



# DESENVOLVIMENTO WEB BACK END

AULA 6



Prof. Rafael Veiga de Moraes



## CONVERSA INICIAL

Um dos principais desafios no desenvolvimento de uma aplicação complexa e de grande porte é criar uma interface gráfica simples e intuitiva para o usuário, de modo a abstrair a complexidade de suas tarefas diárias. Para tal, o desenvolvedor deve implementar, obrigatoriamente, alguns recursos que irão auxiliar o usuário nas suas tarefas, enquanto ele estiver utilizando o sistema.

Dentre esses recursos, podemos elencar a utilização de máscaras nos formulários, para padronizar os campos que possuem uma formatação específica como data, telefone, celular, e-mail, CPF, CNPJ, CEP, entre outros. Além disso, a aplicação também deve indicar para os usuários quando um campo for preenchido com um valor inválido ou uma operação for efetuada pelo usuário, como, por exemplo, salvar, atualizar ou remover um cadastro.

Nesta etapa, veremos como implementar esses recursos e criar o menu principal do sistema, para que o usuário possa acessar as telas da aplicação. Além disso, também criaremos a tela de login, para que somente determinados usuários possam acessar o sistema.

## TEMA 1 – VALIDANDO OS DADOS DO FORMULÁRIO

Embora o formulário de cadastro do cliente esteja realizando a persistência dos dados, há algumas tratativas que devem ser realizadas para assegurar uma maior assertividade no que se refere à integridade dos dados. Por exemplo, se simplesmente abrirmos o formulário de cadastro e clicarmos no botão salvar, será salvo no banco de dados um registro com os dados em branco. Isso não faz qualquer sentido, portanto, devemos aplicar algumas validações no formulário de cadastro, definindo quais campos devem ser obrigatórios.

Outra tratativa importante é assegurar que determinados campos aceitem apenas um determinado tipo ou formato de dado, a fim de evitar erros durante a execução do sistema e manter o dado congruente. Como exemplo, podemos elencar o campo CPF, o qual deve ser preenchido apenas com valores numéricos, pois não existe CPF alfanumérico. Além disso, o CPF tem uma quantidade específica de dígitos e conta com dois dígitos verificadores, dessa forma, não podemos aceitar qualquer valor nesse campo.



A linguagem Java conta com uma biblioteca de validação chamada `javax.validation`, a qual iremos incorporar ao projeto para efetuar a validação dos atributos de um objeto, por meio de suas anotações. Para isso, devemos adicionar a dependência exibida no Quadro 1 ao arquivo `pom.xml` do projeto.

Quadro 1 – Dependência de validação

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

No quadro abaixo, serão abordadas algumas anotações que são amplamente utilizadas para efetuar a validação dos objetos a serem persistidos no banco de dados.

Quadro 2 – Anotações para validação dos atributos

Anotação	Descrição
<b>@NotBlank</b>	Não permite que o atributo seja nulo ou vazio
<b>@NotEmpty</b>	Não permite que o valor do atributo seja vazio
<b>@NotNull</b>	Não permite que o valor do atributo seja nulo
<b>@Digits</b>	Especifica que o atributo é constituído apenas por números
<b>@Email</b>	Especifica que o atributo deverá conter um e-mail válido
<b>@CPF</b>	Especifica que o atributo deverá conter um CPF válido
<b>@Size</b>	Especifica o tamanho mínimo e/ou máximo de caracteres do conteúdo do atributo
<b>@Min</b>	Especifica o valor mínimo de um atributo número
<b>@Max</b>	Especifica o valor máximo de um atributo número

Na classe `Cliente`, serão validados os seguintes atributos:

- **Nome:** campo obrigatório, deve conter no mínimo 3 e no máximo 50 caracteres.
- **CPF:** campo obrigatório, deve conter 11 caracteres e realizar a validação dos dígitos verificadores.
- **Data de nascimento:** campo obrigatório.
- **E-mail:** ao ser preenchido deve ser verificado se é um e-mail válido

Para implementar os requisitos acima, devemos realizar algumas alterações nos atributos da classe `Cliente`, conforme mostra o quadro a seguir.



### Quadro 3 – Atributos da classe Cliente

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
@Column(nullable = false, length = 50)
@NotBlank(message = "informe o nome")
@Size(min = 3, max = 50)
private String nome;
@Column(length = 11)
@CPF(message = "CPF inválido")
private String cpf;
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
@Column(nullable=false, name="data_nascimento", columnDefinition="DATE")
@NotNull(message = "informe a data de nascimento")
private LocalDate dataDeNascimento;
@Enumerated(EnumType.STRING)
private Sexo sexo;
@Column(length = 10)
private String telefone;
@Column(length = 11)
private String celular;
@Column(length = 50)
@email(message = "e-mail inválido")
private String email;
private boolean ativo;
```

Vale ressaltar que, caso não seja especificado o conteúdo da mensagem de validação, será exibida a mensagem padrão da anotação utilizada.

Para que o usuário visualize a mensagem de validação, deve-se realizar mais duas alterações no projeto. A primeira alteração a ser realizada é na classe `ClienteController`, mais especificamente no método `salva`, ao qual deve-se adicionar a anotação `@Valid` e um objeto da classe `BindingResult`. O objeto a ser validado será anotado com a anotação `@Valid`, indicando para o Spring que esse objeto deverá ser validado conforme as anotações empregadas na sua especificação. Para sabermos quais os atributos estão inconsistentes, iremos utilizar o método `hasErrors` do objeto da classe `BindingResult`. Portanto, devemos realizar as seguintes alterações no método `salva` da classe `ClienteController`, conforme mostra o quadro abaixo.

#### Quadro 4 – Método salva da classe ClienteController

