**LINGUAGEM DE PROGRAMAÇÃO**

AULA 4

Prof. Rafael Veiga de Moraes

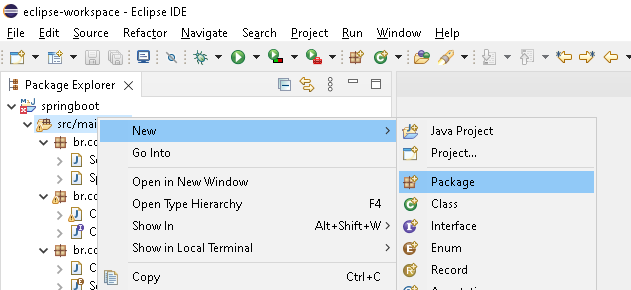
**CONVERSA INICIAL**

Dando sequência ao processo de desenvolvimento da aplicação, nesta aula, criaremos a classe de serviço referente à classe Cliente e realizaremos os testes de integração com o banco de dados, a fim de validar os métodos de acesso a dados. Na sequência, daremos início ao processo de desenvolvimento da interface gráfica da aplicação, abordando alguns conceitos referentes à linguagem HTML e apresentando o Thymeleaf, ferramenta responsável por adicionar dinamicidade às páginas HTML.

**TEMA 1 – CRIANDO A CLASSE DE SERVIÇO**

Dando sequência ao processo de desenvolvimento da aplicação, devemos implementar a classe de negócio referente à classe Cliente, responsável pelo encapsulamento das regras de negócio do objeto, conforme o padrão de projeto BO (*Business Object*). Portanto, todos os serviços devem ser implementados dentro da classe ClienteBO, na qual cada método será responsável por realizar uma determinada tarefa referente ao objeto Cliente. Assim sendo, crie o pacote *br.com.springboot.bo* dentro da pasta *src/main/java*, clicando com o botão direito sobre ela e acessando o menu *New > Package*, conforme a figura a seguir:

Figura 1 – Menu para criação do pacote



Na tela de cadastro de pacote, informe o nome do pacote no campo *Name* e clique no botão *Finish*, conforme a figura a seguir:

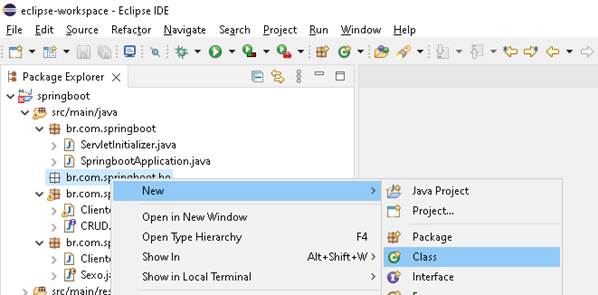
Figura 2 – Tela de criação do pacote

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Adicione a classe ClienteBO ao pacote *br.com.springboot.bo*, clicando com o botão direito sobre ele e acessando o menu *New > Class*, conforme mostra a figura a seguir:

Figura 3 – Menu para adicionar uma classe



Informe, no campo *Name*, o nome da classe e clique no botão *Finish*, conforme mostra a figura a seguir:

Figura 4 – Criação da classe ClienteBO

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Criada a classe, esta deverá implementar a interface CRUD, tornando obrigatória a implementação dos métodos dessa interface. A implementação da classe ClienteBO pode ser visualizada no quadro a seguir:

Quadro 1 – Classe ClienteBO

@Service
public class ClienteBO implements CRUD<Cliente, Long> {
 
       @Autowired
 private ClienteDAO dao;

 @Override
 public Cliente pesquisaPeloId(Long id) {
  return dao.pesquisaPeloId(id);
 }
 @Override
 public List<Cliente> lista() {
  return dao.listaTodos();
 }
 @Override
 public void insere(Cliente cliente) {
  dao.insere(cliente);
 }
 @Override
 public void atualiza(Cliente cliente) {
  dao.atualiza(cliente);
 }
 @Override
 public void remove(Cliente cliente) {
  dao.remove(cliente);
 }
 public void inativa(Cliente cliente) {
  cliente.setAtivo(false);
  dao.atualiza(cliente);
 }
}



Note que aparecem nesta classe duas novas anotações @Service e @Autowired. A primeira anotação @Service serve para indicar que a classe ClienteBO é uma classe de negócio para o Spring, fazendo com que o seu ciclo de vida seja gerenciado pelo próprio framework. A segunda anotação @Autowired é utilizada para realizar a injeção de dependência, uma das principais características do Spring. Por meio da injeção de dependência, o Spring instância o objeto de forma automática, tornando desnecessária a utilização do operador *new*.

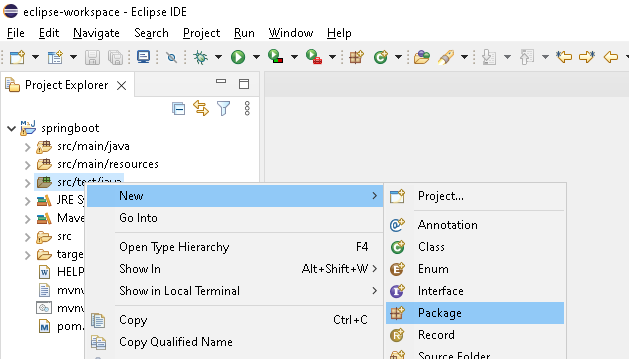
Também foi criado o método inativa, o qual será responsável por inativar o cadastro de um determinado cliente. Em algumas situações, devemos optar por inativar um cadastro em vez de excluí-lo. Ao inativar, tornamos o cadastro invisível dentro do sistema, dando a percepção para o usuário de que o cadastro foi excluído. A diferença entre excluir e inativar é que ao inativar, mantemos o histórico de movimentação desse cadastro dentro do sistema, do contrário, toda movimentação relacionada a esse cadastro seria excluída.

**TEMA 2 – CRIANDO A CLASSE TESTE**

Finalizada a implementação da classe de serviço, devemos validar os seus métodos a fim de nos certificar de que eles estão funcionando corretamente. Para isso, devemos criar uma classe de testes referente à classe ClienteBO. No Spring, as classes de teste devem ser criadas dentro da pasta *src/test/java*. A estrutura de pacotes será a mesma da src/main/java, ou seja, se a classe ClienteBO fica dentro do pacote *br.com.springboot.bo*, será criado um pacote com o mesmo nome na pasta *src/main/java*. A classe de teste referente à classe ClienteBO será o nome da classe a ser testada mais o sufixo Test, portanto, deve-se adicionar a esse pacote uma classe chamada ClienteBOTest.

