

FRAMEWORKS JAVA, .NET &
WEBSERVICES

FRONT-END COM **SPRING WEB**



7A

LISTA DE FIGURAS

Figura 1 – Estrutura do Projeto	5
Figura 2 – Ciclo de Vida de uma requisição no Spring Web	7
Figura 3 – Tela de cadastro de um produto	20
Figura 4 – Tela de listagem de produtos	21
Figura 5 – Modal de confirmação de exclusão de produto	21
Figura 6 – Teste de Validação no Cadastro	25

EMANIP

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Código do ProdutoController.....	9
Código-fonte 2 – Código da Página de Template.....	12
Código-fonte 3 – Dependência do Layout no pom.xml.....	13
Código-fonte 4 – Dependência do WebJars no pom.xml	15
Código-fonte 5 – Código da página form.html	17
Código-fonte 6 – Código da página lista.html.....	19
Código-fonte 7 – Código da Entidade Produto	22
Código-fonte 8 – Código do ProdutoController.....	23
Código-fonte 9 – Código de exibição de erros de validação no HTML.....	24
Código-fonte 10 – Código de exibição de erro específico de validação no HTML25	
Código-fonte 11 – Código de formatação de conteúdo no HTML	26

SUMÁRIO

1 <i>FRONT-END</i> COM SPRING WEB	5
1.1 Recapitulação sobre <i>Back-end</i> em Spring Boot	5
1.2 Spring Web.....	6
1.3 Spring Web – Controller	8
1.4 Spring Web – Template.....	11
2 SPRING WEB – WEBJARS	14
2.1 Thymeleaf.....	15
2.2 Teste da Interface Web	20
2.3 Teste das Validações	22
2.4 Mensagens e formatação no Spring.....	24
CONCLUSÃO.....	27
REFERÊNCIA	28

1 FRONT-END COM SPRING WEB

1.1 Recapitulação sobre *Back-end* em Spring Boot

No capítulo anterior construímos um projeto completo de *back-end*, utilizando o Spring Boot em conjunto com o Spring MV. Note que não trabalhamos aspectos de interface (ou da *view*) por serem de *front-end*, que é o foco desta etapa do material.

Vamos recapitular a primeira etapa do projeto?

Inicialmente, construímos o projeto utilizando o Spring Initializr, com módulos de banco de dados, web, REST e Spring Data JPA ativados. O objetivo, até o final deste capítulo, é apresentar a estrutura exibida na Figura “Estrutura do Projeto”:

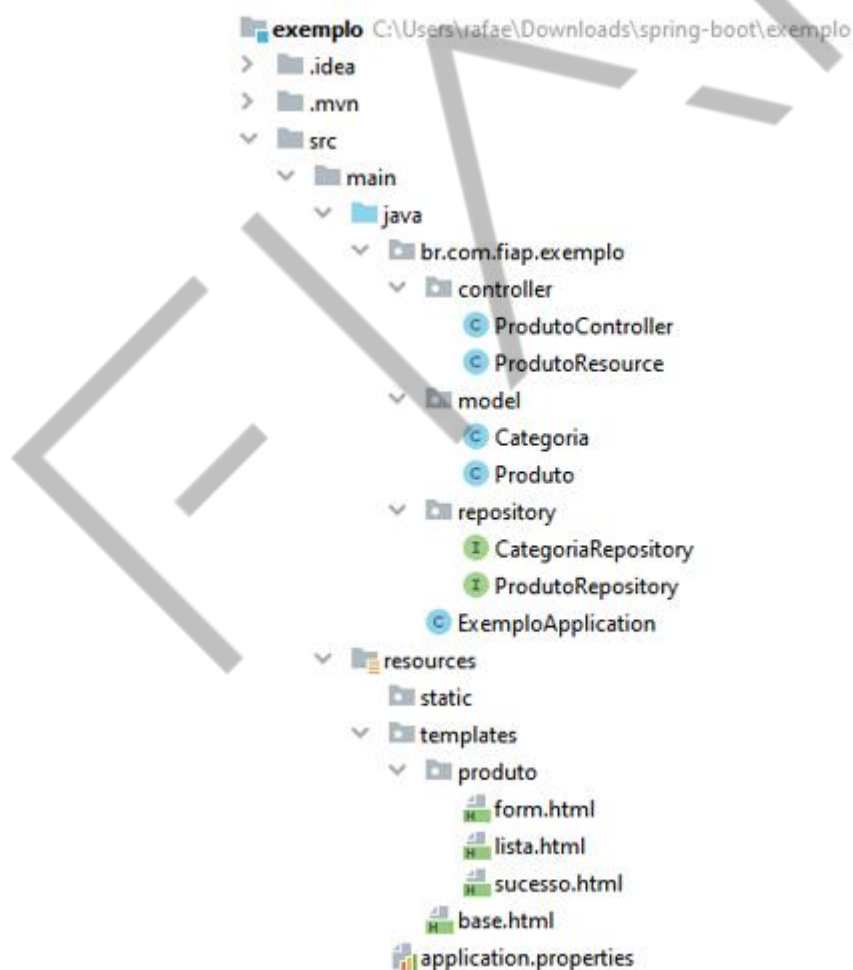


Figura 1 – Estrutura do Projeto
Fonte: Elaborado pelo autor (2019)

Inicialmente, configuramos o H2 Database (banco de dados *embedded* com interface web). Depois, criamos a entity Produto, mapeada em uma tabela do banco de dados, e o repositório ProdutoRepository, que nada mais é do que uma extensão de uma interface JpaRepository.