```
@RequestMapping(value = "", method=RequestMethod.POST)
public String salva(@Valid @ModelAttribute Cliente cliente, BindingResult
result) {
    if (result.hasErrors()) {
        return "/cliente/formulario";
    }

    if (cliente.getId() == null)
        clienteBO.insere(cliente);
    else
        clienteBO.atualiza(cliente);
    return "redirect:/clientes";
}
```

Antes de realizar a persistência do objeto, deve-se verificar se não há inconsistência nesse objeto e, caso tenha algum atributo inconsistente, devemos informar ao usuário quais os campos do formulário apresentam problemas e, para cada campo, uma mensagem de erro para que o usuário possa corrigi-lo.

Dessa forma, a segunda alteração que deverá ser realizada é no código da página do formulário de cadastro, adicionando os elementos responsáveis por exibir as mensagens de erro referentes a validação do formulário. No código fonte do formulário de cadastro, iremos adicionar uma tag de span para cada campo que possui validação, no caso os campos: nome, CPF, data de nascimento e e-mail.

Cada span irá contar com os seguintes atributos do Thymeleaf:

- **th:if**: por meio desse atributo, iremos identificar se um determinado campo possui alguma inconsistência e, caso tenha qualquer inconsistência, o span será exibido para o usuário, do contrário ficará invisível.
- **th:erros**: esse atributo contém os erros de validação de um determinado campo, dessa forma, conseguimos exibir para o usuário as inconsistências que devem ser corrigidas.

Com base no que foi exposto acima, devemos efetuar alguns ajustes no formulário de cadastro do cliente para exibir as mensagens de erro, caso tenha algum campo inconsistente. Os ajustes podem ser visualizados no Quadro 5.

## Quadro 5 – Formulário de cadastro do cliente com validação

```
<form th:action="@{/clientes}" th:object="${cliente}" method="POST">
  <input id="ativo" type="hidden" th:field="*{id}"/>
  <div class="mb-3 form-check form-switch">
    <label class="form-check-label" for="ativo">Registro ativo</label>
    <input class="form-check-input" role="switch" id="ativo"
      type="checkbox" th:field="*{ativo}"/>
  </div>
  <div class="mb-6">
    <label class="form-label" for="nome">Nome</label>
    <input class="form-control" id="nome" type="text"
      th:field="*{nome}"/>
    <span style="color: red" th:if="${#fields.hasErrors('nome')}}"
      th:errors="*{nome}"></span>
  </div>
  <div class="row">
    <div class="col-4 mb-3">
      <label class="form-label" for="cpf">CPF</label>
      <input class="form-control" id="cpf" type="text"
        th:field="*{cpf}" />
      <span style="color: red" th:if="${#fields.hasErrors('cpf')}}"
        th:errors="*{cpf}"></span>
    </div>
    <div class="col-4 mb-3">
      <label class="form-label" for="dataDeNascimento">
        Data de Nascimento
      </label>
      <input class="form-control" id="dataDeNascimento" type="date"
        th:field="*{dataDeNascimento}" />
      <span th:if="${#fields.hasErrors('dataDeNascimento')}}"
        style="color: red" th:errors="*{dataDeNascimento}"></span>
    </div>
  </div>
```

```

<div class="col-4 mb-3">
  <label class="form-label" for="sexo">Sexo</label>
  <select class="form-select" id="sexo" th:field="*{sexo}">
    <option value="MASCULINO">Masculino</option>
    <option value="FEMININO">Feminino</option>
  </select>
</div>
</div>
<div class="row">
  <div class="col-3 mb-3">
    <label class="form-label" for="telefone">Telefone</label>
    <input class="form-control" id="telefone" type="text"
      th:field="*{telefone}"/>
  </div>
  <div class="col-3 mb-3">
    <label class="form-label" for="celular">Celular</label>
    <input class="form-control" id="celular" type="text"
      th:field="*{celular}"/>
  </div>
  <div class="col-6 mb-3">
    <label class="form-label" for="email">E-mail</label>
    <input class="form-control" id="email" type="text"
      th:field="*{email}"/>
    <span style="color: red" th:if="${#fields.hasErrors('email')}"
      th:errors="*{email}"/></span>
  </div>
</div>
<div class="mb-3">
  <input class="btn btn-primary" type="submit" value="Salvar"/>
</div>
</form>

```

Ao realizar as alterações, inicie a aplicação e acesse a URL <http://localhost:8080/clientes/novo>, informe um e-mail inválido e clique no botão salvar. Note que agora os campos estão sendo validados, conforme mostra a figura abaixo.

Figura 1 – Validação do formulário de cadastro

## Dados do Cliente

☒ Registro ativo

Nome

tamanho deve ser entre 3 e 50  
 informe o nome

CPF

CPF inválido

Data de Nascimento

informe a data de nascimento

Sexo

Telefone

Celular

E-mail

e-mail inválido

## TEMA 2 – FORMATANDO OS DADOS DO FORMULÁRIO

Além da validação dos dados, devemos também assegurar que determinados campos aceitem um determinado tipo ou formato de dado, a fim de evitar erros durante a execução do sistema e manter o dado congruente. Como exemplo, podemos elencar o campo CPF, o qual é formado apenas por números, pois não existe CPF alfanumérico. Ainda, referente ao campo CPF, este possui uma máscara, ou seja, um formato de apresentação, sendo exibido geralmente no padrão XXX.XXX.XXX-XX. Adotar esse padrão no formulário de cadastro irá auxiliar o usuário no preenchimento desse campo, pois em muitos sistemas, o usuário não sabe se o CPF deve ser digitado com pontos e traços ou simplesmente os dígitos.

Há três campos no formulário de cadastro do cliente que podem possuir máscaras: CPF, telefone e celular. Para aplicarmos as devidas máscaras a esses campos, devemos efetuar as alterações no front-end utilizando o JavaScript. Ao invés de utilizar o JavaScript puro, vamos utilizar o JQuery, uma biblioteca de JavaScript pequena, rápida e versátil bastante utilizada no mercado, a fim de aumentarmos a nossa produtividade no ambiente de desenvolvimento. Assim como o Bootstrap, precisamos referenciar o JQuery no código fonte da página Web, adicionando os scripts referentes a essa biblioteca antes do fechamento da tag body, conforme mostra o quadro abaixo.

Quadro 6 – Script JQuery

