validacao.html** e o segundo é o arquivo do recurso de novo produto, o arquivo **new_product.html**. Crie ambos na pasta templates. Veja a Figura 17 - Novos arquivo na pasta template.

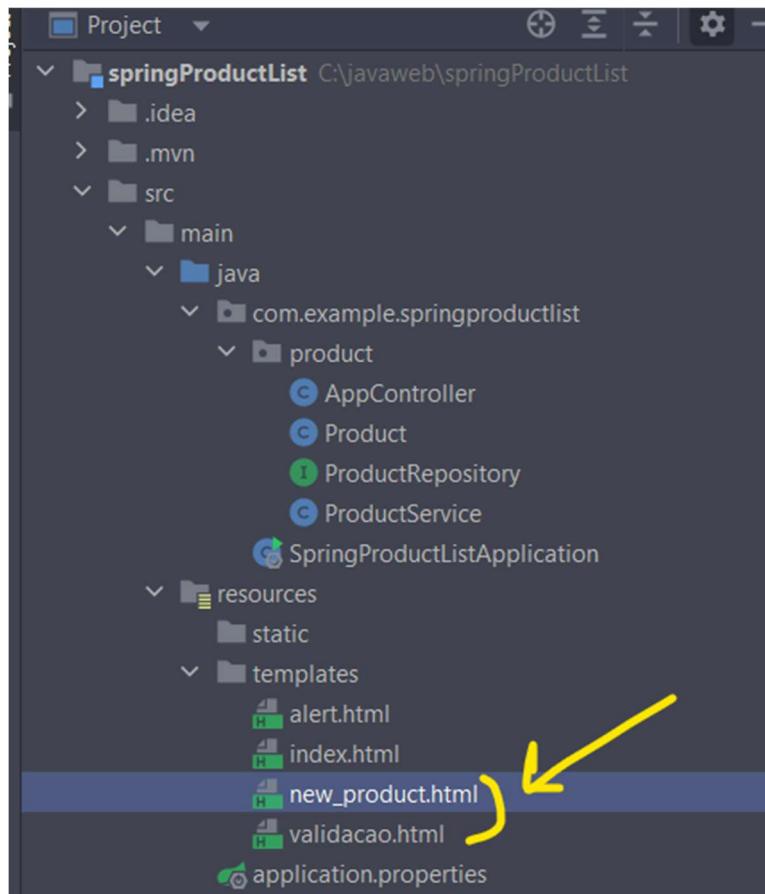


Figura 17 - Novos arquivo na pasta template

9.3 Arquivo visualização.html

Abaixo o código do arquivo **visualizacao.html**:

```
<html>
<head><meta charset="UTF-8"></head>
<body>
<div th:if="${#fields.hasAnyErrors()}" th:fragment="validacao">
    <div class="alert alert-danger alert-dismissible fade show" role="alert">
        <div th:each="error : ${#fields.detailedErrors()}">
            <i class="oi oi-warning"></i>
            <span th:text="${error.message}"></span>
        </div>
        <button type="button" class="btn-close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
    </div>
</div>
</body>
</html>
```

9.4 Arquivo new_product.html

Abaixo código do arquivo new_product.html:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8"/>
    <title>Create New Product</title>
    <link rel="icon" href="/image/favicon.png"/>
    <link th:rel="stylesheet" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}"/>
    <link th:rel="stylesheet" th:href="@{/webjars/open-iconic/font/css/open-iconic-bootstrap.min.css}"/>
</head>
<body>
<div class="container col-sm-6">
    <div class="container-fluid py-5">
        <h1 class="display-5 fw-bold">Create New Product</h1>
        <p class="fs-4">
            Data for the new product
        </p>
    </div>
    <form class="form-control-sm" action="#" th:action="@{/save}" th:object="${product}"
          method="post">

        <div th:replace="validacao :: validacao"></div>
        <div class="mb-3">
            <label class="form-label" for="name">Product Name:</label>
            <input class="form-control" type="text" id="name" th:field="*{name}"/>
        </div>
        <div class="mb-3">
            <label class="form-label" for="brand">Brand:</label>
            <input class="form-control" type="text" id="brand" th:field="*{brand}"/>
        </div>
        <div class="mb-3">
            <label class="form-label" for="made-in">Made in:</label>
            <input class="form-control" type="text" id="made-in" th:field="*{madein}"/>
        </div>
        <div class="mb-3">
            <label class="form-label" for="price">Price:</label>
            <input class="form-control" type="text" id="price" th:field="*{price}"/>
        </div>
        <button type="submit" class="btn btn-primary btn-lg">Save</button>
    </form>
</div>
</body>
</html>
```

Como você pode ver, aqui usamos a sintaxe ThymeLeaf para o formulário em vez de tags de formulário jsp (Java Server Page). A página Novo Produto tem esta aparência, faça uma inserção (Figura 18)

- Formulário Novo Produto) para ver o resultado (Figura 19 - Novo resultado):

localhost:8080/new

Product Name:
Coca-Cola lata

Brand:
Coca-Cola cia

Made in:
Brazil

Price:
1.99

Save

Figura 18 - Formulário Novo Produto

Resultado após salvo os dados:

Product List

List of products available today for sale

[Create New Product](#)

✓ Record saved successfully

Product ID	Name	Brand	Made In	Price	Actions
16	Coca-Cola lata	Coca-Cola cia	Brazil	1,99	
12	Fanta Lata	Coca-cola cia	Brazil	1,23	

Figura 19 - Novo resultado

Na execução, depois que o produto é inserido no banco de dados, ele redireciona para a página inicial para atualizar a lista de produtos.

Como escrevemos as validações no Bean usando nosso AppController (listagem a seguir), ao executarmos o recurso `saveProduct` para

salvamento dos dados ele verifica se houve um ou mais erros. Caso tenha ocorrido qualquer erro, ele redireciona para a página new_product novamente e nos campos de validação, ele exibe as mensagens correspondentes.