Assim sendo, crie o pacote *br.com.springboot.bo* dentro da pasta *src/test/java*, clicando com o botão direito sobre ela, e acesse o menu *New > Package*, conforme a figura a seguir:

Figura 5 – Menu para criação do pacote



Na tela de cadastro de pacote, informe o nome do pacote no campo *Name* e clique no botão *Finish*.

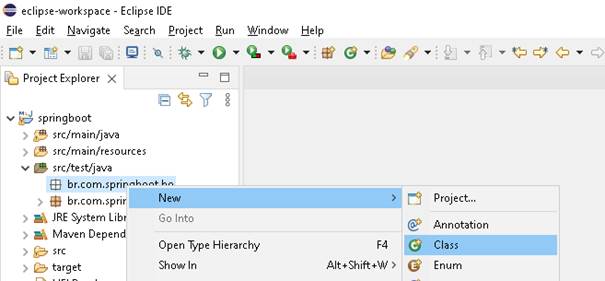
Figura 6 – Criação do pacote br.com.springboot.bo

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Adicione a classe ClienteBOTest ao pacote *br.com.springboot.bo*, clicando com o botão direito sobre ele e acessando o menu *New > Class*, conforme mostra a figura a seguir:

Figura 7 – Menu para criação da classe



Na tela de cadastro da classe, informe o nome do pacote no campo *Name* e clique no botão *Finish*, conforme a figura a seguir:

Figura 8 – Criação da classe ClienteBOTest

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Antes de testar os métodos da classe ClienteBO, é necessário configurar a classe ClienteBOTest. Para isso, serão utilizadas anotações @SpringBootTest, @TestMethodOrder e @ExtendWith, conforme observamos no quadro a seguir:

Quadro 2 – Classe ClientBOTest

package br.com.springboot.bo;

import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@SpringBootTest
@ExtendWith(SpringExtension.class)
@TestMethodOrder(OrderAnnotation.class)
final public class ClienteBOTest {
 
 @Autowired
 private ClienteBO bo;
 
 // Declaração dos métodos a serem testados
}


A anotação @ExtendWith integra o Spring com o JUnit, framework de teste unitário da linguagem Java, responsável por validar se os métodos estão funcionando corretamente. Já a anotação @SpringBootTest inicializa todo o contêiner da aplicação. Dessa forma, podemos realizar a injeção de dependência do objeto de negócio, por meio da anotação @Autowired. Finalmente, a anotação @TestMethodOrder permite que o desenvolvedor defina a ordem de execução dos métodos.

Para cada método da classe ClienteBO, iremos implementar um método de teste com o mesmo nome do método que iremos testar, ou seja, se iremos testar o método insere, deve-se criar um método de teste com o mesmo nome, e assim sucessivamente. Cada método de teste deverá conter duas anotações: @Test e @Order. A anotação @Test indica para o JUnit que o método implementado deverá ser validado pelo framework, já a anotação @Order indica qual será a ordem de execução dos métodos de teste.

Inicialmente, vamos começar os testes pelo método insere. Para testá-lo, devemos instanciar um objeto da classe Cliente, definir um valor aleatório para todos os seus atributos e invocar o método insere do objeto bo, conforme mostra o quadro a seguir:

Quadro 3 – Método insere da classe ClienteBOTest

@Test
@Order(1)
public void insere() {
 Cliente cliente = new Cliente();
 cliente.setNome("José da Silva");
 cliente.setCpf("12345678900");
 cliente.setDataDeNascimento(LocalDate.of(2000, 1, 8));
 cliente.setSexo(Sexo.MASCULINO); cliente.setEmail("email@gmail.com");
 cliente.setTelefone("4133333333");
 cliente.setCelular("41999999999");
 cliente.setAtivo(true);
 bo.insere(cliente);
}


Note que a anotação @Order do método insere recebe o valor “1” por parâmetro, indicando para o JUnit que esse será o primeiro método a ser executado. Quando esse método for executado pelo JUnit, a aplicação tentará inserir na tabela clientes os dados que foram populados no objeto cliente.

O segundo método a ser testado será o método pesquisaPeloId da classe ClienteBO. Para isso, iremos adicionar à classe ClienteBOTest um método com o mesmo nome, a fim de recuperar o registro que foi inserido anteriormente no método insere, caso este tenha sido executado com sucesso. A implementação do método pesquisaPeloId pode ser visualizado no quadro a seguir:

Quadro 4 – Método pesquisaPeloId da classe ClienteBOTest

@Test
@Order(2)
public void pesquisaPeloId() {
 Cliente cliente = bo.pesquisaPeloId(1L);
 System.out.println(cliente);
}


O terceiro método a ser testado será o método atualiza da classe ClienteBO, responsável por atualizar os dados de um determinado cadastro. Portanto, deve-se adicionar à classe ClienteBOTest um método para validar a alteração dos registros. A seguir, encontra-se a implementação do método atualiza da classe ClienteBOTest.

Quadro 5 – Método atualiza da classe ClienteBOTest

@Test
@Order(3)
public void atualiza() {
 Cliente cliente = bo.pesquisaPeloId(1L);
 cliente.setCpf(null);
 cliente.setTelefone(null);
 cliente.setCelular(null);
 cliente.setDataDeNascimento(null);
 bo.atualiza(cliente);
}


O quarto método a ser testado será o método lista da classe ClienteBO, responsável por listar todos os registros cadastrados na tabela clientes. Portanto, deve-se adicionar à classe ClienteBOTest um método para obter todos os registros, retornando uma lista de objetos do tipo Cliente. A seguir, encontra-se a implementação do método lista da classe ClienteBOTest.

Quadro 6 – Método lista da classe ClienteBOTest

@Test
@Order(4)
public void lista() {
 List<Cliente> clientes = bo.listaTodos();
 for (Cliente cliente : clientes) {
  System.out.println(cliente); 
 }
}


O quinto método a ser testado será o método inativa da classe ClienteBO, responsável por inativar um determinado cadastro. Para isso, deve-se adicionar à classe ClienteBOTest um método para inativar o cadastro de um determinado cliente. A seguir, encontra-se a implementação do método inativa da classe ClienteBOTest.

Quadro 7 – Método inativa da classe ClienteBOTest

@Test
@Order(5)
public void inativa() {
 Cliente cliente = bo.pesquisaPeloId(1L);
 bo.inativa(cliente);
}


O sexto e último método a ser testado será o método remove da classe ClienteBO, responsável por inativar um determinado cadastro. Para isso, deve-se adicionar à classe ClienteBOTest um método para remover o cadastro de um determinado cliente. A seguir, encontra-se a implementação do método inativa da classe ClienteBOTest.