```
<script      crossorigin="anonymous"      referrerpolicy="no-referrer"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"
integrity="sha512-
894YE6QWD5I59HgZOGReFYm4dnWc1Qt5NtvySaNCOP+u1T9qYdvdihz0PPSiqn/+3e7Jo4EaG
7TubfWGUrMQ=="></script>
<script      crossorigin="anonymous"      referrerpolicy="no-referrer"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery.maskedinput/1.4.1/jquery
.maskedinput.min.js"
integrity="sha512-
d4KkQohk+HswGs6A1d6Gak6Bb9rMWtxj0a0IiY49Q3TeFd5xAzjwXDCBW9RS7m86FQ4RzM2BdHm
dJnnKRYknxw=="></script>
```

Dentre as bibliotecas de script adicionadas temos:

- **jquery.min.js**: biblioteca de comandos do JQuery;
- **jquery.maskedinput.min.js**: biblioteca do JQuery para formatação das máscaras.





Após adicionar as bibliotecas do JQuery ao corpo da página, vamos utilizar o método mask para criar as máscaras referentes aos objetos elencados anteriormente. Para aplicar a máscara a um determinado campo, precisamos da sua identificação, que é definida através do atributo id. No quadro abaixo, temos a declaração do campo CPF.

#### Quadro 7 – Declaração do campo CPF

```
<input class="form-control" id="cpf" type="text" th:field="*{cpf}" />
```

Note que na tag input referente ao campo CPF contém o atributo id, elemento que será utilizado para que possamos aplicar a máscara do CPF a esse campo em específico. Para isso, vamos utilizar o seletor do JQuery para selecionar a caixa de entrada referente ao campo CPF e, na sequência, aplicar a máscara a esse campo por meio do método mask, conforme mostra o quadro abaixo.

#### Quadro 8 – Aplicação da máscara no campo CPF

```
<script type="text/javascript">
$(document).ready(function(){
    $('#cpf').mask('999.999.999-99');
});
</script>
```

Assim que o formulário for carregado no navegador, será executado o script acima, que irá formatar o campo CPF, aplicando a máscara conforme foi especificada no método mask. Utilizando o mesmo conceito, iremos formatar também os campos Telefone e Celular, de tal forma que o script de formatação dos campos ficará da seguinte forma, conforme mostra o Quadro 9.

#### Quadro 9 – Aplicação da máscara nos campos do formulário de cadastro

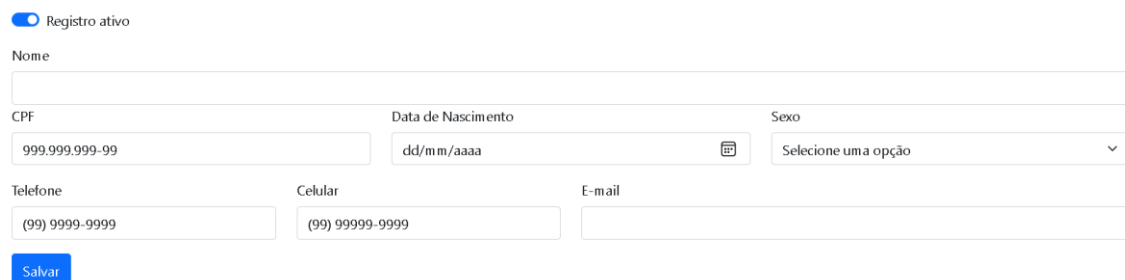
```
<script type="text/javascript">
$(document).ready(function(){
    $('#cpf').mask('999.999.999-99');
    $('#telefone').mask('(99) 9999-9999');
    $('#celular').mask('(99) 99999-9999');
});
</script>
```



Ao aplicar as máscaras aos respectivos campos, o formulário irá formatar os valores conforme mostra a Figura 2.

Figura 2 – Formatação dos campos do formulário

## Dados do Cliente



Registro ativo

Nome

CPF: 999.999.999-99

Data de Nascimento: dd/mm/aaaa

Sexo: Selecione uma opção

Telefone: (99) 9999-9999

Celular: (99) 99999-9999

E-mail:

Salvar

Dessa forma, o próprio sistema irá indicar para o usuário como os campos devem ser preenchidos, evitando dúvidas durante o processo de cadastro garantindo a consistência dos dados. Para finalizar, devemos ajustar o mapeamento objeto-relacional, uma vez que foi alterada a formatação dos campos, mais especificamente dos atributos CPF, telefone e celular. Conforme o Quadro 10, devemos realizar os seguintes ajustes no mapeamento objeto-relacional da classe Cliente.

Quadro 10 – Ajuste no mapeamento objeto-relacional