Por fim, a API REST foi criada e todos os verbos do CRUD (GET, GET{id}, POST, PUT e DELETE) foram testados com uma ferramenta chamada Postman.

Agora criaremos toda a parte de interface web, mas antes precisamos conhecer a teoria e como o Spring vai tratar as requisições web e gerar a página HTML.

1.2 Spring Web

O Spring Web é um importante módulo do Spring Framework, que é uma estrutura de aplicativos Java amplamente utilizada para o desenvolvimento de aplicativos empresariais. O Spring Web tem como objetivo receber, tratar e enviar solicitações web em protocolos como HTTP e HTTPS, garantindo que todo o processamento ocorra de forma transparente e eficiente tanto para o usuário final quanto para o desenvolvedor. O uso do Spring Web permite que o desenvolvedor concentre-se mais na lógica de negócios do aplicativo, em vez de preocupar-se com a manipulação e tratamento de solicitações e respostas HTTP. Isso resulta em maior produtividade e eficiência na criação de aplicativos web em Java. Além disso, o Spring Web oferece recursos avançados de segurança, escalabilidade e desempenho para atender às necessidades de aplicativos empresariais críticos.

Isso significa que o Spring Web permite que o desenvolvedor possa criar aplicações web robustas e eficientes, sem se preocupar com detalhes técnicos complexos, como o processo de receber e enviar requisições. O Spring Web trabalha para que essa interação entre cliente e servidor ocorra de forma clara e eficiente, sempre garantindo a melhor experiência possível para o usuário final. Em resumo, o Spring Web é um importante aliado para o desenvolvimento de aplicações web modernas e eficientes.

O Ciclo de Vida de uma requisição dentro do Spring Web está exemplificado na Figura “Ciclo de Vida de uma requisição no Spring Web”:

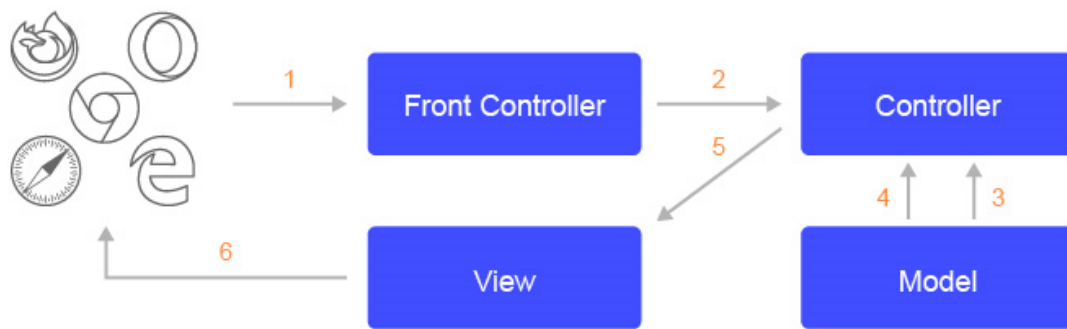


Figura 2 – Ciclo de Vida de uma requisição no Spring Web
Fonte: Yamamoto (2019)

- 1) Quando um usuário faz uma requisição, é interceptado pelo **Front Controller** do Spring.
- 2) O **Front Controller** redireciona a requisição para o **Controller** correto dentro do projeto.
- 3) O **Controller** acessa o **Model** (se necessário) para processar as regras de negócio, entrar no banco de dados etc.
- 4) O **Model** (se necessário) retorna as informações ou os resultados para o **Controller**.
- 5) O **Controller** envia as informações e o nome da *view* (página) para o Spring processar e montar a página final.
- 6) A página final (em HTML) é devolvida ao usuário.

Existem algumas diferenças entre o uso de um **RestController** e o **Controller** visto agora. A principal delas é formato de troca de informações. No REST, é utilizado JSON como formato, enquanto no Controller, a resposta acaba por ser uma página HTML.

Agora que sabemos como funciona o Ciclo de Vida do Spring Web, podemos continuar o projeto da parte anterior. Desta vez, preocupando-nos com a implementação da parte web dele.

Primeiro, vamos construir o Controller e entender o seu funcionamento.

1.3 Spring Web – Controller

O Controller web é uma classe importante em projetos web que tem como função mapear as rotas em métodos, que receberão as requisições HTTP vindas do cliente e encaminharão as informações para a camada de serviço. Cada Controller é responsável por manipular as informações específicas de uma entidade do sistema, como usuários, produtos ou pedidos. Isso torna o código mais organizado e modular. Além disso, o Controller é capaz de retornar as informações em diferentes formatos, como HTML, JSON ou XML, permitindo que a interface web seja exibida de forma adequada em diferentes dispositivos e plataformas. Por exemplo, um Controller de um sistema de e-commerce pode ser responsável por manipular informações de pedidos e retornar em formato JSON para uma aplicação móvel, enquanto outro Controller manipula informações de produtos e retorna em formato HTML para a página web do sistema.