```
@RequestMapping(value = "/save", method = RequestMethod.POST)
public String saveProduct(@Valid @ModelAttribute("product") Product product,
                           BindingResult result, RedirectAttributes attr) {

    if (result.hasErrors()) {
        if(product.getId()==null){
            return "new_product";
        }
        return "edit_product";
    }
    service.save(product);
    attr.addFlashAttribute("success", "Record saved successfully");

    return "redirect:/";
}
```

Listagem: Recurso saveProduct

Na página inicial, você pode ver que há um hiperlink que permite aos usuários editar um produto:

```
<a th:href="@{'/edit/' + ${product.id}}>Edit</a>
```

Para que possa ser redirecionado para um formulário de edição de produto com os dados do registro a ser alterado, precisamos inicialmente criar o arquivo **edit_product** na pasta **template**, veja a Figura 20 - Arquivo edit_product:

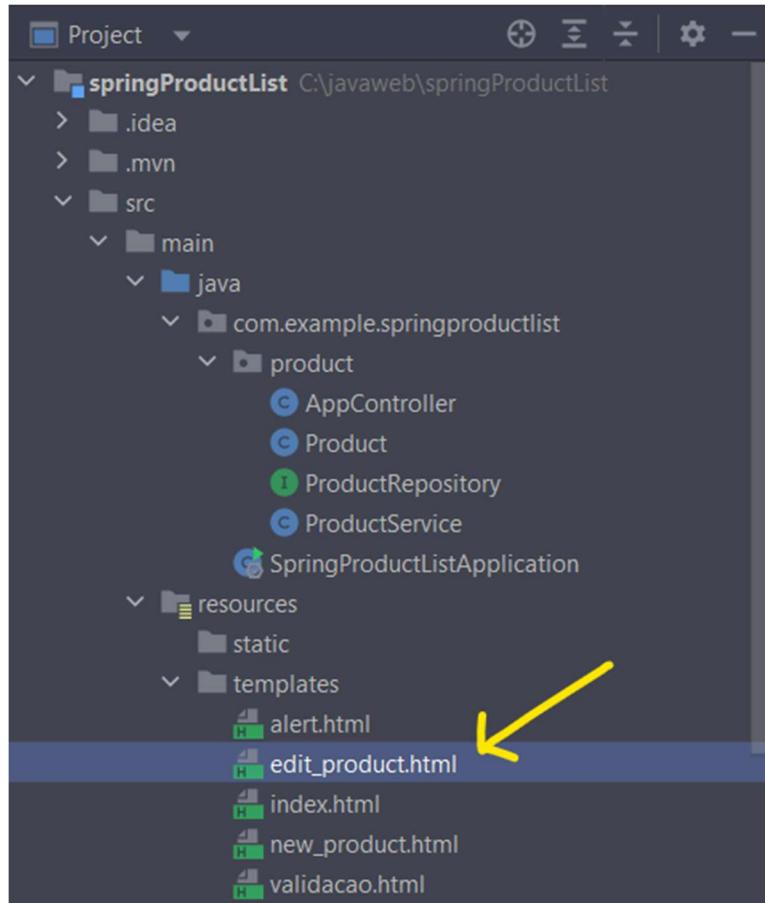


Figura 20 - Arquivo edit_product

E a página **edit_product.html** deve conter o seguinte código:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8"/>
    <title>Edit Product</title>
    <link rel="icon" href="/image/favicon.png"/>
    <link th:rel="stylesheet" th:href="@{/webjars/bootstrap/css/bootstrap.min.css} "/>
    <link th:rel="stylesheet" th:href="@{/webjars/open-iconic/font/css/open-iconic-bootstrap.min.css} "/>
</head>
<body>
<div class="container col-sm-6">
    <div class="container-fluid py-5">
        <h1 class="display-5 fw-bold">Update Product</h1>
        <p class="fs-4">
            New product data
        </p>
    </div>
    <form class="form-control-sm" action="#" th:action="@{/save}" th:object="${product}"
          method="post">
```

```

<div th:replace="validacao :: validacao"></div>
<div class="mb-3">
    <label class="form-label" for="product_id">Product Id:</label>
    <input class="form-control" type="text" readonly="readonly" id="product_id" th:field="*{id}" />
</div>
<div class="mb-3">
    <label class="form-label" for="name">Product Name:</label>
    <input class="form-control" type="text" id="name" th:field="*{name}" />
</div>
<div class="mb-3">
    <label class="form-label" for="brand">Brand:</label>
    <input class="form-control" type="text" id="brand" th:field="*{brand}" />
</div>
<div class="mb-3">
    <label class="form-label" for="made-in">Made in:</label>
    <input class="form-control" type="text" id="made-in" th:field="*{madein}" />
</div>
<div class="mb-3">
    <label class="form-label" for="price">Price:</label>
    <input class="form-control" type="text" id="price" th:field="*{price}" />
</div>
<button class="btn btn-primary btn-lg" type="submit">Save</button>
</form>
</div>
</body>
</html>

```

Ao executarmos a aplicação e selecionarmos um item para edição, o resultado será o redirecionamento para a página de edição, com os dados do registro a ser alterado. Veja a Figura 21 - Editando um Produto [Recurso] e a Figura 22 - Formulário de edição de produto.

Apresentando o formulário de edição:

Update Product

New product data

Product Id:

Product Name:

Brand:

Made in:

Price:

Save

Figura 22 - Formulário de edição de produto

Para excluir um produto, executamos o recurso **Excluir Produto**. Você pode ver o hiperlink para excluir um produto na página inicial:

```
1<a th:href="/@{'/delete/' + ${product.id}}">Delete</a>
```

Resultado da execução da exclusão do produto anteriormente editado.

Product List

List of products available today for sale

Create New Product

Product ID	Name	Brand	Made In	Price	Actions
16	Coca-Cola lata	Coca-Cola cia	Brazil	1,99	 

Figura 23 - Após excluído o registro

10 Hospedando a aplicação no Heroku

O Heroku trata-se de uma PaaS (Plataforma como um Serviço) que permite hospedagem, configuração, testagem e publicação de projetos virtuais na nuvem. Vamos utilizar para hospedar nosso exemplo.

Para isso inicialmente você precisa criar uma conta no Heroku. Você precisa de um cartão de crédito para verificação (usaremos serviços gratuitos), o cartão serve apenas como verificação da existência real do usuário da plataforma.



10.1 Criando conta no Heroku

Para criar uma conta free, acesse o link:

<<<https://signup.heroku.com/login>>>.

Preencha o formulário da Figura 24 - Formulário Heroku [Nova Conta]
- com os seus dados:

The screenshot shows the Heroku sign-up process. On the left, there are three sections: 'Free account' (with a brief description), 'Your app platform' (with a brief description), and 'Deploy now' (with a brief description). On the right, the sign-up form is displayed with the following fields:

- First name *: Marcos
- Last name *: Moraes
- Email address *: prof.maromo@gmail.com
- Company name: maromo
- Role *: Student
- Country/Region *: Brazil
- Primary development language *: Java

At the bottom right is a reCAPTCHA checkbox labeled "I'm not a robot".

Figura 24 - Formulário Heroku [Nova Conta]

Não deixe de selecionar Java como a ferramenta primária de desenvolvimento.

Cheque o email informado. Nele você deve receber um link de ativação da conta.