```
@Column(length = 14)
@CPF(message = "CPF inválido")
private String cpf;

@Column(length = 14)
private String telefone;

@Column(length = 15)
private String celular;
```

Efetuados os ajustes acima, o formulário de cadastro do cliente já está apto a realizar a persistência os dados formatados.

## TEMA 3 – FEEDBACK PARA O USUÁRIO

O foco da aplicação sempre deve estar voltado para o cliente, procurando solucionar as suas necessidades para atendê-lo da melhor maneira. Por isso, devemos sempre estar atentos às dificuldades dos usuários, procurando identificá-las o mais rápido possível e prover uma melhor solução. Além de uma



interface agradável e intuitiva, também é necessário que a aplicação forneça feedbacks para o usuário, ou seja, mensagens interativas informando se o processamento de uma determinada tarefa foi bem-sucedida ou não.

Até o momento, concluímos a implementação do formulário de cadastro do cliente, porém quando editamos ou cadastramos um registro na nossa aplicação, não é exibida nenhuma mensagem para o usuário informando se a operação foi executada com sucesso ou não. Como há poucos registros no banco de dados, é fácil de identificar se o registro foi cadastrado, basta verificar se na tabela de cadastros da tela de gerenciamento do cliente consta o registro na lista. Porém, se houvesse milhares de registros cadastrados, como o usuário iria identificar de forma rápida se o cadastro que ele efetuou foi realmente persistido? Por isso, é muito importante fornecer os feedbacks para os usuários, informando se a operação foi realizada ou não.

Outra situação que deve ser melhorada é com relação as opções de ativar ou inativar um cadastro. Antes de realizar tal ação, é sempre importante perguntar para o usuário se ele deseja realmente efetuar essa operação a fim de evitar erros, já que o usuário pode ter clicado sem querer sobre essa opção ou até mesmo selecionada essa opção para outro registro. Dessa forma, sempre que o usuário for realizar uma operação de grande impacto no sistema, torna-se imprescindível exibir uma mensagem questionando se ele realmente deseja realmente realizar tal operação.

Pensando nisso, devemos realizar algumas alterações no código da nossa aplicação, para que possamos exibir os feedbacks para os usuários. Primeiramente, vamos iniciar as alterações pelos controladores. Sempre que realizarmos as operações de cadastrar, editar, remover, inativar/ativar, devemos exibir o devido feedback para o usuário. Portanto, devemos alterar esses métodos em específico no controlador, adicionando aos seus parâmetros, um objeto da classe `RedirectAttributes`, o qual será utilizado para retornar o feedback referente ao processamento da tarefa efetuada pelo usuário.

Para isso, iremos utilizar o método `addFlashAttribute` da classe `RedirectAttributes`, o qual irá adicionar ao redirecionamento um atributo contendo o feedback para o usuário. Esse método recebe dois parâmetros, sendo o primeiro o nome do atributo e o segundo o seu conteúdo. Por meio desse atributo, o Thymeleaf irá acessar o seu conteúdo e exibir a mensagem na tela, informando o usuário se a tarefa solicitada foi executada com sucesso. Na classe



ClienteController devemos alterar os métodos salva, inativa e ativa, conforme mostra o quadro a seguir.

Quadro 11 – Métodos com feedback para o usuário

```
@RequestMapping(value = "", method=RequestMethod.POST)
public String salva(@Valid @ModelAttribute Cliente cliente,
                   BindingResult result,
                   RedirectAttributes attr) {
    if (result.hasErrors()) {
        System.out.println(result);
        return "/cliente/formulario";
    }

    if (cliente.getId() == null) {
        clienteBO.insere(cliente);
        attr.addFlashAttribute("feedback", "Cliente cadastrado com sucesso");
    }
    else {
        clienteBO.atualiza(cliente);
        attr.addFlashAttribute("feedback", "Cliente atualizado com sucesso");
    }
    return "redirect:/clientes";
}

@RequestMapping(value = "/inativa/{id}", method = RequestMethod.GET)
public String inativa(@PathVariable("id") Long id,
                     RedirectAttributes attr){
    System.out.println(id);
    try {
        Cliente cliente = clienteBO.pesquisaPeloId(id);
        clienteBO.inativa(cliente);
        attr.addFlashAttribute("feedback", "Cliente inativado com sucesso");
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "redirect:/clientes";
}

@RequestMapping(value = "/ativa/{id}", method = RequestMethod.GET)
public String ativa(@PathVariable("id") Long id, RedirectAttributes attr) {
    try {
        Cliente cliente = clienteBO.pesquisaPeloId(id);
        clienteBO.ativa(cliente);
        attr.addFlashAttribute("feedback", "Cliente ativado com sucesso");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Note que para cada método acima foi adicionado um atributo chamado feedback ao redirecionamento para exibir a mensagem de retorno adequada para o usuário. Feito isso, precisamos agora alterar a tela de gerenciamento de cadastro do cliente para que ela exiba a mensagem de retorno. Acima da tabela que exibe todos os clientes cadastrados, iremos adicionar uma div, na qual irá conter um span responsável por exibir o feedback para o usuário. Além disso,



vamos aproveitar para formatar a coluna data de nascimento no formato dia-mês-ano, utilizando o método *format* do objeto *#temporals* do Thymeleaf. O corpo da página de gerenciamento de cadastro do cliente ficará conforme o quadro abaixo.