Nesse caso, vamos configurar o `ProdutoController` (dentro de um subpacote chamado `Controller`), com o “Código do `ProdutoController`”:

```
@Controller
@RequestMapping("produto")
public class ProdutoController {

    @Autowired
    private ProdutoRepository repository;

    @Autowired
    private CategoriaRepository categoriaRepository;

    @GetMapping("cadastrar")
    public String abrirFormulario(Produto produto, Model
model) {
        model.addAttribute("categorias",
categoriaRepository.findAll());
        return "produto/form";
    }

    @PostMapping("cadastrar")
    public String processarForm(@Valid Produto produto,
BindingResult result, RedirectAttributes redirectAttributes) {
        if(result.hasErrors()) {
```



```
        return "produto/form";
    }
    redirectAttributes.addFlashAttribute("msg",
"Cadastrado!");
    repository.save(produto);
    return "redirect:listar";
}

@GetMapping("listar")
public String listarProdutos(Model model){
    model.addAttribute("produtos", repository.findAll());
    return "produto/lista";
}

@GetMapping("editar/{id}")
public String editar(@PathVariable("id") int codigo,
Model model){
    model.addAttribute("produto", repository.findById(codigo));
    return "produto/form";
}

@PostMapping("excluir")
public String remover(int codigo, RedirectAttributes
redirectAttributes) {
    redirectAttributes.addFlashAttribute("msg",
"Removido!");
    repository.deleteById(codigo);
    return "redirect:listar";
}
}
```

Código-fonte 1 – Código do ProdutoController
Fonte: Elaborado pelo autor (2019)

As *annotations* principais a serem estudadas são:

- **@Controller:** informa ao Spring que esta classe possui um Controller com mapeamentos (rotas).
- **@RequestMapping:** o parâmetro dentro desta *annotation* avisa que uma sub-rota com o caminho deve ser encaminhada para este Controller.
- **@GetMapping / @PostMapping:** o parâmetro dentro desta *annotation* informa que o *endpoint* (final de rota) e o respectivo verbo (GET e POST) devem ser encaminhados para o método anotado.

- **@PathVariable:** é uma forma de avisar ao Spring MVC que uma determinada parte da URL virá como se fosse uma variável dentro do método.

As demais *annotations* serão apresentadas posteriormente, no momento apropriado. Basicamente, este código implementa um CRUD e a lógica de navegação da interface web, mas repare que não há nada de código HTML (por enquanto), que fica separado em arquivos à parte, como veremos a seguir.

Vale a pena saber que o papel das strings de retorno é desavisar ao Spring qual o arquivo HTML que devemos servir ao usuário assim que o método terminar de ser executado.

Os arquivos HTML devem ser colocados dentro da pasta **resources/templates** (vide Figura “Estrutura de Projeto”) e podem ser utilizadas subpastas para melhor organização das páginas. Por exemplo, se o retorno é “produto/lista”, o Spring vai procurar o arquivo **lista.html** dentro da subpasta produto em resources/templates.

Não é necessário informar a extensão do arquivo, o Spring cuida de buscar o formato apropriado, então, basta informar o nome do arquivo. O que é também chamado de navegação implícita (pois o caminho para o arquivo é informado indiretamente no código).

Por padrão, o Spring utiliza a política de realizar um *Forward* em vez de um *Redirect* para a navegação entre páginas web.

Um *Forward* tem três características principais:

- É processado internamente pelo servidor.
- Quando o processo é finalizado, a URL no browser do usuário não se altera.
- No caso de ocorrer um *reload* da página, o browser enviará novamente a requisição original.

Já o *Redirect* possui as seguintes características:

- É processado em duas etapas. Quando o processamento é realizado, o browser do usuário recebe uma segunda URL e, por isso, a URL no browser muda.

- Um *reload* de página não reprocessará a requisição original, visto que já está em uma página diferente.
- Como consequência do item anterior, variáveis e atributos da requisição original são perdidos em requisições futuras.

Caso precise utilizar um *redirect*, basta acrescentar **redirect:** no início da String de retorno.

Agora, com o Controller criado, podemos focar na criação das páginas e na interface web. Inicialmente, verificamos como os templates funcionam. Depois, acrescentaremos o bootstrap e, assim, poderemos criar as páginas do projeto.

1.4 Spring Web – Template

Os Templates no Spring Web são moldes ou esqueletos com a estrutura básica de um portal, que são usados para padronizar a aparência e o comportamento de uma página web, permitindo que partes comuns possam ser reutilizadas em diferentes páginas. Eles são úteis quando partes da página são comuns em várias páginas, como cabeçalho ou rodapé. Em vez de repetir o mesmo código em cada página, o desenvolvedor pode criar um Template contendo essas partes comuns e incluí-las em cada página usando o recurso de inclusão de Templates do Spring. Isso ajuda a simplificar o processo de desenvolvimento de páginas web, permitindo que o desenvolvedor se concentre em partes específicas de cada página, sem se preocupar com as partes comuns. Além disso, eles podem ser facilmente atualizados ou modificados, sem a necessidade de alterar todas as páginas que utilizam aquele Template.

