
Spring Boot

2.7.3



Um facilitador para a criação de aplicativos independentes baseados em Spring.
base: // https://github.com/maromo71/palestra_ams

Spring Boot

O Spring Boot é um framework Open Source que nasceu a partir do Spring framework com o objetivo de facilitar as configurações iniciais de um projeto.

Ou seja, antes, os desenvolvedores enfrentavam uma grande dificuldade: a perda de tempo configurando um projeto ao invés de desenvolver.



Histórico

2003

Rod Johnson criou o Spring framework, que veio com o intuito de simplificar essas configurações para aplicações web.

2012

Surgiu o Spring Boot, que fez decolar a plataforma Spring, pois ele nada mais é que uma extensão do Spring Framework

2003

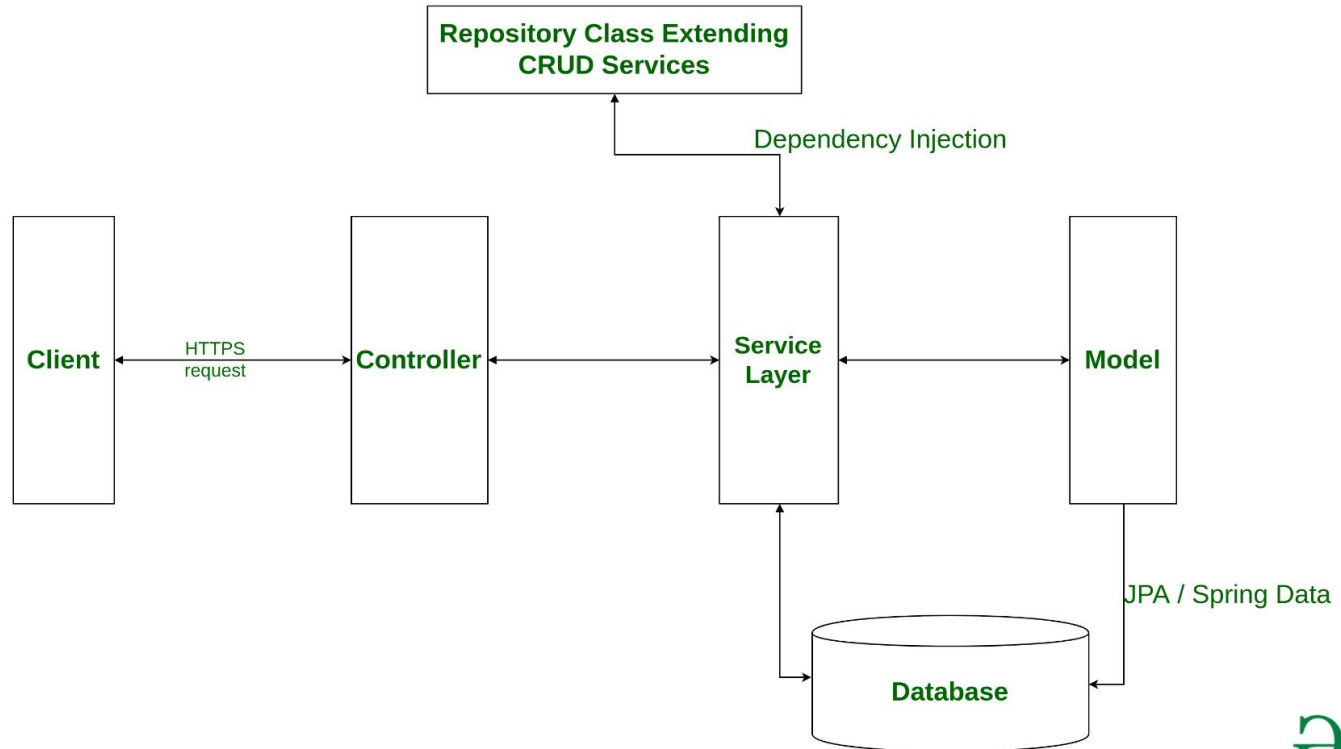
2022

Mas se a aplicação fosse muito grande, as configurações ficavam grandes e complexas também, pois o Spring Framework não isenta configurações, ele apenas simplifica.

Agosto de 2022
Versão atual:

2.7.3 - Ago / 2022

Spring Boot flow architecture



—

Mas como iniciamos um projeto Spring Boot ?


A plataforma Spring disponibilizou a ferramenta **Spring Initializr**, que possibilita ao desenvolvedor realizar toda a configuração inicial de um projeto Spring com alguns cliques.