Quadro 8 – Método remove da classe ClienteBOTest

@Test
@Order(6)
public void remove() {
 Cliente cliente = bo.pesquisaPeloId(1L);
 bo.remove(cliente);
}


**TEMA 3 – REALIZANDO OS TESTES**

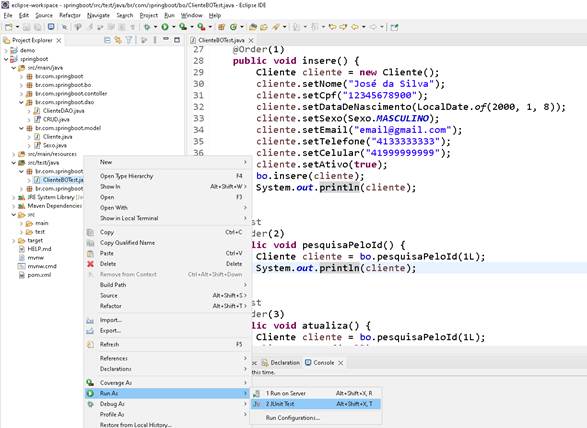
Antes de iniciarmos os testes, vamos adicionar o método toString à classe Cliente. Esse método irá formatar o objeto sempre que tentarmos convertê-lo para String. Dessa forma, podemos utilizar o comando system.out.print para imprimir o conteúdo desse objeto. O método toString pode ser verificado no quadro a seguir:

Quadro 9 – Método toString da classe Cliente

@Override
public String toString() {
 String cliente = "";
 cliente += "CLIENTE\n";
 cliente += "-------------------------\n";
 cliente += "ID.......: " + this.id + "\n";
 cliente += "Nome.....: " + this.nome + "\n";
 cliente += "CPF......: " + this.cpf + "\n";
 cliente += "Data Nasc: " + this.dataDeNascimento + "\n";
 cliente += "Sexo.....: " + this.sexo.getDescricao() + "\n";
 cliente += "Telefone.: " + this.telefone + "\n";
 cliente += "Celular..: " + this.celular + "\n";
 cliente += "Email....: " + this.email + "\n";
 cliente += "Ativo....: " + (this.ativo ? "Sim" : "Não") + "\n";
 return cliente;
}


Para executar os testes, clique com o botão direito na classe ClienteBOTest e selecione o menu *Run As > JUnit Test*, conforme mostra a figura a seguir:

Figura 9 – Execução da classe de teste

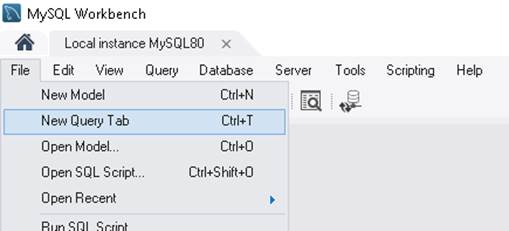


Ao clicar sobre o item *JUnit Test*, o Spring será inicializado e os métodos serão executados de forma sequencial, de acordo com a ordem especificada pela anotação @Order. O primeiro método a ser executado, conforme visto no tema anterior, é o método insere. Esse método irá inserir um registro na tabela clientes com as seguintes informações:

* Nome: José da Silva
* CPF: 12345678900
* Data de nascimento: 08/01/2000
* Sexo: Masculino
* E-mail: email@gmail.com
* Telefone: 4133333333
* Celular: 41999999999
* Ativo: 1 (verdadeiro)

Executado o método insere, deve-se abrir o MySQL Workbench para verificarmos se o registro foi persistido no banco de dados. Faça o login na ferramenta informando o usuário e senha que foi cadastrado durante a instalação dessa ferramenta. Efetuado o login, acesse o menu *File > New Query Tab*ou pressione as teclas de atalho Ctrl + T para abrir uma aba para executar os comandos SQL, conforme mostra a figura a seguir:

Figura 10 – Menu *File > New Query Tab*



Na aba que foi aberta, informe os seguintes comandos SQL conforme o quadro 10. Substitua o termo [base\_dados] pelo nome da base de dados que foi criada para armazenar os dados da aplicação. O primeiro comando será para conectar-se à base de dados. Por sua vez, o segundo comando será para listar todos os registros cadastrados na tabela clientes.

Quadro 10 – Comandos SQL

use [base_dados];

select * from clientes;


Substituído o termo [base\_dados], execute os comandos pressionando as teclas de atalho *Ctrl + F5*.  Ao executá-los, será exibido o resultado apresentado na figura a seguir:

Figura 11 – Cadastro inserido

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente com confiança média

Note que o registro foi inserido com o id igual a 1, pois, no mapeamento objeto-relacional, foi estabelecido que o id seria gerado automaticamente pelo SGBD. Portanto, no método pesquisaPeloId, deve-se passar pelo parâmetro desse id, a fim de que seja retornado os dados desse cadastro. Ao executar o método pesquisaPeloId, será apresentada a seguinte mensagem no console do Eclipse:

Figura 12 – Dados do objeto retornado pelo método pesquisaPeloId



O método atualiza, terceiro método a ser executado, irá atualizar os dados do registro. No caso, esse método irá atualizar os seguintes campos:

* CPF: será atribuído o valor nulo;
* Data de nascimento: será atribuído o valor nulo;
* Telefone: será atribuído o valor nulo;
* Celular: será atribuído o valor nulo.

Após executar esse método, execute novamente os comandos do quadro 10 para verificar se os valores foram alterados.

Figura 13 – Cadastro alterado

Tabela

Descrição gerada automaticamente

O quarto método a ser executado irá retornar todos os registros cadastrados na tabela clientes. Como temos apenas um registro, o método lista irá retornar uma lista com apenas um objeto, o qual foi inserido anteriormente pelo método insere. No console, a aplicação listará todos os objetos que compõem a lista, conforme podemos visualizar na figura a seguir:

Figura 14 – Lista dos objetos cadastrados



O método inativa será o quinto método a ser executado e irá inativar o registro que inserimos anteriormente. Para inativá-lo, será atribuído o valor zero à coluna ativo, tornando esse cadastro invisível dentro do sistema. Após a execução desse método, execute novamente os comandos do quadro 10 para verificar se a coluna *ativo* do registro foi alterada, conforme podemos observar na figura a seguir:

Figura 15 – Registro inativo

Tabela

Descrição gerada automaticamente

Finalmente, ao executar o último método, o registro que foi inserido anteriormente será removido da tabela clientes. Após executar o método remove, execute novamente os comandos do quadro 10 para se certificar de que o registro foi removido com sucesso. Ao executar os comandos SQL, tem-se o seguinte resultado:

Figura 16 – Registro removido

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Note que na figura 16, a consulta SQL não retornou nenhum registro, indicando que o registro foi removido da tabela. Conforme novas classes de teste vão sendo adicionadas ao projeto, torna-se inviável acessar o banco de dados a todo momento, a fim de verificar se os testes foram executados com sucesso.