Para mostrar como funciona um template, vamos configurar o arquivo **base.html**, que ficará dentro do `resources/templates`, conforme o código abaixo:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
  <meta charset="UTF-8">
  <title layout:title-pattern="$LAYOUT_TITLE |
$CONTENT_TITLE">FIAP</title>
```

```
<link
th:href="@{/webjars/bootstrap/css/bootstrap.min.css}"
rel="stylesheet"/>
</head>
<body>

<h1>FIAP Store</h1>
<div class="container">
    <div layout:fragment="conteudo"></div>
</div>
<hr>
<div layout:fragment="rodape"></div>

<script th:src="@{/webjars/jquery/jquery.min.js}"></script>
<script
th:src="@{/webjars/bootstrap/js/bootstrap.min.js}"></script>
</body>
</html>
```

Código-fonte 2 – Código da Página de Template
Fonte: Elaborado pelo autor (2019)

Nesse código HTML, inicialmente, configuramos a taglib do Thymeleaf (que será explicado em breve) e o Bootstrap via WebJars (próxima seção). Repare na taglib de layout, é ela que possibilita a criação de templates.

O template será basicamente preenchido com três informações:

- Título da Página.
- Div de Conteúdo.
- Div de Rodapé.

É possível acrescentar outros campos para serem preenchidos. Normalmente itens como menus, rodapés e estruturas comuns entre páginas costumam constar no template. No código anterior, o texto FIAP Store e o *import* do bootstrap que ficaram no template e as demais informações virão das páginas que se utilizam desse template. Vale destacar os grandes benefícios de utilizar templates: reúso de código e melhor manutenabilidade em caso de necessidade de mudanças no esqueleto do template porque basta alterá-lo, em vez de mudar todas as páginas.

Ao utilizar um template, é preciso colocar a dependência no Maven (Infelizmente, o Spring Initializr não oferece a opção no momento da criação do projeto). Para isso, basta acrescentar a dependência indicada no Código-fonte “Dependência do Layout no pom.xml”:

```
<dependencies>
  ...
  <dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
  </dependency>
  ...
</dependencies>
```

Código-fonte 3 – Dependência do Layout no pom.xml

Fonte: Elaborado pelo autor (2019)

É fundamental que essa dependência seja colocada dentro da tag **<dependencies>**, com as demais dependências do projeto.

Agora, só falta acrescentar a integração com o Bootstrap para podermos montar as páginas desse projeto.

2 SPRING WEB – WEBJARS

O WebJars é uma ferramenta que permite integrar bibliotecas de interface web, como o Bootstrap, em projetos que utilizam o Spring Framework de forma fácil e prática. Com o WebJars, o processo manual de download e referência dos arquivos CSS e JavaScript é eliminado, já que o próprio WebJars realiza essa tarefa automaticamente. Ao adicionar uma dependência WebJars em um projeto Spring, os arquivos do Bootstrap são baixados e configurados automaticamente. Isso significa que não é necessário fazer o download dos arquivos nem referenciá-los manualmente nas tags de script e stylesheet do HTML. Além disso, o WebJars permite manter as bibliotecas de interface web atualizadas com facilidade, bastando atualizar a versão da dependência no arquivo pom.xml ou build.gradle. O WebJars é uma ferramenta muito útil para desenvolvedores que utilizam o Spring Framework em seus projetos web, pois permite uma integração fácil e rápida de bibliotecas de interface, facilitando o processo de desenvolvimento e manutenção do projeto.

Repare que, no código do template, já existe essa configuração feita (a rota utilizada será explicada posteriormente). Falta adicionar as dependências do WebJars no Maven, sendo necessário acrescentar estas três dependências apresentadas no Código-fonte “Dependência do WebJars no pom.xml”:

```
<dependencies>
...
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>4.1.0</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.6.0</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
    <version>0.30</version>
</dependency>
...
```

```
</dependencies>
```

Código-fonte 4 – Dependência do WebJars no pom.xml
Fonte: Elaborado pelo autor (2019)

Em caso de surgir uma nova versão do Bootstrap, basta trocar a versão no pom.xml e reiniciar o sistema, que a atualização já é refletida no sistema.

Agora estamos preparados para utilizar o Thymeleaf no projeto.

2.1 Thymeleaf

O Thymeleaf é uma engine de template utilizada no desenvolvimento de aplicações web com o Spring Framework. Ele permite a criação de templates HTML com marcações específicas que geram conteúdo dinâmico e personalizável. Além disso, o Thymeleaf é facilmente integrado com o Spring Framework, permitindo a transferência eficiente de dados entre o controlador e a camada de visualização. Em resumo, o Thymeleaf é uma ferramenta poderosa e flexível para a criação de interfaces web com o Spring Framework.

Além disso, o Thymeleaf é muito versátil, permitindo que sejam utilizados diversos recursos avançados como internacionalização, manipulação de datas e hora, criação de formulários dinâmicos, entre outros. Isso torna o desenvolvimento de aplicações web com o Spring muito mais fácil e produtivo. Existem várias funcionalidades embutidas nesse módulo do Spring, entre elas, as principais são:

- Cuidar da transferência de parâmetros entre a página e o sistema.
- Possibilitar o uso de lógica básica de tela para permitir a criação de páginas dinâmicas.
- Criação de rotas dinâmicas e de Beans.

Existem as famosas *expressions* que o Thymeleaf utiliza, cada uma para um uso específico:

- `${..}` – Variable Expression.
- `*{..}` – Selection Expression.

- `#{..}` – Message Expression.
- `@{..}` – Link (URL) Expression.
- `~{..}` – Fragment Expression.

Agora vamos criar uma tela contendo um *form* de cadastro para explicar como funcionam essas *expressions*, configurando o arquivo `form.html` dentro da pasta **resources/templates/produto**:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{base}">
<head>
    <meta charset="UTF-8">
    <title>Cadastro de Produtos</title>
</head>
<body>

<div layout:fragment="conteudo">

    <form th:action="@{/produto/cadastrar}"
th:object="${produto}" method="post">
        <div class="alert alert-danger"
th:if="${#fields.hasErrors('*')}">
            <p th:each="err : ${#fields.errors('*')}"
th:text="${err}"></p>
        </div>

        <input th:field="*{codigo}" type="hidden">

        <div class="form-group">
            <label for="nome">Nome</label>
            <input th:field="*{nome}" id="nome" class="form-
control">
        </div>

        <div class="form-group">
            <label for="data">Data Fabricação</label>
            <input th:field="*{dataFabricacao}" id="data"
class="form-control">
        </div>

        <div class="form-group">
            <input th:field="*{novo}" id="novo"
type="checkbox">
            <label for="novo">Novo</label>
        </div>

    </form>

</div>
```



```
<div class="form-group">
  <label for="preco">Preço</label>
  <input th:field="*{preco}" id="preco"
class="form-control">
</div>

  <label for="categoria">Categoria</label>
  <select th:field="*{categoria}" id="categoria"
class="form-control">
    <option value="">Selecione</option>
    <option th:each="c:${categorias}"
                th:value="${c.codigo}"
th:text="${c.nome}"></option>
  </select>

  <button class="btn btn-success">Cadastrar</button>

</form>
</div>

<div layout:fragment="rodape">
</div>

</body>
</html>
```