Neste exemplo, vamos começar com algo **inesperado**.

Vamos criar um projeto Spring Boot a partir do Spring Initializr apresentado no próximo slide.

Configurando um ponto só!



spring initializr

Project

☒ Maven Project

☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT) ☐ 3.0.0 (M4) ☐ 2.7.4 (SNAPSHOT) ☒ 2.7.3

☐ 2.6.12 (SNAPSHOT) ☐ 2.6.11

Project Metadata

Group

Artifact

Name

Description

Projeto de Exemplo

Nestes slides, você verá a desenvolver um aplicativo Java para Web que gerencia informações em um banco de dados – com operações CRUD padrão: Criar, Recuperar, Atualizar e Excluir dados do Banco. Utilizamos as seguintes tecnologias:

- Spring Boot
 - Spring Web
 - Spring Data JPA
 - Spring Bean Validation
 - Hibernate
 - Thymeleaf
 - Bootstrap
 - Webjars
 - MySQL
-

[Why Spring](#) ▾[Learn](#) ▾[Projects](#) ▾

Web Applications

Spring makes building web applications fast and hassle-free. By removing much of the boilerplate code and configuration associated with web development, you get a modern web programming model that streamlines the development of server-side HTML applications, REST APIs, and bidirectional, event-based systems.

Spring Web ou

Spring Web MVC

Um aplicativo escrito em Java que permite a visualização de dados geográficos armazenados em um servidor remoto.

Simplifica o desenvolvimento de aplicativos HTML do lado do servidor, APIs REST e sistemas bidirecionais baseado em eventos

Spring Data JPA

Parte da família do Spring Data, facilita a implementação de repositórios baseados em JPA.

Este módulo lida com suporte aprimorado para camadas de acesso a dados baseadas em JPA.

Facilita a criação de aplicativos baseados em Spring que usam tecnologias de acesso a dados.

Spring Data JPA 2.7.2

OVERVIEW

LEARN

SUPPORT

SAMPLES

Spring Data JPA, part of the larger Spring Data family, makes it easy to e repositories. This module deals with enhanced support for JPA based da to build Spring-powered applications that use data access technologies

7. Validation, Data Binding, and Type Conversion

7.1 Introduction

JSR-303 Bean Validation

The Spring Framework supports JSR-303 Bean Validation adapters.

An application can choose to enable JSR-303 Bean Validation adapters to meet its validation needs.

An application can also register additional Spring `Validator` instances. This may be useful for plugging in validation logic without the use of

Spring Bean Validation

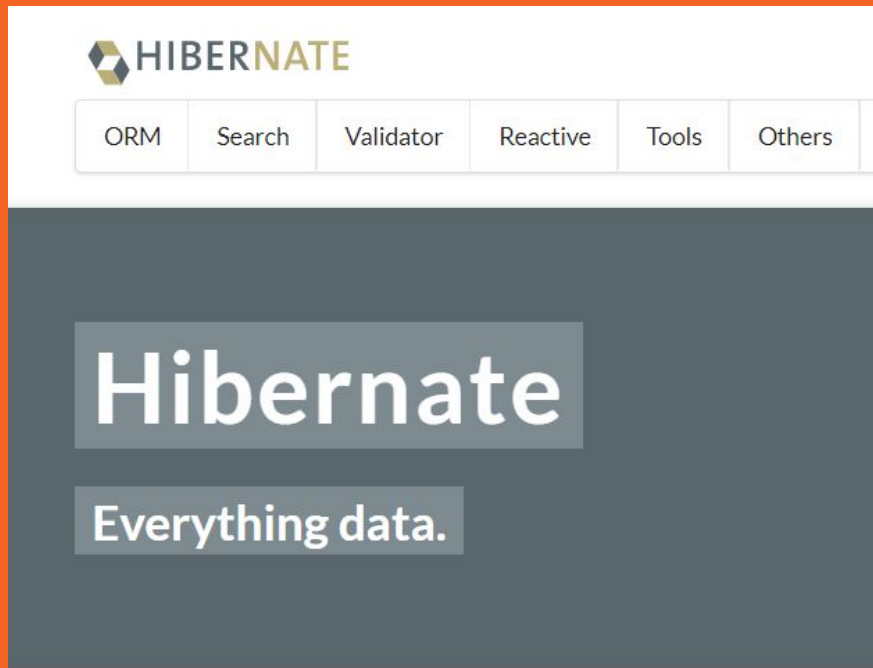
A API de validação é uma especificação Java que é usada para aplicar restrições no modelo de objeto por meio de anotações.