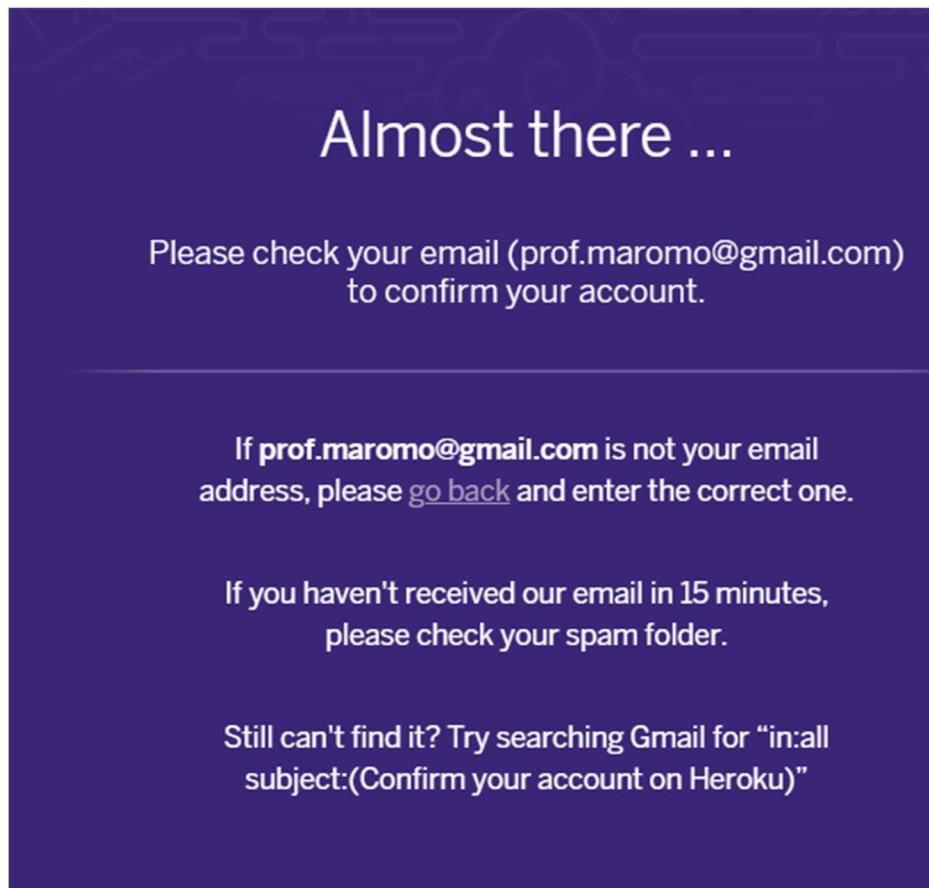


Figura 25 - Ativação da conta

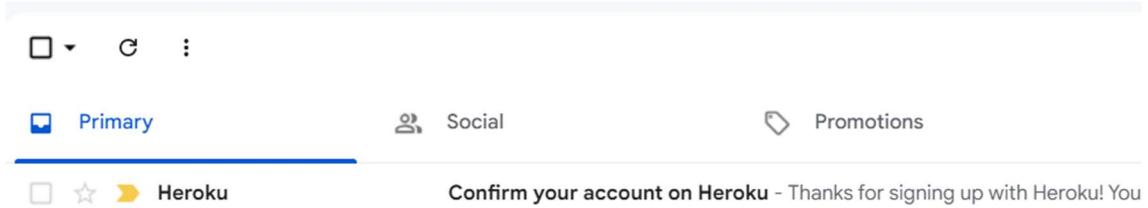


Figura 26 - Email recebido

Ao confirmar, altere as suas credenciais de acesso, alterando a senha.

Veja

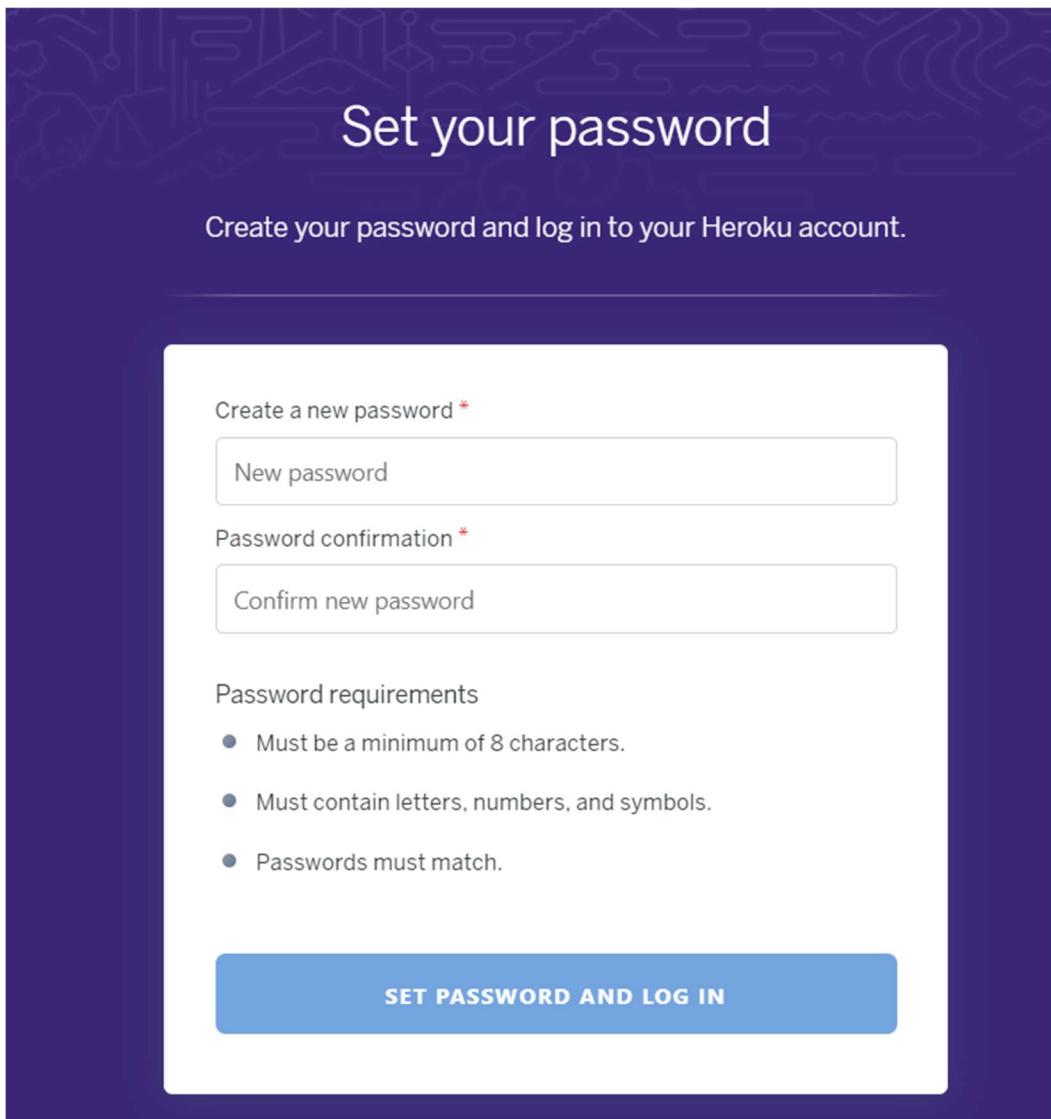


Figura 27 - Definindo a senha

Após isso você é redirecionado a página de saudação. Nela clique sobre o botão para proceder a finalização. Veja Figura 28 - Página de Saudação.