Quadro 12 – Métodos com feedback para o usuário

```
<body>
  <div class="container">
    <h1>Clientes</h1>
    <hr>
    <div>
      <a class="btn btn-primary" th:href="@{/clientes/novo}">Novo</a>
    </div>
    <hr>
    <div th:if="${!#strings.isEmpty(feedback)}"
      class="alert alert-success" role="alert">
      <span th:text="${feedback}"></span>
    </div>
    <table class="table table-hover">
      <thead>
        <tr>
          <td><b>NOME</b></td>
          <td><b>DATA NASCIMENTO</b></td>
          <td><b>CPF</b></td>
          <td></td>
          <td></td>
        </tr>
      </thead>
      <tbody>
        <tr th:each="cliente : ${clientes}">
          <td th:text="${cliente.nome}"></td>
          <td th:text="${#temporals.format(cliente.dataDeNascimento,
            'dd/MM/yyyy')}">
          </td>
          <td th:text="${cliente.cpf}"></td>
          <td>
            <a th:href="@{/clientes/edita/{id}(id=${cliente.id})}"
              class="btn btn-sm btn-secondary">Editar</a>
          </td>
          <td>
            <a class="btn btn-sm btn-secondary"
              th:href="@{/clientes/ativa/{id}(id=${cliente.id})}"
              th:if="${cliente.ativo == false}">Ativar</a>
            <a class="btn btn-sm btn-secondary"
              th:href="@{/clientes/inativa/{id}(id=${cliente.id})}"
              th:unless="${cliente.ativo == false}">Inativar</a>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</body>
```



Para verificar se o feedback está sendo exibido na tela de gerenciamento, basta cadastrar, editar, inativar ou ativar um registro. Ao alterar um registro por exemplo, será exibida a mensagem de “Cliente atualizado com sucesso”, conforme mostra a imagem abaixo.

Figura 3 – Feedback para o usuário

## Cientes

<a href="#">Novo</a>				
Cliente atualizado com sucesso				
NOME	DATA NASCIMENTO	CPF		
José da Silva	01/01/2000	288.901.820-24	<a href="#">Editar</a>	<a href="#">Inativar</a>

Com os dados devidamente formatados e os feedbacks sendo exibidos para os usuários, finalizamos o desenvolvimento da tela de gerenciamento de cadastro do cliente.

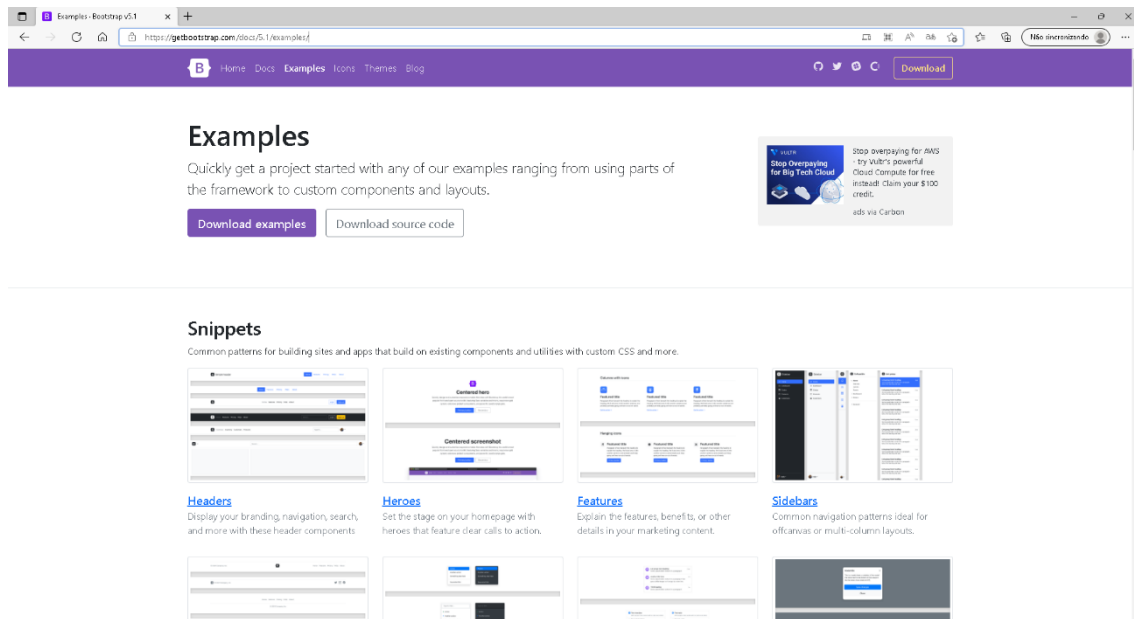
## TEMA 4 – CRIANDO A TELA INICIAL DO SISTEMA

Finalizadas as telas tanto de cadastro quanto de gerenciamento do cliente, precisamos criar a tela inicial do sistema, a fim de fornecer um menu para que o usuário possa navegar pelo sistema, uma vez que ainda, estamos acessando as telas por meio das URLs. Em um projeto comercial, não faz sentido que o usuário tenha que decorar todas as URLs do sistema, para que este possa acessar determinadas telas.