Figura 17 – Aba JUnit

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Para verificar se o teste se aplica à situação de cada método, o JUnit nos mostra quais testes falharam e quais foram executados com sucesso. Após a execução dos testes, acesse a aba JUnit, conforme mostra a figura 17. Os métodos indicados na cor verde indicam que eles foram executados com sucesso, do contrário, estariam na cor vermelha. Caso um método não tenha sido executado com sucesso, basta clicar sobre ele, e, no quadro *Failure Trace*, verificar o motivo do erro, conforme mostra a figura a seguir:

Figura 18 – Visualizando os erros de um método

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

**TEMA 4 – CRIANDO A INTERFACE GRÁFICA**

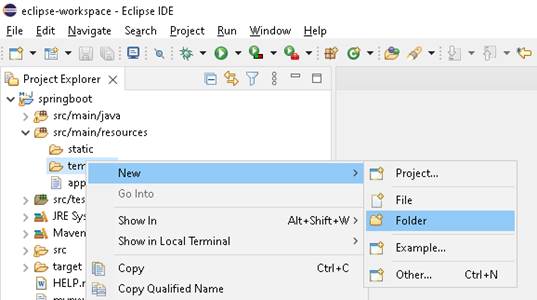
Validados os métodos de negócio referentes à classe Cliente, podemos então iniciar a criação do formulário de cadastro do cliente. Para realizar essa tarefa, iremos utilizar a ferramenta Thymeleaf, um *template engine* Java que permite o desenvolvimento de páginas Web dinâmicas do lado servidor, embutindo código Java em meio ao código HTML. Sendo assim, deve-se adicionar a dependência referente do Thymeleaf ao projeto, apresentada no a seguir:

Quadro 11 – Dependência do Thymeleaf

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


No Spring, as páginas e bibliotecas da camada Web ficam dentro da pasta src/main/resources. Dentro dessa pasta, além do arquivo de configuração do projeto, há outras duas pastas: *static* e *template*. Na pasta *static* são adicionados os recursos que serão utilizados para a construção das páginas Web. Já na pasta *template* iremos encontrar as páginas Web da aplicação. Nessa última, iremos criar a pasta *cliente*, a qual irá conter as páginas Web referentes à classe *cliente*. Para isso, clique com o botão direito sobre a pasta *template* e selecione o menu *New > Folder*, conforme mostra a figura a seguir:

Figura 19 – Menu para criar uma pasta



Na tela de cadastro da pasta, informe o nome da pasta a ser criada no campo *Folder name* e clique no botão *Finish*, conforme mostra a figura 20.

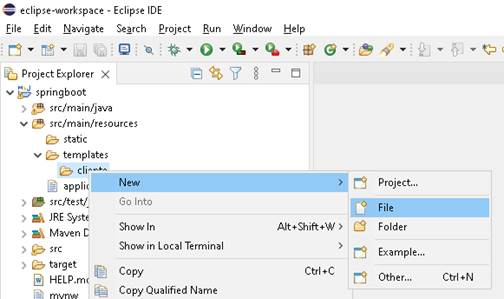
Figura 20 – Tela de criação da pasta

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Na pasta cliente, adicionaremos inicialmente o arquivo formulario.html, no qual iremos implementar a página Web de cadastro do cliente. Para adicionar esse arquivo à pasta, basta clicar com o botão direito sobre a pasta *cliente* e selecionar o menu *New > File*, conforme a figura 21.

Figura 21 – Adicionando um arquivo à pasta



Na tela de criação do arquivo, informe no campo *File name* o nome do arquivo a ser criado e clique no botão *Finish*, conforme mostra a figura a seguir:

Figura 22 – Tela de criação do arquivo

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Adicionado o arquivo formulario.html à pasta cliente, podemos iniciar o desenvolvimento da interface gráfica do formulário de cadastro. Como as páginas Web são desenvolvidas em HTML, vamos abordar os elementos dessa linguagem, para, na sequência, iniciarmos o desenvolvimento dessa tela.

**4.1 LINGUAGEM HTML**

A linguagem HTML (*HyperText Markup Language*) é a principal linguagem de desenvolvimento das páginas Web. Por meio das tags de linguagem, podemos adicionar ao corpo da página elementos como botões, caixa de entrada, tabelas, links, entre outros. As tags são identificadas por meio de palavras entre os símbolos de menor que (<) e maior que (>). A estrutura básica de qualquer página Web em HTML contém, obrigatoriamente, as tags exibidas no quadro a seguir:

Quadro 12 – Estrutura base de uma página HTML

<!DOCTYPE html>
<html>
   <head>
 
   </head>
   <body>
 
   </body>
</html>


Dentre as tags que compõem a estrutura base do documento HTML, temos:

Tabela 1 – Tags de estrutura base do documento HTML

|  |  |
| --- | --- |
| **Tag** | **Descrição** |
| **<!DOCTYPE html>** | Indica que a página foi desenvolvida utilizando HTML5 |
| **<html>** | Representa a raiz do documento HTML |
| **<head>** | Adiciona metadados que serão processados pelo navegador |
| **<body>** | Define o corpo do documento (layout da tela) |

Para o desenvolvimento da interface do formulário de cadastro do cliente, vamos utilizar as seguintes tags:

Tabela 2 – Tags do formulário de cadastro do cliente

I. A tag label adiciona uma etiqueta ao documento V

II. A tag input adiciona um campo de entrada ao documento V

III. A tag select adiciona as opções para um campo de seleção

|  |  |
| --- | --- |
| **Tag** | **Descrição** |
| **<div>** | Define uma divisão ou seção no documento |
| **<label>** | Adiciona uma etiqueta ao documento |
| **<input>** | Adiciona um campo de entrada ao documento |
| **<select>** | Adiciona um campo de seleção ao documento |
| **<option>** | Adiciona as opções para um campo de seleção |
| **<h1> a <h6>** | Adiciona um cabeçalho ao documento |
| **<hr>** | Adiciona uma quebra temática ao documento |
| **<form>** | Adiciona um formulário ao documento |

No quadro 13, é apresentado um exemplo de formulário de cadastro em HTML. Esse formulário conta com os seguintes elementos:

* ***Checkbox***: utilizado para atributos booleanos (verdadeiro ou falso). Esse componente é adicionado à tela por meio da *tag input*, cujo atributo *type* é igual a *checkbox*;
* **Caixa de texto**: utilizado para digitação de caracteres alfanuméricos. Esse componente é adicionado à tela por meio da *tag input*, cujo atributo *type* é igual a *text*.
* ***Combobox***: utilizado para que o usuário possa escolher uma opção dentre um conjunto de opções. Esse componente é adicionado à tela por meio da *tag select*, ao passo que suas opções são adicionadas por meio da *tag option*;
* **Campo data**: utilizado para atributos data. Esse componente é adicionado à tela por meio da *tag input*, cujo atributo *type* é igual a *date*;
* **Botão**: utilizado para submeter o formulário. Esse componente é adicionado à tela por meio da *tag input*, cujo atributo *type* é igual a *submit*.