Aqui, podemos validar um comprimento, número, expressão regular, etc.

Hibernate

O Hibernate é uma ferramenta de mapeamento relacional de objetos (ORM) de código aberto que fornece uma estrutura para mapear modelos de domínio orientados a objetos para bancos de dados relacionais para aplicativos da web.

O mapeamento relacional de objetos é baseado na containerização de objetos e na abstração que fornece essa capacidade. A abstração possibilita endereçar, acessar e manipular objetos sem precisar considerar como eles estão relacionados às suas fontes de dados.





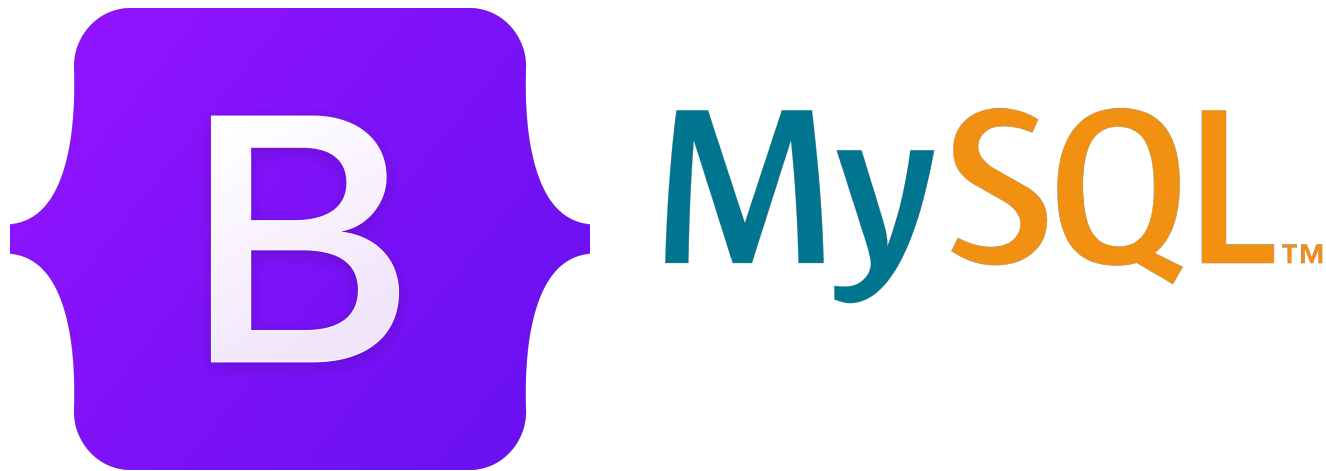
ThymeLeaf

Com módulos para Spring Framework, uma série de integrações com suas ferramentas e a capacidade de conectar sua própria funcionalidade, o Thymeleaf é ideal para o desenvolvimento web em HTML5 com JVM.

Outros

Bootstrap: framework front-end que fornece estruturas de CSS para a criação de sites e aplicações responsivas de forma rápida e simples.

MySQL: Sistema Gerenciador de Banco de Dados que utilizaremos para este exemplo.



—

O projeto de amostra
deste material poderá
ser baixado no github
após este **code labs.**

1. Banco

No Workbench crie o Banco (sales).

→ **Table (tabela)**

Crie a tabela "product" no banco "sales".
Veja figura no próximo slide,

2. Table: product

[illegible]

3. Aplicativo: springBootList

New Project


Empty Project

Generators

- Maven Archetype
- Java Enterprise
- Spring Initializr**
- JavaFX
- Quarkus
- Micronaut
- Ktor
- Kotlin Multiplatform
- Compose Multiplatform
- HTML
- React
- Express
- Angular CLI
- IDE Plugin
- Android

Server URL: start.spring.io

Name:

Location: 

Project will be created in: C:\mydirectory\springBootList

☐ Create Git repository

Language: ☐ Java ☐ Kotlin ☐ Groovy

Type: ☐ Maven ☐ Gradle

Group:

Artifact:

Package name:

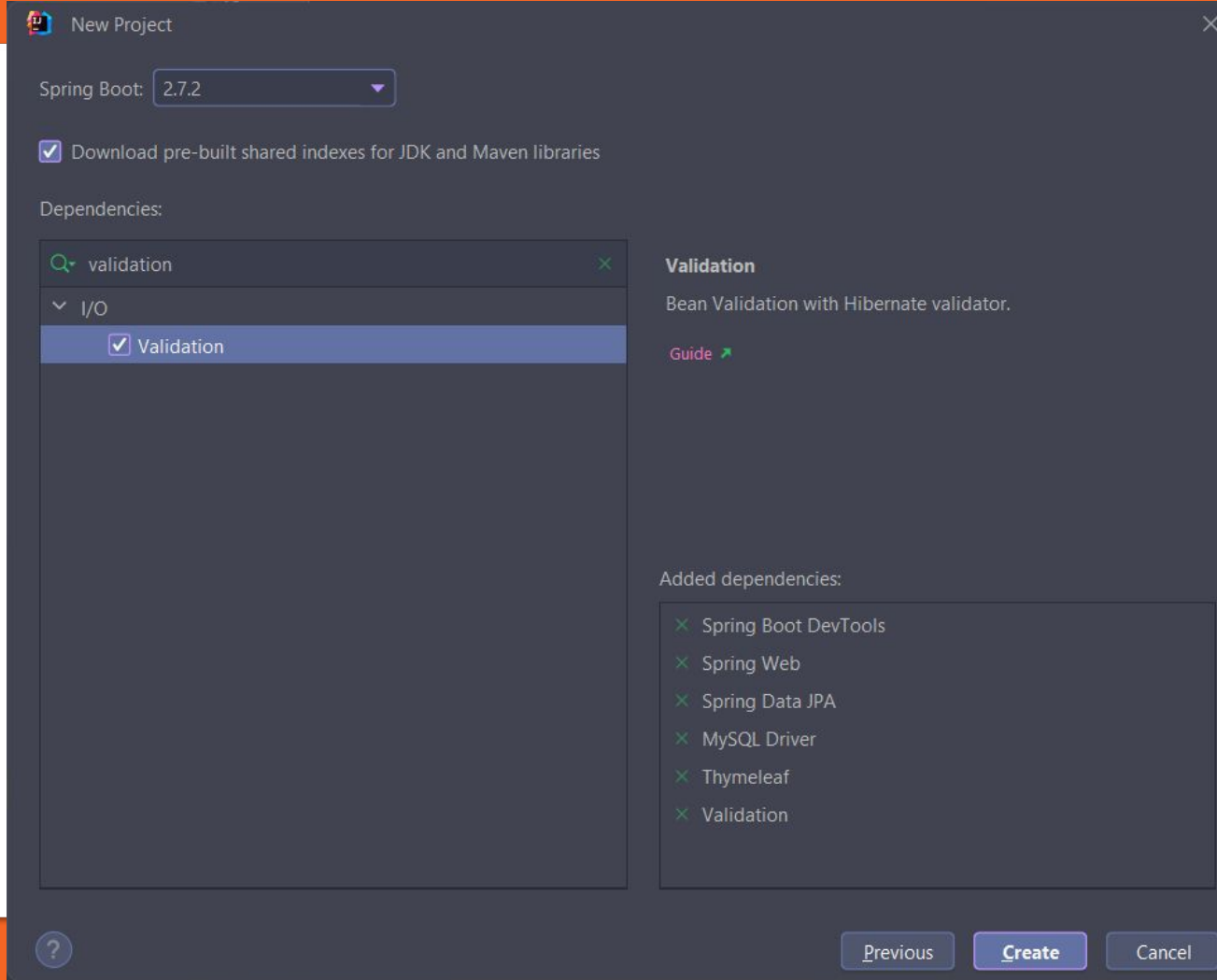
JDK:

Java:

Packaging: ☐ Jar ☐ War

[Next](#) [Cancel](#)

4. Dependências



—

Depois de criado o projeto, adicione **as demais dependências** que serão necessárias ao seu projeto. O arquivo pom.xml contém elas. Procure adicionar as versões que foram indicadas para **não ter problemas** de compatibilidade com este tutorial.



Nota

Observe cuidadosamente as versões indicadas, usando as mesmas.

Essa medida se faz necessária, pois podem haver modificações em versões futuras e você teria que adaptar, por vezes, seu aplicativo.

Dependências adicionadas

```
<!--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>
```

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.



Dica

Qualquer problema na execução, efetue o download do projeto no endereço disponibilizado na palestra.

Nota

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`.