Código-fonte 5 – Código da página form.html
Fonte: Elaborado pelo autor (2019)

Repare que a *expression* `~{..}` funciona como ligação entre um template e a página que a utiliza, injetando informação. Enquanto a *expression* `@{..}` cuida da geração de rotas dinâmicas, no caso do IntelliJ, ele consegue até verificar sintaxe e se a rota é válida.

Já a *expression* `${..}` faz referência a uma variável que veio do sistema, a qual foi exportada pelo método `model.addAttribute` no Controller. E o *expression* `*{..}` é um seletor que chama uma variável que está contida dentro de outro objeto; ele é útil para evitar chamar várias vezes o mesmo objeto dentro de um formulário extenso.

A *expression* `#{..}` será explicada em breve.

A página em si é um formulário básico que faz o cadastro de um novo produto, com o preenchimento das informações de nome, preço, categoria, data de fabricação e se é novo ou não.

Agora, vamos criar uma página que conterá a listagem de produtos, com botões de editar e remover para cada item da lista. Para isso, configuraremos o arquivo `lista.html` dentro da pasta **resources/templates/produto**.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{base}">
<head>
    <meta charset="UTF-8">
    <title>Lista de Produtos</title>
</head>
<body>

<div layout:fragment="conteudo">
    <div class="alert alert-success" th:if="{msg !=null}"
th:text="{msg}"></div>

    <table class="table">
        <tr>
            <th>Nome</th>
            <th>Preço</th>
            <th>Data Fabricação</th>
            <th>Novo</th>
            <th>Categoria</th>
            <th></th>
        </tr>
        <tr th:each="prod:{produtos}">
            <td th:text="{prod.nome}"></td>
            <td
th:text="{#{numbers.formatDecimal(prod.preco, 1,
2)}}"></td></td>
            <td
th:text="{#{temporals.format(prod.dataFabricacao,
'dd/MM/yyyy') }}"></td>
            <td th:text="{prod.novo?'Sim':'Não'}"></td>
            <td th:text="{prod.categoria?.nome}"></td>
            <td>
                <a
th:href="{@{/produto/editar/}+{prod.codigo}"
                class="btn btn-outline-
primary">Editar</a>

                <button th:onclick="{|produtoId.value =
{prod.codigo}|"
                    type="button" class="btn btn-
outline-danger btn-sm" data-toggle="modal" data-
target="#exampleModal">
                    Excluir
                </button>
            </td>
        </tr>
    </table>
</div>
```

```
        </td>
      </tr>
    </table>

    <div class="modal fade" id="exampleModal" tabindex="-1"
    role="dialog" aria-labelledby="exampleModalLabel" aria-
    hidden="true">
      <div class="modal-dialog" role="document">
        <div class="modal-content">
          <div class="modal-header">
            <h5 class="modal-title"
            id="exampleModalLabel">Deseja excluir o produto?</h5>
            <button type="button" class="close"
            data-dismiss="modal" aria-label="Fechar">
              <span aria-
            hidden="true">&times;</span>
            </button>
          </div>
          <div class="modal-body">
            <form th:action="@{/produto/excluir}"
            method="post">
              <input type="hidden" name="codigo"
              id="produtoId">
              <button type="button" class="btn
            btn-secondary"
              data-
            dismiss="modal">Não</button>
              <button type="submit" class="btn
            btn-danger">Sim</button>
            </form>
          </div>
        </div>
      </div>
    </div>