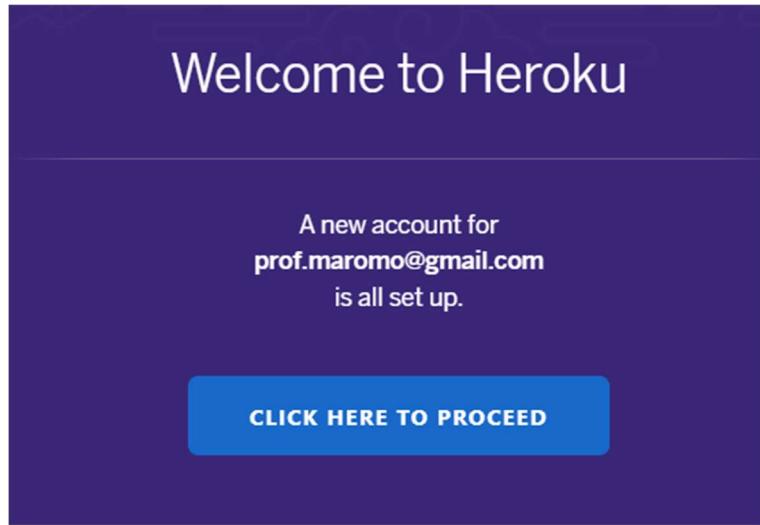


Figura 28 - Página de Saudação

Finalmente clique no botão Accept. Veja Figura 29 - Confirmação de dados e aceite.

Add-ons, Buildpacks or Buttons that are not Heroku Services ("Heroku Elements") are governed by the [Heroku Elements Terms of Use \(Default\)](#), unless the Heroku Elements Marketplace provider has furnished to Heroku a separate terms of use, in which case such provider's terms of use govern the use or purchase of the applicable Heroku Elements. [Additional Terms](#) apply to credit card customers of the Heroku Services.

Heroku Marketplace Providers

If you are or become a Heroku Elements Marketplace Provider, you further agree that your participation in the Heroku Elements Marketplace is governed by the [Salesforce License and Distribution Agreement for the Heroku Elements Marketplace](#).

Italian Customers

Are you domiciled in Italy?

No.

Figura 29 - Confirmação de dados e aceite.

10.2 Instalar o Heroku CLI

Na página principal da Heroku, ao logar depois de confirmada as suas credenciais, você deve selecionar a opção Java, veja Figura 30 - Heroku [Novo App Java].

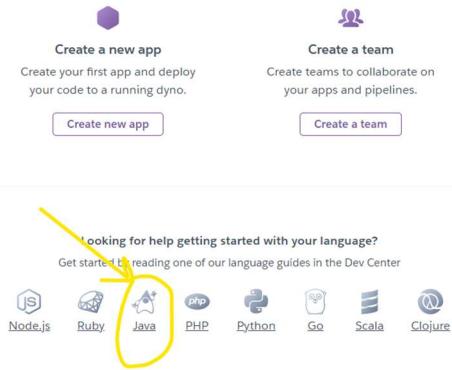


Figura 30 - Heroku [Novo App Java]

Ao surgir a tela de Introdução ao Heroku com Java, você deve clicar sobre o botão [I'm ready to start].

Getting Started on Heroku with Java

English — 日本語に切り替え

The screenshot shows the "Introduction" section of the Java tutorial. On the left is a sidebar with links: Set Up, Prepare the App, Declare App Dependencies, Deploy the App, Scale the App, View Logs, Provision Add-ons, Use a Database, and Prepare the Local Environment. The "Introduction" section contains text about deploying a Java app and a list of prerequisites. At the bottom right is a purple button labeled "I'm ready to start". A yellow arrow points from the top of the image towards this button.

Figura 31 - Botão [I'm ready to start]

Ao ser direcionado à pagina de setup, você receberá a informação de que necessita do Git (ferramenta de versionamento) instalada. Caso você não possua, é necessário baixar a ferramenta antes de continuar o processo. O link para efetuar o download da ferramenta é <https://git-scm.com/download/win>.

Com o git instalado, agora podemos instalar o Heroku CLI. Você usa a CLI para gerenciar e dimensionar seus aplicativos, provisionar complementos, visualizar seus logs de aplicativos recentes e executar seu aplicativo localmente.

Selecione a opção correta para o seu sistema operacional, clicando sobre um dos botões indicados. Veja ..

Set Up

-  The Heroku CLI requires [Git](#), the popular version control system. If you don't already have Git installed, complete the following before proceeding:
- [Git installation](#)
 - [First-time Git setup](#)

In this step, you install the Heroku Command Line Interface (CLI). You use the CLI to manage and scale your applications, provision add-ons, view your recent application logs, and run your application locally.

Download and run the installer for your platform:

 **macOS**

[Download the installer](#)

Also available via Homebrew:

```
$ brew install heroku/brew/heroku
```

 **Windows**

Download the appropriate installer for your Windows installation:

[64-bit installer](#)

[32-bit installer](#)

Figura 32 - Informações sobre o Set Up necessário para o Heroku

Após baixado, faça a instalação padrão, ou seja, avançando pelo assistente até a conclusão da operação de instalação.

10.3 Testando a instalação do Heroku CLI

Caso esteja no Windows, para testar se a instalação ocorreu com sucesso, você deve acessar o prompt de comando e digitar o seguinte comando:



Como resultado da execução do comando você verá a versão instalado do Heroku CLI.

Ainda logado, volte a página web da Heroku e acesse o item Dashboard. Veja a Figura 32 - Informações sobre o Set Up necessário para o Heroku.

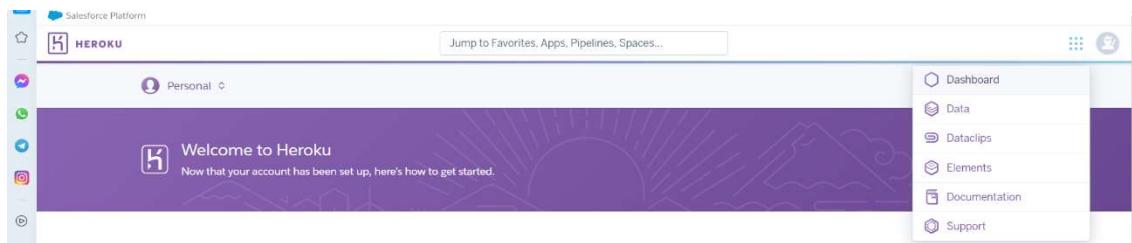


Figura - Dashboard [Link]

Agora clique sobre o botão **[Create new app]**.