Quadro 13 – Exemplo de uma página HTML

<!DOCTYPE html>
<html>
<head>
   <meta charset="UTF-8">
   <title>HTML - Campos de entrada</title>
</head>
<body>
   <div>
      <h1>Formulário</h1>
      <hr>
      <div>
         <form>
            <div>
               <label for="checkbox">Checkbox</label>
               <input id="checkbox" type="checkbox"/>
            </div>
            <div>
               <label for="texto">Texto</label>
               <input id="texto" type="text"/>
            </div>
            <div>
               <label for="data">Data</label>
               <input id="data" type="date"/>
            </div>
            <div>
               <label for="combo">Combobox</label>
               <select id="combo">
                  <option value="1">Opção 1</option>
                  <option value="2">Opção 2</option>
               </select>
            </div>
            <div>
               <input type="submit" value="Salvar"/>
            </div>
         </form>
      </div>
      </div>
</body>
</html>


A página HTML do quadro acima, quando executada em um navegador, produzirá o resultado mostrado na figura 23:

Figura 23 – Formulário HTML

Texto

Descrição gerada automaticamente com confiança baixa

**TEMA 5 – CRIANDO O FORMULÁRIO DE CADASTRO DO CLIENTE**

Como abordado anteriormente, a linguagem HTML é apenas uma linguagem de marcação que nos permite criar páginas Web estáticas, portanto, para adicionar dinamicidade à tela, será utilizado o Thymeleaf. Dessa forma, a estrutura base do formulário de cadastro do cliente terá a estrutura a seguir:

Quadro 14 – Estrutura base do formulário do cliente



Antes de iniciar o desenvolvimento do formulário de cadastro, precisamos adicionar duas informações ao cabeçalho do documento HTML. Assim sendo, vamos adicionar ao corpo da *tag head* o título da página por meio da *tag* *title* e definir a codificação dos caracteres do documento, por meio do atributo *charset* da *tag* *meta*. É muito importante definir a codificação dos caracteres, a fim de evitar problemas na visualização do conteúdo da página, especialmente em países cujo idioma contenha caracteres especiais. No quadro a seguir, é exibida a estrutura da *tag head*.

Quadro 15 – Estrutura da *tag head*

<head>
   <title>Sistema de Estoque</title>
   <meta charset="UTF-8">
</head>


Definido o cabeçalho da página, podemos iniciar o desenvolvimento da interface da página. Dentro da *tag body*, vamos adicionar os seguintes elementos:

* **Cabeçalho**: adicionado para dar um título para o formulário de cadastro por meio da *tag* *h1*;
* **Quebra temática**: adicionada para separar o cabeçalho do formulário de cadastro por meio da *tag* *hr*;
* **Formulário**: adicionado para que possamos criar o layout do formulário de cadastro por meio da *tag form*.

Quadro 16 – Estrutura da *tag body*

<body>
   <div>
      <h1>Dados do Cliente</h1>
      <hr>
      <form th:action="@{/clientes}" th:object="${cliente}" method="POST">
      </form>
   </div>
</body>


Note que à *tag form* foram adicionados três atributos com a seguinte finalidade:

* **th:action**: atributo do Thymeleaf que especifica a url que será responsável por efetuar o processamento da requisição quando o formulário for submetido;
* **th:object**: atributo do Thymeleaf que especifica um objeto de comando. O objeto de comando modela os campos de um formulário, fornecendo métodos *getters* e *setters* que serão usados pelo Spring para obter os valores informados pelo usuário no navegador;
* **method**: atributo que especifica o método da requisição http ao submeter o formulário.

Dentro da *tag form*, precisamos adicionar os campos de entrada do formulário para que o usuário possa preencher os dados referentes ao cadastro do cliente. Para isso, devemos criar os campos de acordo com o tipo de cada atributo da classe Cliente e especificar o atributo th:field. Esse atributo será responsável por vincular o dado de entrada ao objeto de comando, especificado por meio do atributo th:objeto. No quadro a seguir, temos a disposição dos campos do formulário de cadastro.

Quadro 17 – Campos do formulário de cadastro

<input id="ativo" type="hidden" th:field="*{id}"/>
<div>
   <label for="ativo">Registro ativo</label>
   <input id="ativo" type="checkbox" th:field="*{ativo}"/>
</div>
<div>
   <label for="nome">Nome</label>
   <input id="nome" type="text" th:field="*{nome}"/>
</div>
<div>
   <label for="cpf">CPF</label>
   <input id="cpf" type="text" th:field="*{cpf}"/>
</div>
<div>
   <label for="dataDeNascimento">Data de Nascimento</label>



   <input id="dataDeNascimento" type="date"
          th:field="*{dataDeNascimento}" />
</div>
<div>
   <label for="sexo">Sexo</label>
   <select id="sexo" th:field="*{sexo}">
      <option value="MASCULINO">Masculino</option>
      <option value="FEMININO">Feminino</option>
   </select>
</div>
<div>
   <label for="telefone">Telefone</label>
   <input id="telefone" type="text" th:field="*{telefone}"/>
</div>
<div>
   <label for="celular">Celular</label>
   <input id="celular" type="text" th:field="*{celular}"/>
</div>
<div>
   <label for="email">E-mail</label>
   <input id="email" type="text" th:field="*{email}"/>
</div>
<div>
   <input type="submit" value="Salvar"/>
</div>


Ao adicionar os campos ao formulário da tela de cadastro do cliente, a interface do formulário de cadastro ficará conforme segue:

Figura 24 – Formulário de cadastro do cliente

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

**FINALIZANDO**

Nesta aula, finalizamos a classe de serviço referente à classe Cliente, implementando as regras de negócio pertinentes a essa entidade do sistema. Os métodos relacionados a essa classe de serviço também foram validados por meio do JUnit, ferramenta de teste unitário da linguagem Java. Com base nisso, mudamos o foco do desenvolvimento para a camada Web, focando no desenvolvimento da interface gráfica dessa entidade. Após abordar alguns conceitos referentes à linguagem HTML e ao Thymeleaf, desenvolvemos o formulário de cadastro do cliente, tela que o usuário utilizará para inserir novos registros e editar cadastros já existentes.