    <div layout:fragment="rodape">
    </div>

  </body>
</html>
```

Código-fonte 6 – Código da página lista.html
Fonte: Elaborado pelo autor (2019)

Essa página contém uma tabela que listará todos os produtos. Para cada produto, existem um botão de editar e um botão de remover.

Finalmente estamos preparados para testar a interface web do nosso projeto.

2.2 Teste da Interface Web

Basta executar o projeto na sua IDE. E vamos, primeiro, testar o cadastro de um novo produto pela URL <http://localhost:8080/produto/cadastrar>.

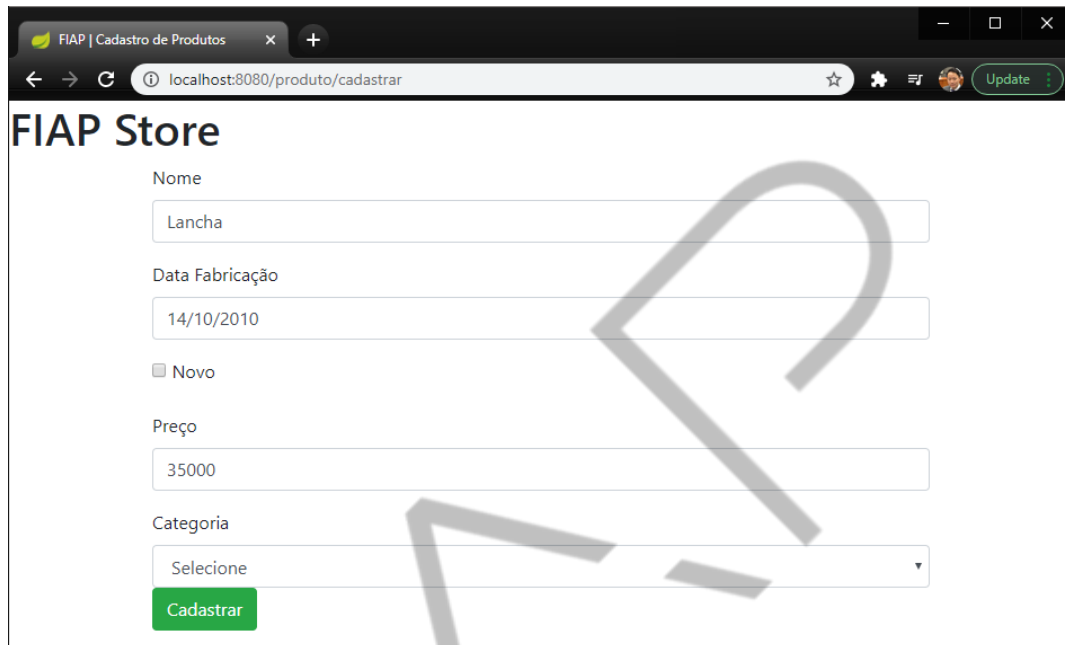
A imagem mostra uma interface web no navegador com o título "FIAP Store". O formulário de cadastro contém os seguintes campos: "Nome" com o valor "Lancha", "Data Fabricação" com o valor "14/10/2010", uma caixa de seleção "Novo" desmarcada, "Preço" com o valor "35000", e "Categoria" com o menu suspenso aberto mostrando "Selecione". Um botão verde "Cadastrar" está na base do formulário. O navegador mostra a URL "localhost:8080/produto/cadastrar" e uma aba "FIAP | Cadastro de Produtos".

Figura 3 – Tela de cadastro de um produto
Fonte: Elaborado pelo autor (2019)

Coloque as informações do produto conforme a Figura “Tela de cadastro de um produto”. Por enquanto, o campo “Categoria” ficará em branco por não termos uma categoria cadastrada. Depois, clique em Cadastrar.

Ao clicar em Cadastrar, é feita uma requisição ao sistema para a rota /produto/cadastrar (usando método POST, conforme está na *action* do HTML do *form*). A rota é mapeada para o método processarForm no ProdutoController, que cuida de inserir os dados do novo produto no banco de dados e redireciona o browser do usuário para a página de listagem de produtos, conforme a Figura “Tela de listagem de produtos”.