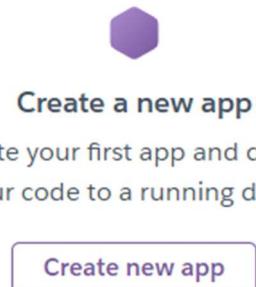


Figura 34 - Create new app

Na próxima janela, você deve digitar um nome para o aplicativo e na sequência clicar sobre o botão [Create app]. Veja a Figura 35 - Criando aplicativo.

App name

listaprodutos is available

Choose a region

United States

Add to pipeline...

Create app

Figura 35 - Criando aplicativo

Na próxima tela você encontrará informações de como subir sua aplicação para a nuvem usando a ferramenta git. Veja a Figura 36 - Instalação do Heroku CLI.

Install the Heroku CLI

Download and install the [Heroku CLI](#).

If you haven't already, log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

Create a new Git repository

Initialize a git repository in a new or existing directory

```
$ cd my-project/
$ git init
$ heroku git:remote -a listaprodutos
```

Deploy your application

Commit your code to the repository and deploy it to Heroku using Git.

```
$ git add .
$ git commit -am "make it better"
$ git push heroku master
```



You can now change your main deploy branch from "master" to "main" for both manual and automatic deploys, please follow the instructions [here](#).

Figura 36 - Instalação do Heroku CLI

Antes de continuar o envio da aplicação para a nuvem, precisamos fazer duas preparações. A primeira, trata-se de criar o banco de dados

na Nuvem da Heroku, e a segunda, refere-se a alteração do arquivo de configuração da aplicação para o ambiente de produção. Vamos a elas.

10.4 Preparando o Banco de Dados

Na mesma janela, role ao início da página e acesse a opção Resources. Clique sobre ela. Veja Figura 37 - Janela Resources.

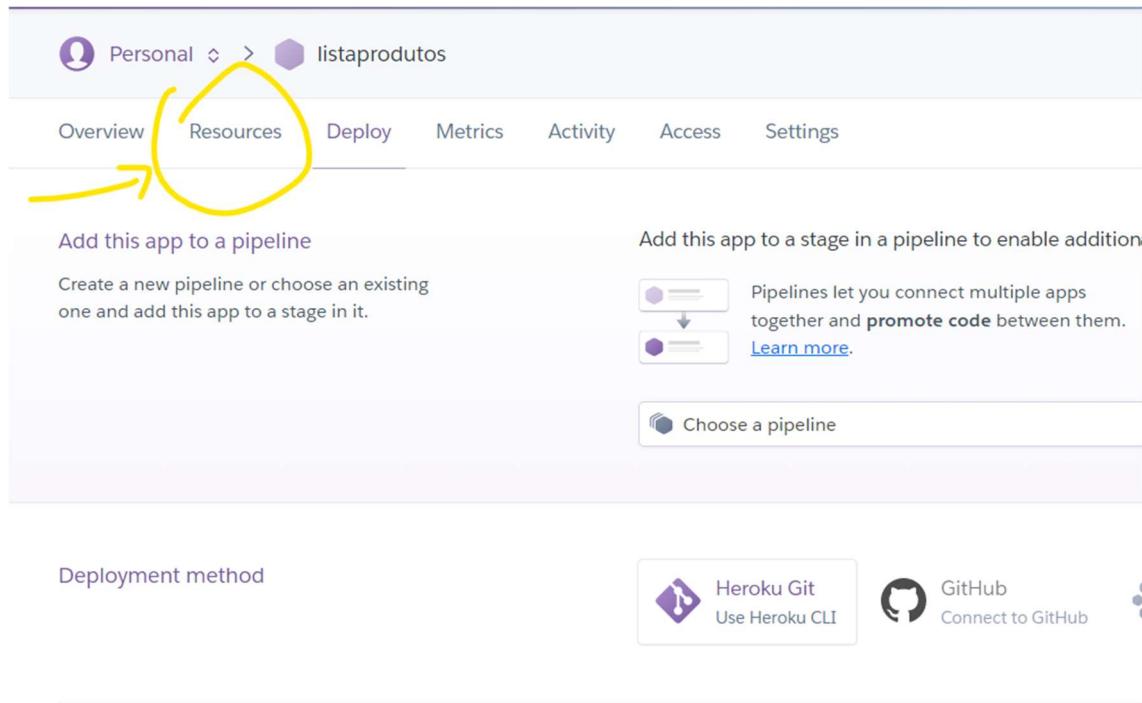


Figura 37 - Janela Resources

Ao surgir a nova tela, Figura 38 - Janela Add-ons, procure pela opção Add-ons e digite Mysql. Na lista de opções, selecione a opção ClearDB MySQL.

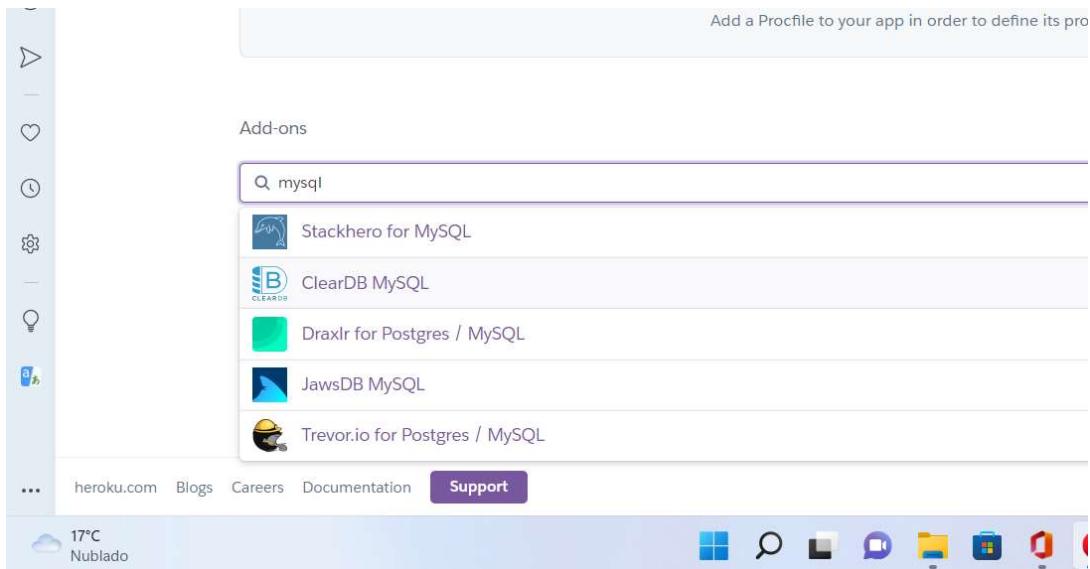


Figura 38 - Janela Add-ons

Se você não cadastrou até o momento um cartão de crédito, é agora momento que o deve fazer. Lembre-se que a escolha foi um plano “free”, e que depois dos testes você pode desabilitar para evitar cobranças desnecessárias. Se bem que o limite é muito grande para que nossa pequena aplicação o ultrapasse.