Muitas empresas costumam comprar um *template* ou contratar um designer gráfico para desenvolver a interface gráfica da aplicação, para que o sistema tenha sua própria identidade visual, visto que uma boa interface gráfica requer um tempo considerável de desenvolvimento, além de exigir um bom conhecimento técnico das linguagens HTML, CSS e JavaScript.

Para não perdermos tempo desenvolvendo uma interface gráfica do zero, vamos adotar um dos *templates* fornecidos pelo próprio Bootstrap. Os *templates* estão disponíveis no site do Bootstrap no menu *Examples*, conforme mostra a figura abaixo.

Figura 4 – Templates do Bootstrap



No nosso projeto, iremos adotar o *template Navbar Fixed*, que fornece um menu na parte superior da página, o qual nos permitirá navegar pelas páginas da aplicação. Esse *template* se destaca, por manter o menu fixo na parte superior da página, mesmo que seja utilizada a barra de rolagem, estando sempre visível para o usuário. Para obter o código dos *templates*, basta clicar no botão *Download examples* na parte superior da página. Ao clicar sobre esse botão, serão baixados todos os *templates* disponibilizados pelo Bootstrap. No código fonte do template Navbar Fixed, iremos fazer algumas modificações, remover a caixa de pesquisa no canto superior direito da página, alterar o texto da página inicial e ajustar os links de navegação do menu superior, conforme mostra o quadro a seguir.

## Quadro 13 – Página inicial da aplicação

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8">
    <title>Sistema de Estoque</title>
    <link href="https://getbootstrap.com/docs/5.1/examples/navbar-fixed/"
      rel="canonical" >
    <link rel="stylesheet" crossorigin="anonymous"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/
        bootstrap.min.css"
      integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCE
        XSU1oBoqyl2QvZ6jIW3">
    <style> body { padding-top: 4.5rem; } </style>
  </head>
  <body>
    <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">Sistema de Estoque</a>
        <button class="navbar-toggler" type="button"
          data-bs-toggle="collapse" data-bs-target="#navbarCollapse"
          aria-controls="navbarCollapse" aria-expanded="false"
          aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarCollapse">
          <ul class="navbar-nav me-auto mb-2 mb-md-0">
            <li class="nav-item">
              <a class="nav-link" href="/produtos">Produto</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="/clientes">Cliente</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="/fornecedores">Fornecedor</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="/nota-entrada">Nota de entrada</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="/nota-saida">Nota de saída</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="/estoque">Estoque</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
    <div class="container">
      <div class="bg-light p-5 rounded">
        <h1>Página inicial</h1>
        <p class="lead">Seja bem-vindo ao sistema de estoque</p>
      </div>
    </div>
  </body>
</html>
```



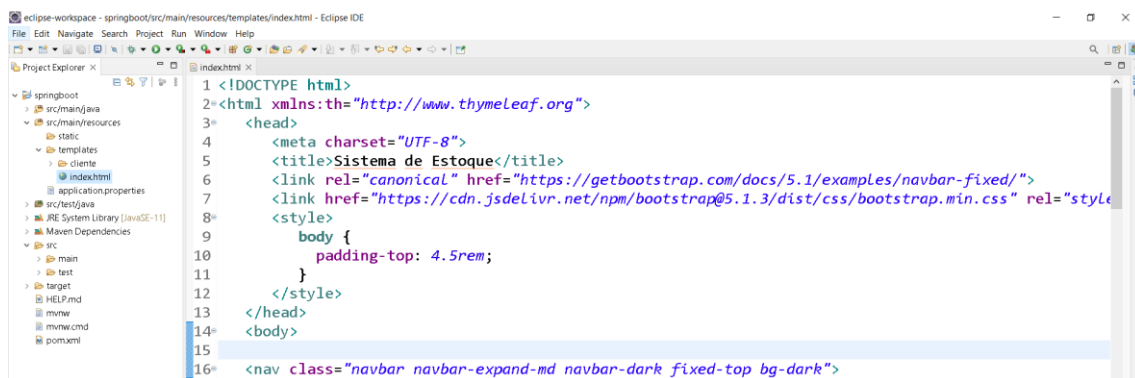
```

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/
bootstrap.bundle.min.js"
integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+ILRH
9sENBO0LRn5q+8nbTov4+1p"
crossorigin="anonymous"></script>
</body>
</html>

```

O código fonte da página inicial da aplicação exibida no quadro acima, deve ser adicionado ao arquivo index.html. Esse arquivo será criado dentro da pasta templates do projeto, conforme mostra a figura abaixo.

Figura 5 – Pasta templates



Para que a tela inicial seja exibida, torna-se necessário a implementação de um controlador. Portanto, vamos criar a classe AplicacaoController no pacote br.com.springboot.controller e adicionar o método index, o qual será responsável por exibir a tela inicial da aplicação, conforme mostra o quadro abaixo.

Quadro 14 – Página inicial da aplicação

```

package br.com.springboot.controller;