5. Nosso Main

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

```
package net.maromo;

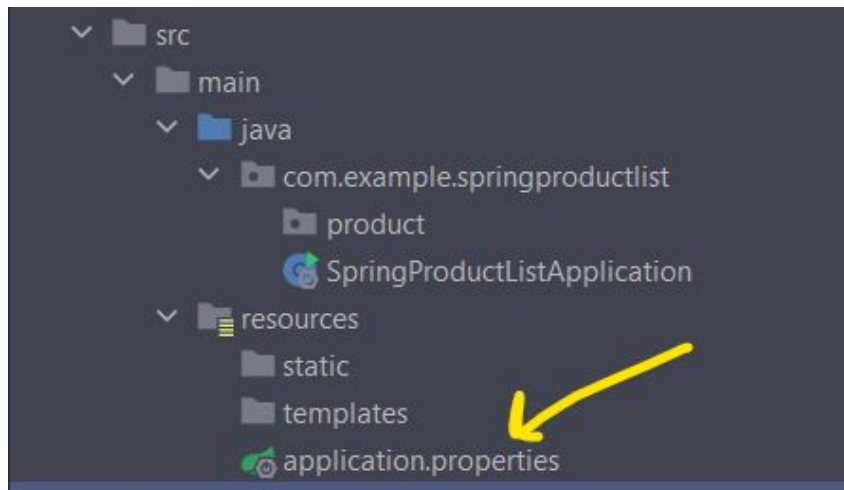
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringProductListApplication {

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

Configure o Arquivo de Propriedades

Configure ou crie o arquivo **application.properties** no diretório **src/main/resources**.



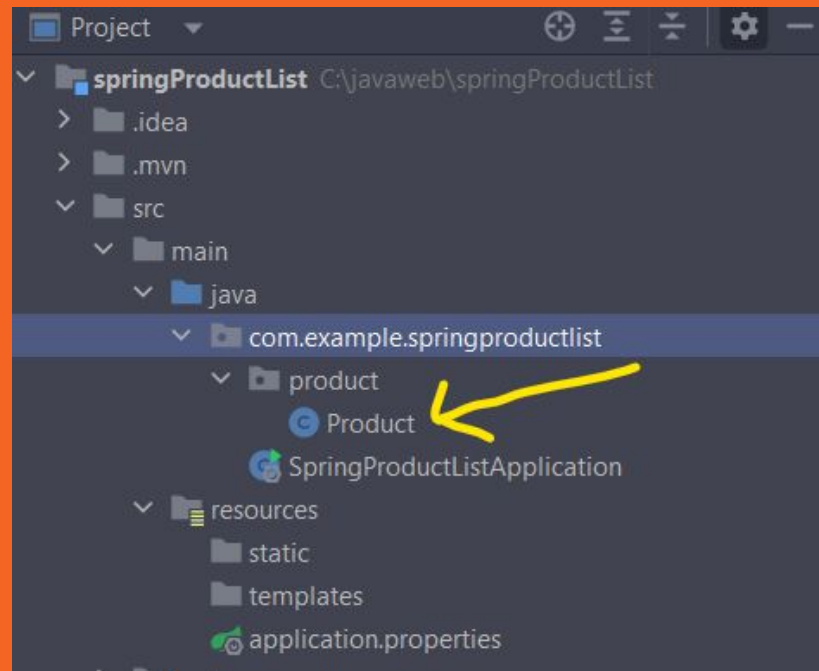
application.properties

```
#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
```

Definindo o Modelo



Model

Atenção para as Anotações do
BeanValidation

Product		
m ?	Product(Long, String, String, String, float)	
m ?	Product()	
f 🔒	name	String
f 🔒	id	Long
f 🔒	brand	String
f 🔒	price	float
f 🔒	madein	String
m 🔒	getPrice()	float
m 🔒	setPrice(float)	void
m 🔒	getName()	String
m 🔒	setName(String)	void
m 🔒	getId()	Long
m 🔒	setId(Long)	void
m 🔒	getMadein()	String
m 🔒	getBrand()	String
m 🔒	setMadein(String)	void
m 🔒	setBrand(String)	void