Link para o cadastro do cartão

<https://devcenter.heroku.com/categories/billing>.

Ou logado acesse:

<https://dashboard.heroku.com/account/billing>.

Depois de adicionar o cartão, volte e confirme a submissão do banco de dados.

Resultado da instalação:



Figura 39 - ClearDB MySQL instalado

10.5 Preparando o Aplicativo Java para Nuvem

Para preparar o aplicativo para hospedagem na Heroku devemos providenciar uma cópia do arquivo de propriedades `application.properties` e nomeá-lo como `application.properties-prod.properties`. Veja a Figura 40 - `application.properties-prod.properties`

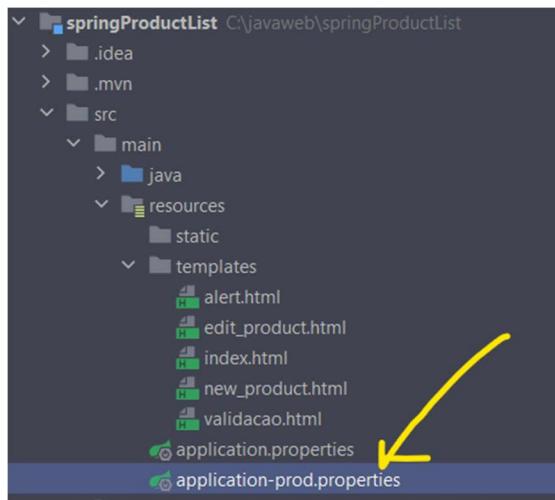


Figura 40 - `application.properties-prod.properties`

Após isto. Você deve modificar o arquivo, deixando inicialmente como abaixo:

```
#DATASOURCE
spring.datasource.url = 

#JPA
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = false
spring.jpa.open-in-view = true
spring.jpa.hibernate.use-new-id-generator-mappings = false

#THYMELEAF
spring.thymeleaf.cache=true
```

Observe que a propriedade `spring.datasource.url` está vazia. Aqui precisamos inserir a url do Heroku. Voltamos em breve neste ponto.

Agora crie um arquivo vazio na raiz do projeto, chamado ProcFile deve ficar no mesmo nível do pom.xml. Figura 41 - Arquivo Procfile.

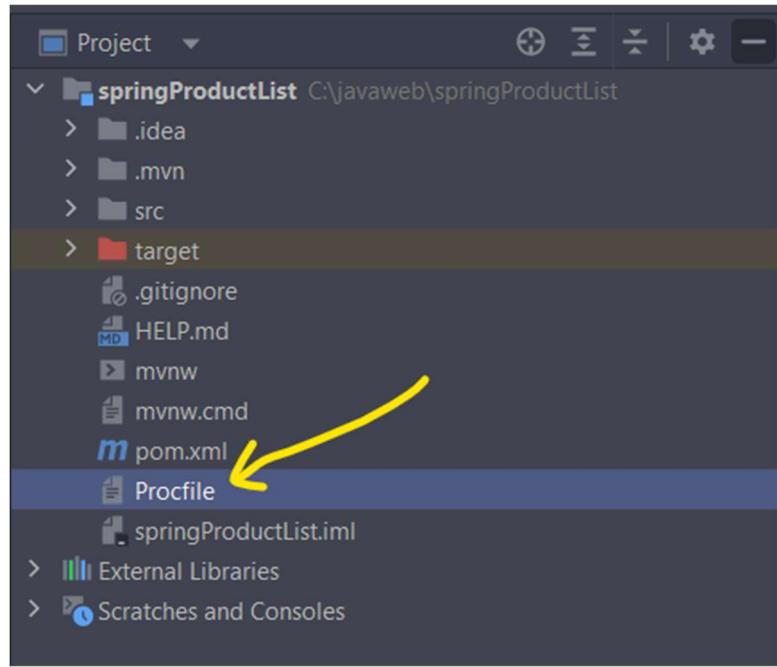


Figura 41 - Arquivo Procfile

No arquivo **Procfile**, você especifica os comandos que serão executados pelo aplicativo durante a inicialização. Segundo a Heroku nele você pode declarar vários tipos de processos, incluindo:

- O servidor web do seu aplicativo
- Vários tipos de processos de trabalho
- Um processo singleton, como um relógio
- Tarefas a serem executadas antes da implantação de uma nova versão

Digite a seguinte linha dentro deste arquivo (uma única linha):

```
java -Dspring.profiles=prod -jar target/springProductList.jar
```

Nesta linha informamos que se trata de um aplicativo Java pra web que deverá ser executado na porta **\$PORT**, variável que o Heroku definirá qual a porta que será executada a aplicação. Já a propriedade **-Dspring.profiles=prod** é a forma pela qual o Spring será inicializado em produção, cujo arquivo alteramos anteriormente. Por fim indicamos onde será o Heroku armazenará o arquivo jar quando a aplicação for compilada na nuvem.

10.6 Enviando a aplicação para a nuvem

Primeiramente antes de enviar o arquivo para a nuvem precisamos completar as informações do banco de dados no arquivo de propriedades do ambiente de produção, já que o local aponta para nossa máquina, e desejamos que o aplicativo em produção aponte para o aplicativo web armazenado no Heroku.

10.7 Logando no heroku login

Pela linha de comando (prompt de comando) você deve digitar o comando heroku login.

```
C:\>heroku login
» Warning: heroku update available from 7.53.0 to 7.60.2.
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.herokuapp.com/auth/cli/browser/656dfab1-0b39-4fd7-b88a-2f61b307Y.g2gDbQAAA0xNzcuOTUuMTQyLjQ3bgYA8BXAfIBYgABUYA.0YaHc-vfznBhiQ-Ha4RL1zC_hxPzJzWJ6CWI1D-bpjw
Logging in... done
Logged in as prof.maromo@gmail.com
```

Você deve usar um mecanismo de segurança assim que for redirecionada para que ele complete a instalação, eu use o gerador de senha e instalei o Google Authenticator para ler o QR-Code.

Pronto!

Uma vez logado, certifique-se no intelliJ Idea que todos os arquivos do projeto são salvos e vamos subir a aplicação para a nuvem, usando o git. Para isso, ainda no prompt de comando, navegue para a pasta onde está o seu aplicativo. No meu caso, o caminho é c:\javaweb\springProductList.

```
C:\>cd \javaweb\springProductList
```

```
C:\javaweb\springProductList>
```

Agora vamos vincular nossa aplicação ao diretório do git. As instruções que serão mostradas a seguir, são as mesmas que se encontram na página de instalação do Heroku CLI. Veja a Figura 36 - Instalação do Heroku CLI.