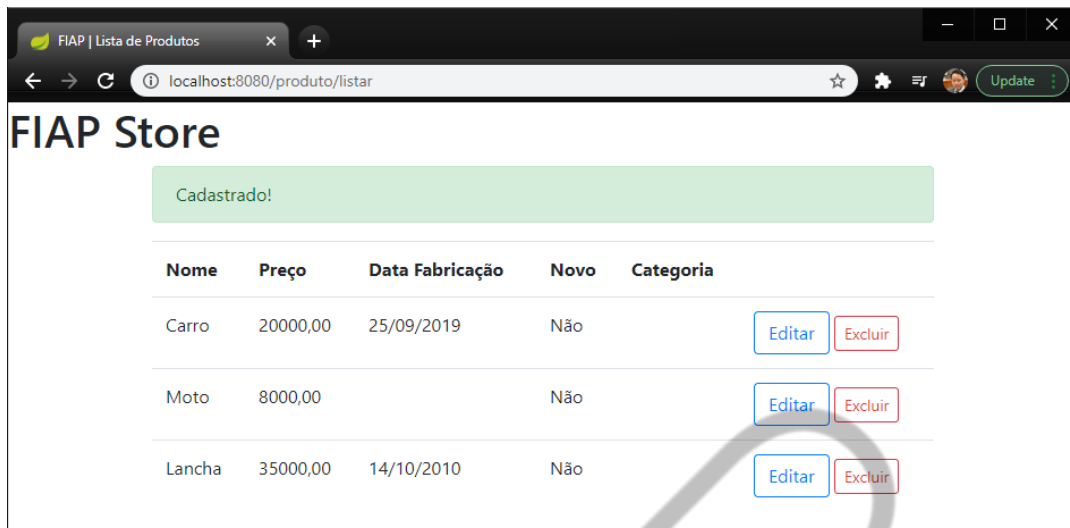


Figura 4 – Tela de listagem de produtos
Fonte: Elaborado pelo autor (2019)

Nessa tela de listagem de produtos, para cada produto listado, existe um botão de ação associado. Repare que para a ação de editar, é feita uma chamada para o mesmo formulário (página) de cadastro, com a diferença de que são repassadas as informações atuais do produto para a edição.

Já na ação de excluir, existe um HTML mais elaborado, fazendo uso de um modal, a famosa tela de confirmação, ilustrada na Figura “Modal de confirmação de exclusão de produto”:

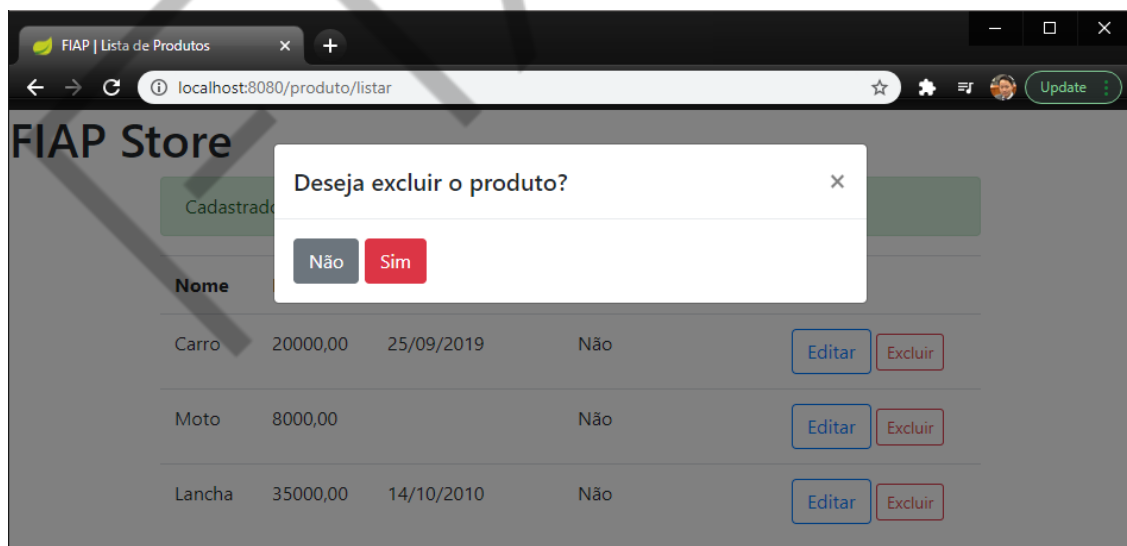


Figura 5 – Modal de confirmação de exclusão de produto
Fonte: Elaborado pelo autor (2019)

Nesse modal, em caso de confirmação da exclusão, é acionada a rota respectiva (com o id de produto) para a sua subsequente remoção no banco de dados. No caso de cancelamento da operação, o modal é removido da tela.

Basicamente, com duas páginas e um modal, é possível materializar todo um CRUD de tela, o Thymeleaf. Nesse aspecto, nos auxilia e facilita a construção de páginas web completas, cuidando também da integração com o Spring.

Faltam dois assuntos para explicar sobre a parte de interface web, teste das validações e mensagens, que serão apresentados agora.

2.3 Teste das Validações

Quando criamos a entidade Produto, além de estabelecermos quais eram os atributos necessários, definimos o mapeamento no banco de dados. Colocamos ainda algumas validações, conforme o Código-fonte “Código da Entidade Produto”:

```
@Entity
@SequenceGenerator(name = "produto", sequenceName =
"SQ_PRODUTO", allocationSize = 1)
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "produto")
    private int codigo;

    @NotBlank(message = "Nome obrigatório!")
    @Size(max = 50)
    private String nome;

    @Min(value=0, message = "Preço não pode ser negativo")
    private double preco;

    private boolean novo;

    @Past
    private LocalDate dataFabricacao;

    // criar getters e setters aqui.
}
```

Código-fonte 7 – Código da Entidade Produto
Fonte: Elaborado pelo autor (2019)

Na entidade, foram definidas as seguintes validações para os campos:

- Nome não pode ser em branco (@NotBlank) e deve ter tamanho máximo de 50 caracteres (@Size).

- Preço não pode ser negativo (@Min).
- Data de fabricação não pode ser no futuro (@Past).

Note que para colocar uma validação, basta utilizar a *annotation* adequada e passar o valor apropriado (quando aplicável). Agora mapeie a validação nos pontos corretos no Controller para essa verificação:

```
public class ProdutoController {  
    ...  
    @PostMapping("cadastrar")  
    public String processarForm(@Valid Produto produto,  
        BindingResult result, RedirectAttributes  
        redirectAttributes) {  
        if(result.hasErrors()) {  
            return "produto/form";  
        }  
        redirectAttributes.addFlashAttribute("msg",  
            "Cadastrado!");  
        repository.save(produto);  
        return "redirect:listar";  
    }  
    ...  
}
```

Código-fonte 8 – Código do ProdutoController
Fonte: Elaborado pelo autor (2019)

Neste código só foi destacada a parte de validação. Inicialmente, é necessário acrescentar a *annotation* @Valid no parâmetro a ser validado (que virá da tela que, no caso, será o formulário de cadastro).

Depois, acrescente o parâmetro do tipo BindingResult, que tem a função de guardar eventuais mensagens de erro na validação. Esse objeto possui um método chamado hasErrors(), responsável por verificar se existem ou não erros de validação no código.

Durante o processamento de uma requisição de cadastro, o método processarForm() verificará se existe algum erro de validação. Caso tenha, é retornada a mesma tela de cadastro, só que com as mensagens de erro (vide próxima seção para mais detalhes).

Caso não existam erros de validação, é guardada uma mensagem de sucesso no campo msg. O novo produto é cadastrado no banco de dados e é realizado um *redirect* para a página de listagem de produtos.

Cabe comentar sobre o uso do `RedirectAttributes`. Esse objeto cuida de armazenar atributos após um *redirect* (lembrando que o *redirect* não aproveita as informações da requisição original). O seu uso fará mais sentido quando estudarmos sobre mensagens, assunto que trataremos a seguir.

2.4 Mensagens e formatação no Spring

Utilizar campos de mensagem no Spring é bem prático e fácil. Existem diversas aplicações que o Spring comporta. As duas principais são as que passam os erros de validação e, depois, as mensagens de sucesso.

O Thymeleaf possui uma *expression* adequada para isso, a `#{..}`, *expression*, responsável pela exibição de mensagens. Nos códigos de cadastro e de lista de produtos, repare que há a especificação de espaços para esses tipos de mensagens (inclusive. com as tags do bootstrap apropriadas).

No caso de erros de validação, existe um campo chamado *message*, no qual você pode especificar qual será o texto da mensagem de erro para deixar mais compreensível ao usuário sobre o problema ocorrido.

Existem duas formas de exibir essa mensagem de erro na tela: a primeira é mostrando todos os erros de validação, como no “Código de exibição de erros de validação no HTML”:

```
<div class="alert alert-danger"
th:if="${#fields.hasErrors('*')}">
    <p th:each="err : ${#fields.errors('*')}"
th:text="${err}"></p>
</div>
```

Código-fonte 9 – Código de exibição de erros de validação no HTML
Fonte: Elaborado pelo autor (2019)

Neste código verifica-se, inicialmente, se há erros (caso contrário, não faz sentido exibir um bloco de alerta vazio). Caso tenha pelo menos um erro de validação, é passado o parâmetro `*`, que lista todos os erros nesta parte da tela. Opção utilizada na tela de cadastro, conforme a Figura “Teste de Validação no Cadastro”.

FIAP | Cadastro de Produtos

localhost:8080/produto/cadastrar

FIAP Store

deve estar no passado

Preço não pode ser negativo

Nome obrigatório!

Nome

Data Fabricação

14/10/2030

☐ Novo

Preço

-35.0

Categoria

Selecione

Cadastrar

Figura 6 – Teste de Validação no Cadastro
Fonte: Elaborado pelo autor (2019)

Caso não seja informado nenhum campo *message* customizado, o próprio Spring fornecerá uma mensagem genérica.

A outra opção é listar somente um erro específico por vez. Essa técnica é útil caso queiramos colocar as mensagens mais próximas ao campo que apresentou erro, como no “Código de exibição de erro específico de validação no HTML”.

```
<p th:if="${#fields.hasErrors('nome')}" class="text text-danger" th:errors="*{nome}"></p>
```

Código-fonte 10 – Código de exibição de erro específico de validação no HTML
Fonte: Elaborado pelo autor (2019)

Este código aponta apenas o erro relacionado ao atributo nome.

Por fim, existem outras maneiras de se formatar a exibição de determinados tipos de dados, como valores monetários e datas. Para isso, o Thymeleaf possui alguns submódulos dentro da *#{..}* *expression*. Mais notadamente, a **#numbers** e **#temporals**, que, no caso da listagem de produtos, cuidam de mostrar o preço do produto até os centavos e, depois, a data de fabricação com a máscara dd/MM/yyyy, conforme o Código-fonte “Código de formatação de conteúdo no HTML”.

```
<td th:text="${#numbers.formatDecimal(prod.preco, 1,  
2)}"></td></td>  
  
<td th:text="${#temporals.format(prod.dataFabricacao,  
'dd/MM/yyyy')}"></td>
```

Código-fonte 11 – Código de formatação de conteúdo no HTML
Fonte: Elaborado pelo autor (2019)

Existem outros pequenos “truques” de estilização da interface que estão nestes dois arquivos HTML (de cadastro e de listagem de produto). Fica como um pequeno desafio descobrir como funcionam.

CONCLUSÃO

O objetivo deste capítulo foi apresentar como é estruturada a parte de *front-end* de um projeto em Spring Boot + Spring MVC.

Para isso, continuamos a desenvolver o projeto do capítulo anterior (responsável por todo o *back-end*), construindo, inicialmente, o template e o layout-padrão. Depois, incluindo o Bootstrap para a estilização de tela. E, por fim, construindo o *Controller* e as respectivas páginas HTML.

Aspectos, como navegação, validação, mensagens e formatação, também foram abordados, cobrindo uma série de funcionalidades e facilidades que o Spring Boot provê para a criação de um projeto.

O Spring (mais especificamente, o Spring Boot) é o *framework web* mais avançado dentro do mundo Java, mas existem outros *frameworks* bem conhecidos e utilizados, só que em outras linguagens de programação. Esse será o foco do próximo capítulo.

No link abaixo, é possível baixar a solução deste capítulo:

Git: <<https://github.com/FIAP/example-springboot-api/tree/mvc-web>>.

Zip: <<https://github.com/FIAP/example-springboot-api/archive/refs/heads/mvc-web.zip>>.

REFERÊNCIA

SPRING. **Introduction to the Spring Framework.** Disponível em: <<https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/overview.html>>. Acesso em: 13 jan. 2021.

EMANIP