```
package net.maromo.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;  
}
```

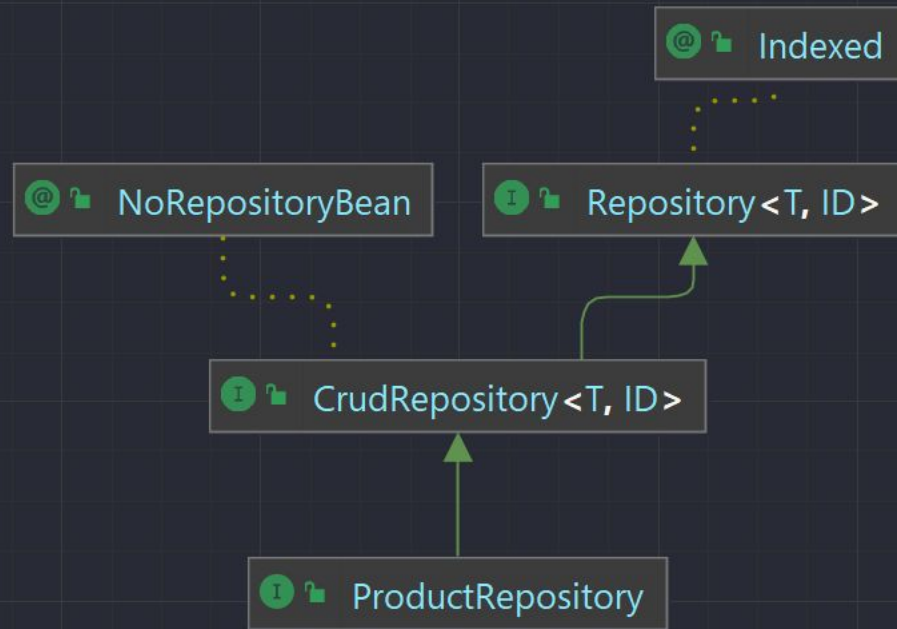
```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
public Long getId() {  
    return id;  
}
```

```
//Demais gets/Sets
```

```
}
```

Repository

Interface que expande a
Interface CrudRepository



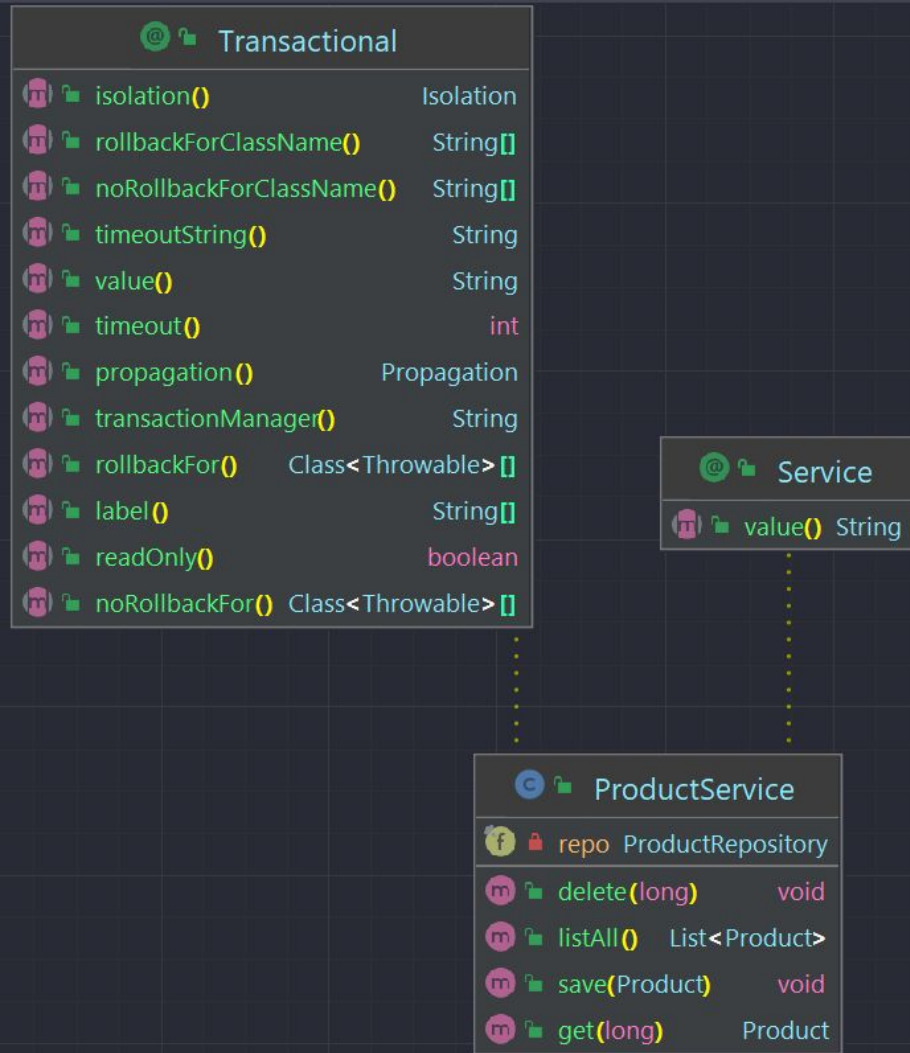
```
package net.maromo.product;

import org.springframework.data.repository.CrudRepository;

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

Service

Camada de Serviço



Camada de Serviço

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`.

```
package net.maromo.product;
```

```
import java.util.List;
```

```
import org.springframework.stereotype.Service;
```

```
import org.springframework.transaction.annotation.Transactional;
```

```
@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);
```

```
    }
```

```
}
```

Controller

Class ApplicationController

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.

```
package net.maromo.product;
```

```
import java.util.List;
```

```
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;
```

```
@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:/";
```

```
}
```

```
}
```

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.