**LINGUAGEM DE PROGRAMAÇÃO**

AULA 5

Prof. Rafael Veiga de Moraes

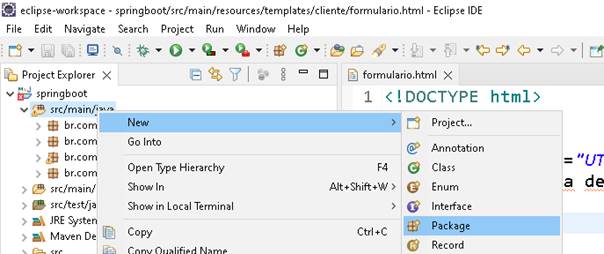
**CONVERSA INICIAL**

Desenvolvida a tela de cadastro do cliente, na sequência precisamos exibir o formulário de cadastro do cliente para o usuário, a fim de que ele possa realizar a persistência de dados. Além disso, temos que disponibilizar uma tela para que o usuário possa gerenciar os cadastros existentes, permitindo que ele possa alterar os dados cadastrais caso seja necessário. Para tornar isso possível, teremos que implementar os controladores para atender às requisições efetuadas pelo usuário, além de prover a tela de gerenciamento de cadastro do cliente.

**TEMA 1 – CRIANDO O CONTROLLER**

Finalizado o desenvolvimento do formulário de cadastro do cliente, precisamos disponibilizá-lo para o usuário por meio de uma url. Para isso, devemos implementar um controller, componente responsável por tratar as requisições http. A fim de manter o projeto organizado, vamos criar um pacote específico para os controllers da aplicação denominado br.com.springboot.controller. Isso posto, clique com o botão direito sobre a pasta src/main/java e acesse o menu New > Package, conforme mostra a figura a seguir:

Figura 1 – Menu para criação do pacote



Na tela de cadastro de pacote, informe no campo Name o nome do pacote e clique no botão Finish, conforme mostra a figura a seguir:

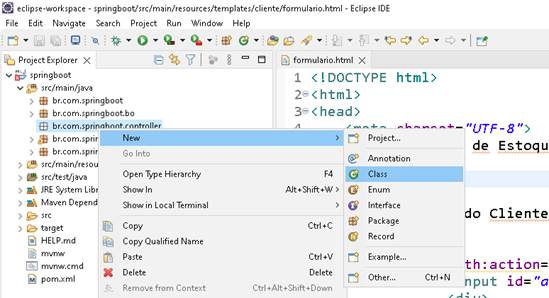
Figura 2 – Tela de criação do pacote

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Adicionado o pacote referente aos controllers da aplicação, vamos adicionar a classe ClienteController a esse pacote. Para tal, clique com o botão direito sobre o pacote e acesse o menu New > Class, conforme mostra a figura a seguir:

Figura 3 – Menu para criação da classe



Na tela de cadastro da classe, informe o nome da classe no campo Name e clique no botão Finish, conforme mostra a figura a seguir:

Figura 4 – Tela de criação da classe

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Criada a classe ClienteController, vamos anotá-la utilizando duas anotações do Spring: @Controller e @RequestMapping, conforme mostra o quadro a seguir:

Quadro 1 – Configuração da classe ClienteController

Caixa de Texto: package br.com.springboot.controller;

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

@Controller
@RequestMapping("/clientes")
public class ClienteController {

}



A anotação @Controller fará com que o Spring reconheça essa classe como um controller e passe a gerenciar o seu ciclo de vida. Já a anotação @RequestMapping irá configurar a url path inicial das requisições para esse controller. Dessa forma, ao tentar acessar o endereço <http://localhost:8080/clientes> por meio de um navegador, essa requisição será encaminhada para que a classe ClienteController realize o seu processamento.

Como queremos exibir o formulário de cadastro do cliente para o usuário, devemos criar um método dentro da classe ClienteController para realizar essa tarefa. Esse formulário será disponibilizado por meio da url <http://localhost:8080/clientes/novo> por meio do método GET. Portanto, vamos adicionar à classe ClienteController o método novo, conforme mostra o quadro a seguir:

Quadro 2 – Método novo da classe ClienteController

Caixa de Texto: package br.com.springboot.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import br.com.springboot.model.Cliente;

@Controller
@RequestMapping("/clientes")
public class ClienteController {

 @RequestMapping(value = "/novo", method = RequestMethod.GET)
 public ModelAndView novo(ModelMap model) {
  model.addAttribute("cliente", new Cliente());
  return new ModelAndView("/cliente/formulario", model);
 }
}

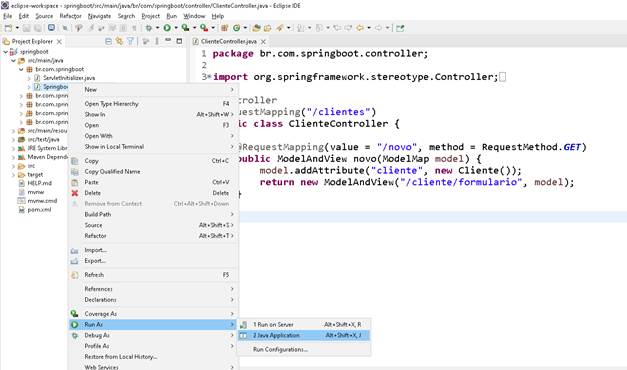


O método novo irá retornar um objeto do tipo ModelAndView, ou seja, um objeto com a página Web (view) e os dados que serão populados na tela (model). A página Web que será exibida no navegador é o formulário de cadastro do cliente, portanto, no primeiro parâmetro do objeto ModelAndView, deve-se informar o caminho do arquivo que implementar essa tela com base na pasta templates, sem a necessidade de informar a extensão do arquivo. O segundo parâmetro a ser fornecido consiste nos dados que serão populados na tela por meio de um objeto do tipo ModelMap.

Para adicionar dados a esse objeto, deve-se utilizar o método addAttribute, no qual o primeiro parâmetro é o nome do atributo que será acessado pelo Thymeleaf e, no segundo parâmetro, será fornecido o conteúdo desse atributo, que pode ser uma variável ou objeto. No caso, vamos adicionar ao objeto ModelMap o atributo cliente, que irá conter os dados de uma nova instância da classe Cliente. Dessa forma, iremos iniciar o formulário com o campo “Registro ativo” selecionado e os demais campos em branco.