Digite o comando: git init

```
C:\javaweb\springProductList>git init  
Initialized empty Git repository in C:/javaweb/springProductList/.git/
```

Em seguida digite o comando: heroku git:remote -a listaprodutos

```
C:\javaweb\springProductList>heroku git:remote -a listaprodutos  
» Warning: heroku update available from 7.53.0 to 7.60.2.  
set git remote heroku to https://git.heroku.com/listaprodutos.git
```

Pronto, nosso diretório está vinculado a aplicação na nuvem. Falta apenas definir os dados de acesso ao banco de dados. Para isso, feche o prompt de comando e abra o aplicativo git bash.

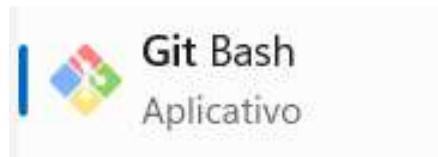


Figura 42 - Aplicativo Git Bash

Ele é mais completo e possui o comando de opção que vamos usar. Primeiro navegue para o diretório da aplicação. Padrão Linux. Veja a Figura 43 - Trocando de diretório pelo Git Bash como exemplo no meu caso.

```
marco@maromo MINGW64 ~  
$ cd /c/javaweb/springProductList/  
  
marco@maromo MINGW64 /c/javaweb/springProductList (master)  
$ |
```

Figura 43 - Trocando de diretório pelo Git Bash

Definindo o caminho do nosso banco. Digite o comando: `heroku config`

`| grep CLEARDB_DATABASE_URL` no diretório (master) da aplicação.

Veja Figura 44 - Propriedade de conexão com o MySQL.

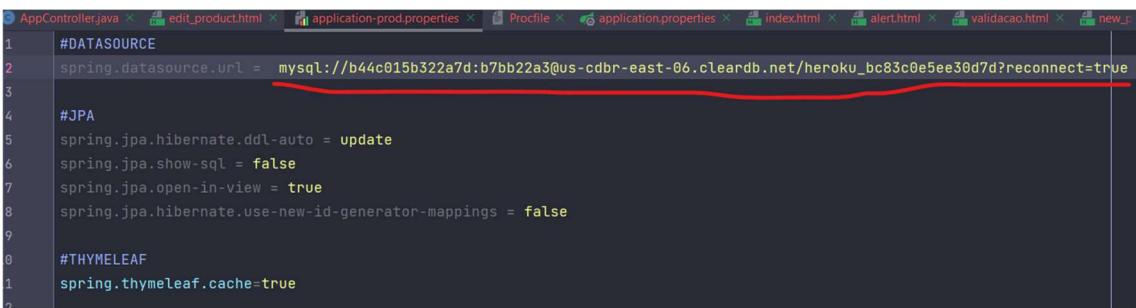
```

marco@maromo MINGW64 /c/javaweb/springProductList (master)
$ heroku config | grep CLEARDB_DATABASE_URL
» Warning: heroku update available from 7.53.0 to 7.60.2.
CLEARDB_DATABASE_URL: mysql://b44c015b322a7d:b7bb22a3@us-cdbr-east-06.cleardb.net/heroku_bc83c0e5ee30d7d?reconnect=true
marco@maromo MINGW64 /c/javaweb/springProductList (master)
$ |

```

Figura 44 - Propriedade de conexão com o MySQL

Copie essa linha para o arquivo de propriedades do nosso projeto. Veja a Figura 45 - Arquivo de propriedades atualizado.



```

#DATASOURCE
spring.datasource.url = mysql://b44c015b322a7d:b7bb22a3@us-cdbr-east-06.cleardb.net/heroku_bc83c0e5ee30d7d?reconnect=true
#JPA
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = false
spring.jpa.open-in-view = true
spring.jpa.hibernate.use-new-id-generator-mappings = false
#THYMELEAF
spring.thymeleaf.cache=true

```

Figura 45 - Arquivo de propriedades atualizado

10.8 Passos Finais

Volte para o git bash e digite o comando `git add .`

```

marco@maromo MINGW64 /c/javaweb/springProductList (master)
$ git add .
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in .mvn/wrapper/maven-wrapper.properties.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in mvnw.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in mvnw.cmd.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in pom.xml.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/main/java/com/example/springproductlist/SpringProductListController.java
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/main/resources/application-prod.properties.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/main/resources/application.properties.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/test/java/com/example/springproductlist/SpringProductListControllerTest.java
The file will have its original line endings in your working directory

marco@maromo MINGW64 /c/javaweb/springProductList (master)
$ |

```

O parâmetro “.” depois do comando `git add`, indica que é para incluir todos os arquivos deste caminho (à partir do diretório que se executa).

A seguir será executado o comando que é uma das funções principais do Git. Antes de usar o comando `git add` é necessário selecionar as alterações que vão ser preparadas para o próximo commit. Então, `git`

commit é usado para criar um instantâneo das alterações preparadas em um cronograma de um histórico de projetos do Git.

Digite o comando `git commit -am "Primeiro Git"`. Veja o resultado da execução do comando na Figura 46 - Resultado do Git commit.

```
marco@maromo MINGW64 /c/javaweb/springProductList (master)
$ git commit -am "Primeiro Git"
[master (root-commit) de93cf2] Primeiro Git
 20 files changed, 1047 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 .mvn/wrapper/maven-wrapper.jar
 create mode 100644 .mvn/wrapper/maven-wrapper.properties
 create mode 100644 Procfile
 create mode 100644 mvnw
 create mode 100644 mvnw.cmd
 create mode 100644 pom.xml
 create mode 100644 src/main/java/com/example/springproductlist/SpringProductListApplication.java
 create mode 100644 src/main/java/com/example/springproductlist/product/AppController.java
 create mode 100644 src/main/java/com/example/springproductlist/product/Product.java
 create mode 100644 src/main/java/com/example/springproductlist/product/ProductRepository.java
 create mode 100644 src/main/java/com/example/springproductlist/product/ProductService.java
 create mode 100644 src/main/resources/application-prod.properties
 create mode 100644 src/main/resources/application.properties
 create mode 100644 src/main/resources/templates/alert.html
 create mode 100644 src/main/resources/templates/edit_product.html
 create mode 100644 src/main/resources/templates/index.html
 create mode 100644 src/main/resources/templates/new_product.html
 create mode 100644 src/main/resources/templates/validacao.html
 create mode 100644 src/test/java/com/example/springproductlist/SpringProductListApplicationTests.java
```

Figura 46 - Resultado do Git commit

Agora finalmente para enviar nosso projeto para o Heroku, digite o comando `git push heroku master`. Se tudo correr bem, você terá como resultado o que se apresenta abaixo:

```
remote: [INFO] -----
remote: [INFO] BUILD SUCCESS
remote: [INFO] -----
remote: [INFO] Total time: 11.915 s
remote: [INFO] Finished at: 2022-08-08T19:24:57Z
remote: [INFO] -----
remote: -----> Discovering process types
remote:      Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:      Done: 91M
remote: -----> Launching...
remote:      Released v5
remote:      https://listaprodutos.herokuapp.com/ deployed to Heroku
remote:
remote: This app is using the Heroku-20 stack, however a newer stack is available.
remote: To upgrade to Heroku-22, see:
remote: https://devcenter.heroku.com/articles/upgrading-to-the-latest-stack
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/listaprodutos.git
 * [new branch]      master -> master

marco@maromo MINGW64 /c/javaweb/springProductList (master)
```

10.9 Navegando na aplicação

Acesse primeiramente o Dashboard no Heroku. Veja Figura 47 - Dashboard atual.