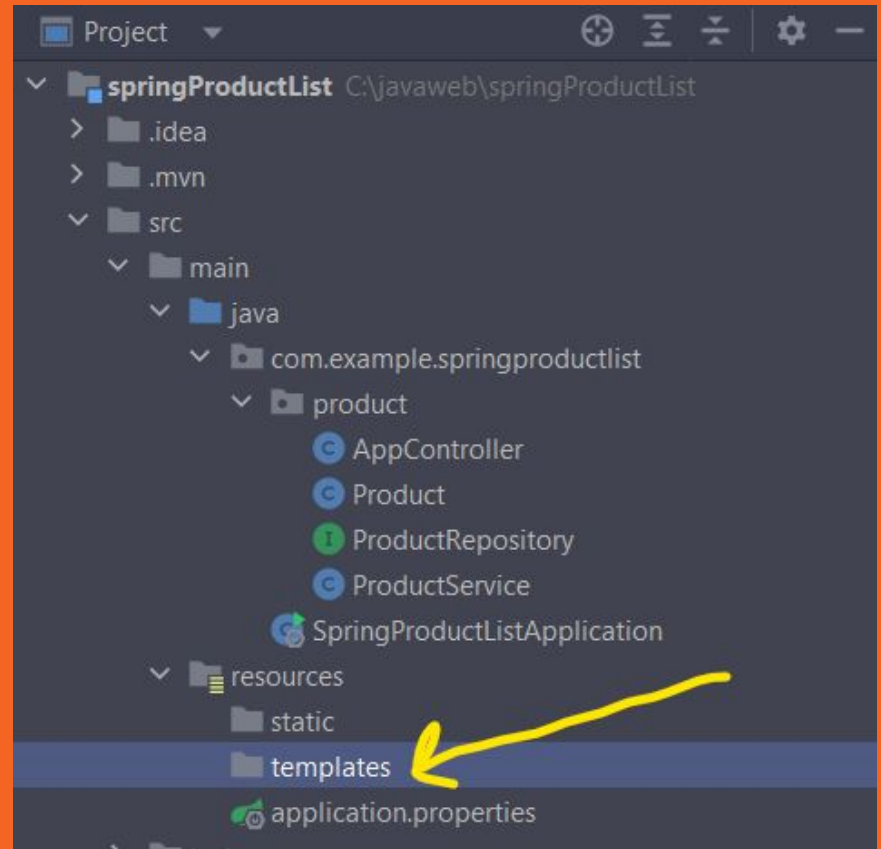
Usando o Thymeleaf

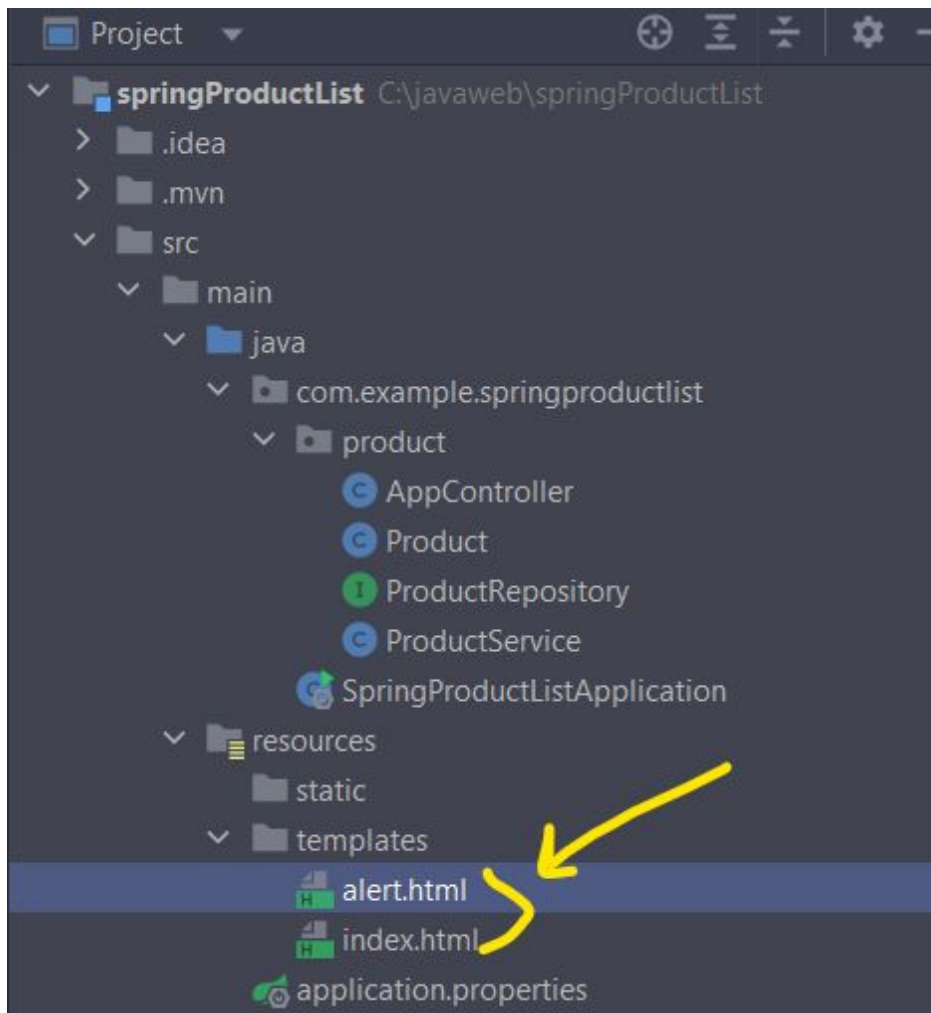
Thymeleaf

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.

resources/
templates





Neste exemplo usamos **Thymeleaf** ao invés de JSP, para isso no diretório de templates em **src/main/resources** (usado para armazenar arquivos de templates HTML) crie um arquivo chamado **alert.html** e outro chamado **index.html**.

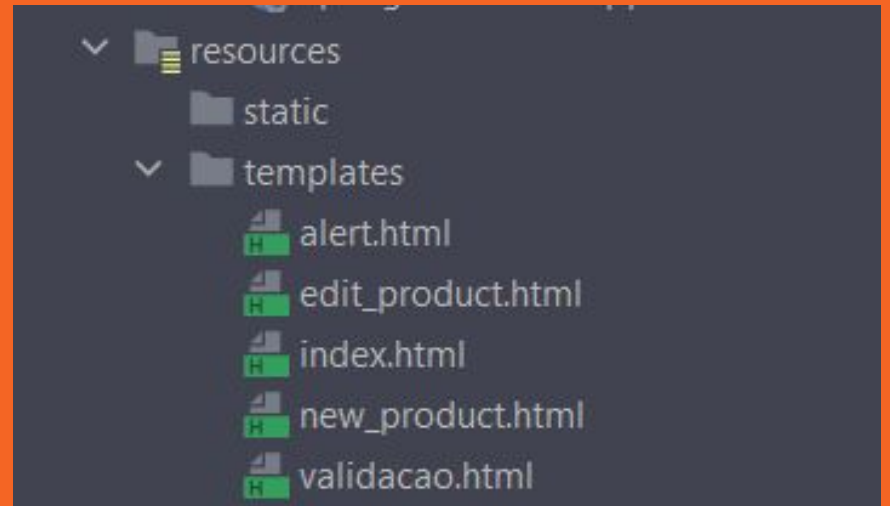
Arquivo alert.html

```
<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>
```

Nota: para facilitar, os
códigos dos arquivos
html estão disponíveis
no github.
Serão comentados no
code labs.

Estrutura Final



O que as pessoas estão dizendo

Sem Spring Framework, o código do aplicativo tende a ser fortemente acoplado (interdependente), o que não é considerado uma boa prática de codificação.

O Spring sabe que o desenvolvedor deseja criar uma instância de uma classe e que o ele deve gerenciá-la.

No núcleo do Spring Framework, os objetos são criados, conectados, configurados e gerenciados ao longo de seu ciclo de vida.

The background features a grayscale image of a hand holding a central node of a network diagram. The network consists of numerous circular nodes connected by thin lines, radiating outwards from the hand. The overall aesthetic is technical and digital.

**Fim
Obrigado**

Maromo