Inicie a aplicação clicando com o botão direito sobre a classe SpringApplication e acesse o menu Run As > Java Application, conforme mostra a figura a seguir:

Figura 5 – Menu para iniciar a aplicação



Iniciada a aplicação, abra o navegador e acesse a url <http://localhost:8080/clientes/novo> para que seja carregado o formulário de cadastro do cliente, conforme mostra a figura a seguir:

Figura 6 – Formulário de cadastro do cliente

Interface gráfica do usuário, Tabela

Descrição gerada automaticamente

**TEMA 2 – REALIZANDO O PRIMEIRO CADASTRO VIA APLICAÇÃO**

Exibido o formulário de cadastro do cliente, precisamos agora implementar a ação que será executada quando o usuário clicar no botão salvar. Para isso, na tag form desse formulário deve-se informar, obrigatoriamente, dois atributos para que possamos submeter esse formulário para o servidor da aplicação. O primeiro atributo é a ação que será executada, identificada por meio do atributo *th:action*. Por sua vez, o segundo atributo é o método da ação que será executada, identificado por meio do atributo method. No quadro a seguir, temos a configuração que foi adicionada anteriormente à tag form do formulário de cadastro do cliente.

Quadro 3 – Configuração da tag form

Caixa de Texto: <form th:action="@{/clientes}" th:object="${cliente}" method="POST">

</form>


Ao submeter o formulário de cadastro do cliente, ou seja, quando o usuário clicar no botão salvar, será enviada uma requisição HTTP do tipo POST para a url <http://localhost:8080/clientes>, conforme foram parametrizados os atributos *th:action* e *method*. Portanto, deve-se adicionar mais um método dentro da classe ClienteController para que possamos realizar a persistência de dados assim que o usuário submeter o formulário de cadastro à aplicação. Dentro da classe ClienteController, vamos adicionar o método salva para realizar essa tarefa, conforme mostra o quadro a seguir:

Quadro 4 – Método salva da classe ClienteController

Caixa de Texto: @RequestMapping(value = "", method=RequestMethod.POST)
public String salva(@ModelAttribute("cliente") Cliente cliente) {

}


Novamente, faremos o uso da anotação @RequestMapping para definir a url path e o método da requisição HTTP, conforme foram parametrizados os atributos *th:action* e *method* na tag form do formulário de cadastro do cliente. Além dessa anotação, vamos utilizar também a anotação @ModelAttribute do Spring, a qual será responsável em converter o objeto de comando do Thymeleaf, atributo *th:object* da tag form, em um objeto Java.

Dessa forma, o método salva irá prover ao desenvolvedor, por parâmetro, um objeto do tipo Cliente já populado com os valores digitados pelo usuário no formulário de cadastro do cliente.

Como o método salva será responsável tanto por inserir um novo registro quanto por atualizar um registro já existente, devemos implementar essa regra de negócio baseado no atributo id do objeto cliente. Caso o id seja nulo, será invocado o método insere da classe ClienteBO, do contrário, será invocado o método atualiza dessa mesma classe. Ao realizar a persistência dos dados, vamos redirecionar a aplicação, por ora, novamente para a tela de cadastro do cliente. Portanto, conforme mostra quadro a seguir, teremos a seguinte implementação para a classe ClienteController:

Quadro 5 – Classe ClienteController

Caixa de Texto: package br.com.springboot.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import br.com.springboot.bo.ClienteBO;
import br.com.springboot.model.Cliente;

@Controller
@RequestMapping("/clientes")
public class ClienteController {
 
 @Autowired
 private ClienteBO clienteBO;
 
 @RequestMapping(value = "/novo", method = RequestMethod.GET)
 public ModelAndView novo(ModelMap model) {
  model.addAttribute("cliente", new Cliente());
  return new ModelAndView("/cliente/formulario", model);
 }
 
 @RequestMapping(value = "", method=RequestMethod.POST)
 public String salva(@ModelAttribute("cliente") Cliente cliente) {
  if (cliente.getId() == null)
   clienteBO.insere(cliente);
  else 
   clienteBO.atualiza(cliente);
  return "redirect:/clientes/novo";
 }
}


Inicie novamente a aplicação e, por meio do navegador, acesse a url <http://localhost:8080/clientes/novo>, preencha todos os campos da tela e clique no botão salvar. Ao salvar, caso não tenha ocorrido qualquer erro, acesse o MySQL e verifique se o registro foi inserido com sucesso. Na figura 7 é exibido o formulário de cadastro do cliente preenchido e, na figura 8, os dados que foram persistidos no banco de dados ao acionar o botão salvar do formulário.

Figura 7 – Formulário de cadastro do cliente preenchido

Tabela

Descrição gerada automaticamente com confiança baixa

Figura 8 – Registro persistido

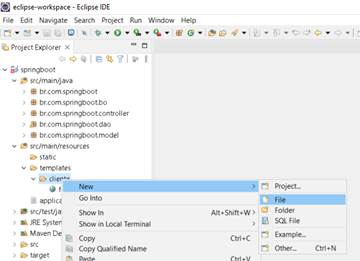
Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

**TEMA 3 – CRIANDO A TELA DE GERENCIAMENTO DE CADASTRO DO CLIENTE**

Após criar a tela de cadastro do cliente, torna-se necessário o desenvolvimento da tela de gerenciamento dos cadastros para que o usuário possa visualizar todos os clientes cadastrados no sistema, além de ter acesso às ações de cadastrar, editar e ativar/inativar um cliente. Para isso, adicione o arquivo lista.html dentro da pasta cliente, clique com o botão direito sobre ela e acesse o menu New > File, conforme mostra a imagem a seguir:

Figura 9 – Adicionar um novo arquivo



Após clicar sobre o item do menu citado anteriormente, será aberta a tela para criação do arquivo. Informe no campo File name o nome do arquivo e clique no botão Finish, conforme a figura a seguir:

Figura 10 – Tela de criação do arquivo

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Além das tags HTML abordadas anteriormente, visando ao desenvolvimento da tela de cadastro do cliente, serão utilizadas mais algumas tags para o desenvolvimento da tela de gerenciamento de cadastro, conforme mostra a tabela a seguir:

Tabela 1 – Tags para o desenvolvimento da tela de gerenciamento

|  |  |
| --- | --- |
| **Tag** | **Descrição** |
| **<a>** | Adiciona um link ao documento |
| **<table>** | Adiciona uma tabela ao documento |
| **<thead>** | Define um cabeçalho para a tabela |
| **<tbody>** | Define o corpo da tabela |
| **<tr>** | Adiciona uma linha à tabela |
| **<td>** | Adiciona uma célula à linha da tabela |
| **<b>** | Deixa o texto em negrito |

No quadro a seguir, temos a estrutura base da tela de gerenciamento de cadastro do cliente.