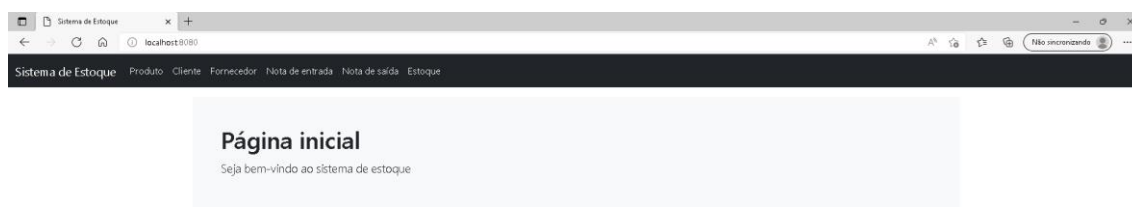
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class AplicacaoController {
    @RequestMapping (value = {"", "/"}, method = RequestMethod.GET)
    public String index() {
        return "index";
    }
}

```

Após criar o controlador, inicie a aplicação e acesse a tela inicial do sistema utilizando a URL `http://localhost:8080`. Ao acessá-la será exibida a tela abaixo, conforme mostra a Figura 6.

Figura 6 – Tela inicial da aplicação



## TEMA 5 – REALIZANDO O LOGIN NA APLICAÇÃO

Toda aplicação comercial conta com uma tela de login, a qual é responsável por efetuar a autenticação do usuário, garantindo que apenas pessoas autorizadas possam acessar a aplicação. Além disso, também precisamos controlar o nível de acesso de cada usuário, definindo quais URLs determinado usuário pode acessar, a fim de garantir que algumas funcionalidades do sistema só possam ser acessadas por um determinado grupo de usuários.

O *Spring* conta com um módulo de segurança chamado *Spring Security*, o qual prove uma poderosa e altamente personalizável estrutura de autenticação e controle de acesso. Além disso, o *Spring Security* também protege a aplicação de diversos ataques como *session fixation*, *clickjacking*, CSRF (Cross-Site Request Forgery), entre outros. Para que possamos utilizar esse recurso na nossa aplicação, temos que adicionar a dependência do *Spring Security* ao projeto, a qual podemos visualizar no quadro abaixo.

Quadro 15 – Dependência do Spring Security

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

Adicionada a dependência do *Spring Security*, vamos criar a classe de configuração de segurança da aplicação, devendo esta ser uma extensão da classe *SecurityWebConfigurerAdapter*, para que possamos utilizar os recursos do *Spring Security*. Além disso, essa classe deverá ser anotada com as



anotações `@Configuration` e `@EnableWebSecurity`, já que se trata de uma classe de configuração de segurança e adicionada ao pacote `br.com.springboot`, conforme mostra o quadro abaixo.

#### Quadro 16 – Classe `SecurityConfiguration`

```
package br.com.springboot;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration
    .EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration
    .WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

Feito isso, vamos criar um método para criptografar as senhas do usuário e sobrescrever os métodos `configure` da classe `SecurityWebConfigurerAdapter` para definir as permissões de acesso e efetuar a autenticação do usuário. Para criptografar a senha, será criado o método `passwordEncoder`, o qual irá retornar uma instância da classe `BCryptPasswordEncoder` do Spring Security, que prove o método `encode` responsável por efetuar a criptografia da senha, conforme mostra o quadro a seguir.

#### Quadro 17 – Método `passwordEncoder`

```
@Bean
public static BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Como ainda não temos uma classe de usuários e tampouco uma tabela de usuários no banco de dados, vamos criar dois usuários com permissões de acesso diferentes em memória. O primeiro poderá acessar o sistema utilizando o usuário `admin`, senha `12345` e permissão de administrador, e o segundo com o usuário `user`, senha `12345` e permissão de usuário. Para criar ambos os usuários em memória, iremos sobrescrever o método `configure` da autenticação, conforme mostra o Quadro 18.



## Quadro 18 – Método configure referente a autenticação do usuário

```
@Override
protected void configure(final AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .inMemoryAuthentication()
            .withUser("admin")
            .password(passwordEncoder().encode("12345"))
            .roles("ADMINISTRADOR")
        .and()
            .withUser("user")
            .password(passwordEncoder().encode("12345"))
            .roles("USUARIO");
}
```

Finalmente, a última configuração a ser definida é com relação as permissões de acesso, para isso iremos sobrescrever o método configure das requisições http. Vamos adotar que somente o usuário admin poderá acessar os menus Nota de entrada, Nota de saída e Estoque. Para garantir que somente esse usuário possa acessar essas telas, devemos configurar que as URLs referentes a esses menus sejam acessadas somente por quem tem a permissão de Administrador. Para isso, vamos utilizar três métodos do objeto http fornecido pelo método configure:

- **authorizeRequests**: utilizado para especificar as URLs que possuem restrição de acesso.
- **antMatchers**: utilizado para especificar qual URL será acessada mediante permissão de acesso.
- **hasRole**: especifica qual grupo de permissão de acesso poderá acessar a URL especificada no método antMachers.

Na sequência, especificamos que as demais URLs da aplicação serão acessadas, somente se o usuário estiver logado no sistema, utilizando os métodos:

- **anyRequest**: especifica a regra de acesso das demais URLs da aplicação
- **authenticated**. assegura que caso o usuário não esteja logado e tente acessar qualquer URL, ele será redirecionado para a tela de login, garantindo que somente pessoas autorizadas possam acessar tal recurso.

Por fim, devemos definir como será realizado o login e o logout do usuário. A configuração do login será especificada por meio dos seguintes métodos:



- **formLogin**: habilita o formulário de login;
- **loginPage**: define a URL da página de login;
- **permitAll**: especifica que qualquer pessoa pode acessar a URL de login.

Já para configurar o logout, serão utilizados os seguintes métodos:

- **logout**: habilitar o recurso de logout na aplicação;
- **logoutRequestMatcher**: especifica a URL do logout;
- **logoutSuccessUrl**: define a URL para qual a aplicação será redirecionada assim que o usuário realizar o logout.

Utilizando os conceitos acima, a implementação do método configure referente as requisições do sistema, ficará conforme mostra o quadro a seguir.

Quadro 19 – Método configure referente as requisições http