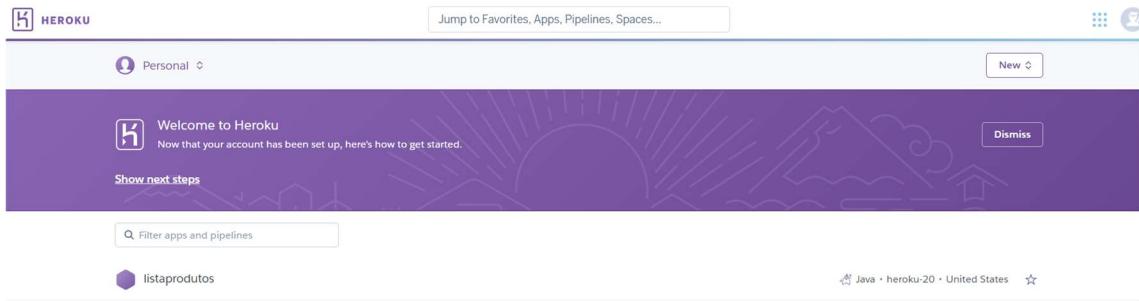


Figura 47 - Dashboard atual

Como vimos, nosso app já está na nuvem. Clique sobre ele e na janela que se abrirá clique sobre o botão **Open App** para abrir nosso aplicativo no navegador. E efetuar os testes.

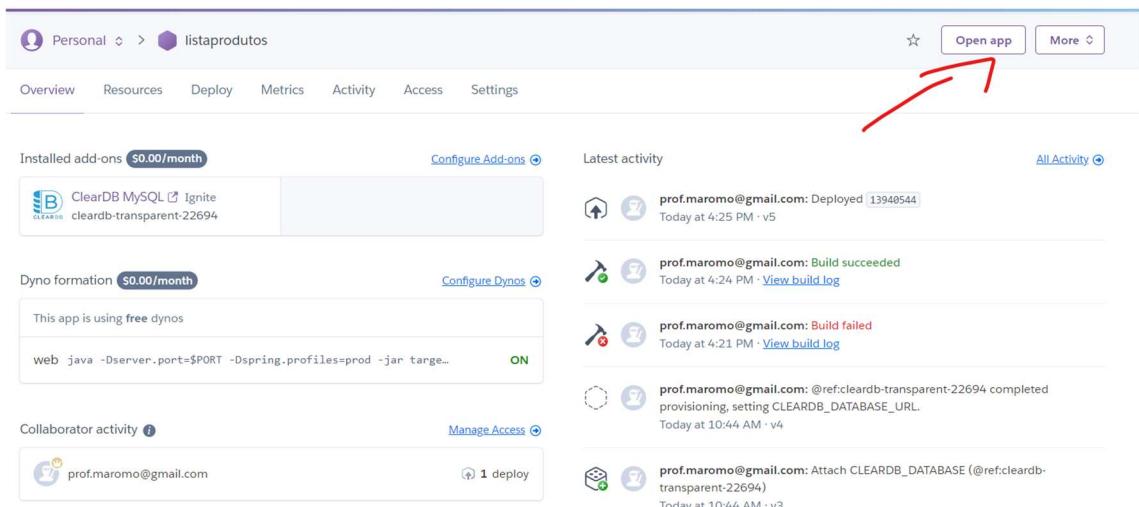


Figura 48 - Abrindo o aplicativo

Pronto, o resultado será nosso projeto disponibilizado na Web.

Agora é só testar.

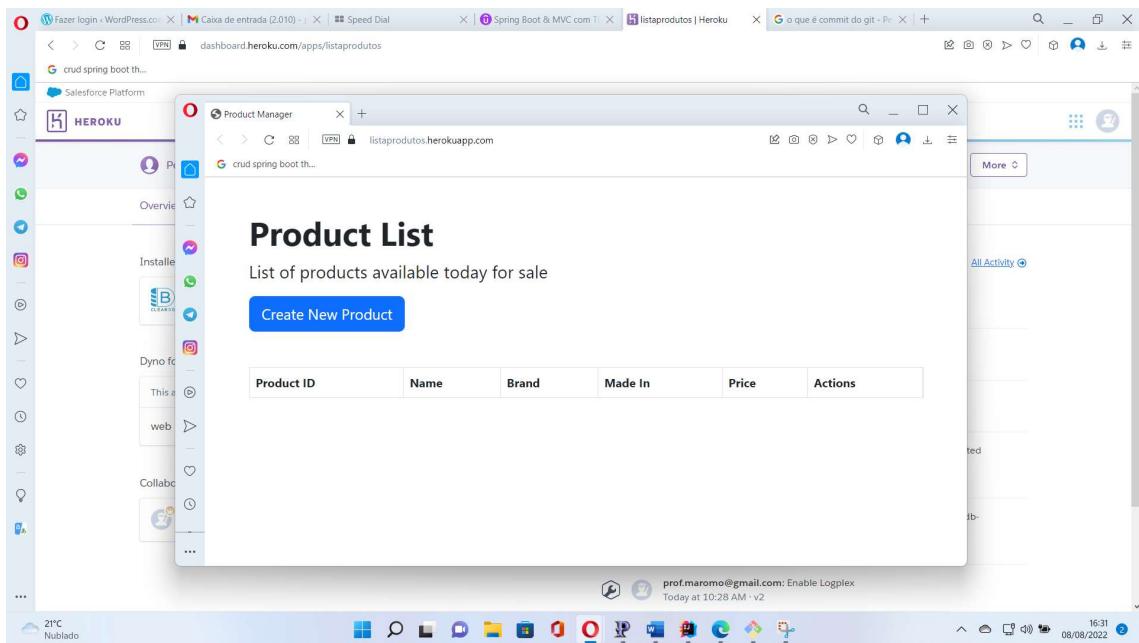


Figura 49 - Resultado da execução na nuvem

Para você que leu e testou este tutorial até aqui, agradeço muito.

Se encontrar erros, por favor, reporte para o email:
professormoraes@gmail.com