```
@Override
protected void configure(final HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/nota-entrada")
                .hasRole("ADMINISTRADOR")
            .antMatchers(HttpMethod.GET, "/nota-saida")
                .hasRole("ADMINISTRADOR")
            .antMatchers(HttpMethod.GET, "/estoque")
                .hasRole("ADMINISTRADOR")
        .anyRequest()
            .authenticated()
        .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
        .and()
            .logout()
                .logoutRequestMatcher(new AntPathRequestMatcher("/logout", "GET"))
                .logoutSuccessUrl("/login");
}
```

Concluídas as devidas configurações referentes à autenticação e permissão de acesso, deve-se implementar a tela de login da aplicação e criar um método dentro da classe AplicacaoController para exibi-la. Portanto, adicione à classe AplicacaoController o método login, conforme mostra o Quadro 20.



## Quadro 20 – Método para exibição do formulário de login

```
@RequestMapping(value = "/login", method = RequestMethod.GET)
public String login() {
    return "login";
}
```

Adicionado o método, deve-se criar o formulário de login. Para isso, devemos criar na pasta templates um arquivo chamado login.html. Esse arquivo irá conter o seguinte código HTML, conforme o quadro a seguir.

## Quadro 21 – Formulário de login

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8">
    <title>Sistema de estoque</title>
    <link rel="stylesheet" crossorigin="anonymous"
          integrity="sha384-1BmE4kWBq78iYhFLdvKuhfTAU6auU8tT94WrHftjD
          brCEXSU1oBoqyL2QvZ6jIW3"
          href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/
          bootstrap.min.css" >
  </head>
  <body>
    <div class="container">
      <form th:action="@{/Login}" method="POST">
```

```

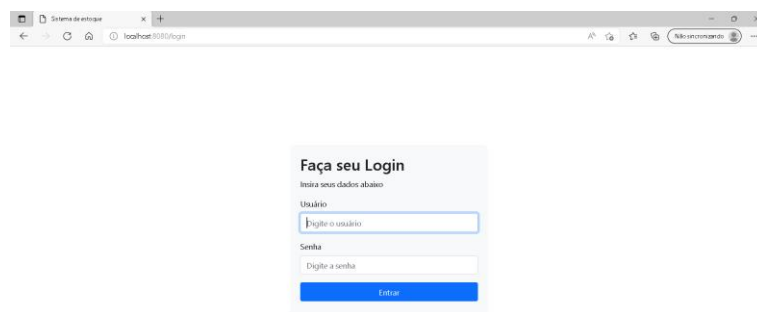
<h2>Faça seu Login</h2>
<p>Insira seus dados abaixo</p>
<div class="mb-3">
  <label class="form-label" for="username">Usuário</label>
  <input type="text" id="username" name="username"
    class="form-control" placeholder="Digite o usuário"
    autofocus />
</div>
<div class="mb-3">
  <label class="form-label" for="password">Senha</label>
  <input type="password" id="password" name="password"
    class="form-control" placeholder="Digite a senha">
</div>
  <div class="mb-3">
    <input class="btn btn-primary" type="submit"
      value="Entrar"/>
  </div>
</form>
</div>
</body>

<style>
  .container {
    margin-top: 200px;
    width: 400px;
    border-radius: 10px;
    background-color: #f8f9fa;
    padding: 20px;
  }
  input {
    width: 100%;
  }
</style>
</html>

```

Implementada a tela de login, basta iniciar a aplicação e acessar a URL <http://localhost:8080/login>. Ao acessar a URL de login, será exibida a tela a seguir, conforme mostra a Figura 7.

Figura 7 – Tela de login da aplicação





Para efetuar o login, informe no campo usuário qualquer um dos usuários que foram criados e memória e a sua respectiva senha. Ao realizar o login, a aplicação será redirecionada para a tela inicial do sistema.

## **FINALIZANDO**

Nesta etapa, abordamos como melhorar a experiência do usuário no manuseio do sistema, de forma a tornar as suas tarefas mais fáceis e intuitivas, sem a necessidade de ter que recorrer ao manual da aplicação ou até mesmo ao suporte técnico.

Para isso, foram implementados diversos recursos como as máscaras, para a formatação dos campos de entrada, efetuada a validação do formulário, para auxiliar o usuário no preenchimento correto dos campos, e a exibição de feedbacks, indicando para o usuário se a tarefa que foi realizada com sucesso.

Com relação à segurança da aplicação, também foi implementada a tela de login e definidas as permissões de acesso de cada usuário da aplicação, garantindo que somente pessoas autorizadas possam utilizar o sistema, restringindo o acesso e assegurando a proteção e privacidade dos dados.