Quadro 6 – Estrutura base da tela de gerenciamento de cadastro do cliente



Note que foram adicionados à estrutura base da tela de gerenciamento de cadastro de cliente um link e uma tabela. O link permitirá que o usuário possa, por meio dessa tela, acessar a tela de cadastro do cliente e efetuar um novo cadastro. Para isso, deve-se informar no atributo th:href do link, a url que irá redirecionar a aplicação para a tela de cadastro do cliente. A tabela deverá exibir todos os clientes cadastrados no sistema, exibindo suas principais informações e, para cada registro listado na tabela, disponibilizar opções para que o usuário possa editar ou ativar/inativar um determinado registro. Portanto, deve-se adicionar a tag table mais duas tags: thead e tbody.

Na *tag thead* será definido o cabeçalho da tabela, ou seja, nela serão especificadas as colunas que irão compor essa tabela. Para isso, deve-se acionar uma linha ao cabeçalho da tabela por meio da tag tr e especificar suas colunas por meio da tag td. A tabela da tela de gerenciamento de clientes será composta pelas seguintes colunas.

* **Nome**: coluna que irá exibir o nome do cliente.
* **Data de nascimento**: coluna que irá exibir a data de nascimento do cliente.
* **CPF**: coluna que irá exibir o CPF do cliente.
* **Editar**: coluna que irá disponibilizar um link para que o usuário possa editar o cadastro.
* **Ativar/Inativar**: coluna que irá disponibilizar um link para que o usuário possa ativar/inativar o cadastro.

Já na tag tbody serão adicionadas à tabela as informações referentes a cada cliente cadastrado no sistema. Para isso, iremos acessar o objeto que contém a lista de clientes cadastrados no sistema, utilizando para isso o atributo th:each do Thymeleaf, a fim de acessar cada elemento da lista por meio de um laço de repetição. A cada laço, será adicionada uma linha à tabela e o conteúdo de cada objeto populado à coluna correspondente, por meio do atributo th:text da tag td. As colunas responsáveis pelas operações de editar e ativar/inativar irão conter os links referentes às operações pertinentes. A estrutura da tag table da tela de gerenciamento de cadastro do cliente é exibida no quadro a seguir:

Quadro 7 – Estrutura da tag table

Caixa de Texto: <table>
  <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>


Caixa de Texto:     <tr th:each="cliente : ${clientes}">
      <td th:text="${cliente.nome}"></td>
      <td th:text="${cliente.dataDeNascimento}"></td>
      <td th:text="${cliente.cpf}"></td>
      <td>
        <a th:href="@{/clientes/edita/{id}(id=${cliente.id})}">Editar</a>
      </td>
      <td>
        <a th:if="${cliente.ativo == false}" th:href="@{/clientes/ativa/{id}(id=${cliente.id})}">Ativar</a>
        <a th:unless="${cliente.ativo == false}" th:href="@{/clientes/inativa/{id}(id=${cliente.id})}">Inativar</a>
      </td>
    </tr>
  </tbody>
</table>


Repare que nos links foram adicionados os atributos th:if e th:unless. Dessa forma, conseguimos habilitar a opção de ativar para o registro cujo atributo ativo seja falso, e caso o atributo ativo seja verdadeiro, habilitar a opção de inativar o registro.

**TEMA 4 – IMPLEMENTANDO AS FUNCIONALIDADES DA TELA DE GERENCIAMENTO**

Criada a tela de gerenciamento de cadastro de clientes, devemos implementar as funcionalidades dessa tela, adicionando à classe ClienteController os métodos responsáveis pelas tarefas de editar e ativar/inativar um cadastro, além do método que irá exibir a tela de gerenciamento para o usuário.

Primeiramente, vamos implementar o método lista, o qual será responsável por redirecionar a aplicação para a tela de gerenciamento de cadastro de clientes, que será acessada por meio de uma requisição do tipo GET utilizando a url <http://localhost:8080/clientes>. O método lista deverá retornar um objeto do tipo ModelAndView contendo a página de gerenciamento e a lista de clientes cadastrados no sistema, conforme o quadro a seguir:

Quadro 8 – Método lista da classe ClienteController

Caixa de Texto: @RequestMapping(value = "", method=RequestMethod.GET)
public ModelAndView lista(ModelMap model) {
   model.addAttribute("clientes", clienteBO.listaTodos());
   return new ModelAndView("/cliente/lista", model);  
}


Na sequência, iremos adicionar o método edita à classe ClienteController, o qual será responsável por exibir a tela de cadastro do cliente preenchida com os dados do registro que será editado pelo usuário. A edição do registro será feita por meio de uma requisição do tipo GET utilizando a url <http://localhost:8080/clientes/edita/{id}>, na qual o termo {id} será substituído pelo id do cliente que será editado. Para que tenhamos acesso ao id do cliente presente na url da requisição, devemos utilizar a anotação PathVariable do Spring, a qual irá extrair esse dado da url e atribuir a uma variável.

Em posse do id, podemos obter os dados do cliente por meio do método pesquisaPeloId do objeto da classe ClienteBO injetado na classe ClienteController. Dessa forma, conseguimos retornar um objeto do tipo ModelAndView contendo o formulário de cadastro do cliente e os dados que serão utilizados pelo Thymeleaf para renderizar a tela. O código referente ao método edita pode ser visto no quadro a seguir:

Quadro 9 – Método edita da classe ClienteController

Caixa de Texto: @RequestMapping(value = "/edita/{id}", method = RequestMethod.GET)
public ModelAndView edita(@PathVariable("id") Long id, ModelMap model) {
   try {
      model.addAttribute("cliente", clienteBO.pesquisaPeloId(id));
   } catch (Exception e) {
      e.printStackTrace();
   }
   return new ModelAndView("/cliente/formulario", model);
}


Outro recurso dessa tela que devemos implementar é a funcionalidade de inativar um cadastro. Portanto, deve-se implementar dentro da classe ClienteController o método inativa, que será responsável por realizar tal tarefa. Esse método irá funcionar de forma semelhante ao método edita, especificando na url a operação que será realizada e o id do cadastro, de tal modo que a url terá o seguinte padrão <http://localhost:8080/clientes/inativa/{id}>. No quadro a seguir temos a implementação do método inativa:

Quadro 10 – Método inativa da classe ClienteController

Caixa de Texto: @RequestMapping(value = "/inativa/{id}", method = RequestMethod.GET)
public String inativa(@PathVariable("id") Long id) {
   try {
      Cliente cliente = clienteBO.pesquisaPeloId(id); 
      clienteBO.inativa(cliente);
   } catch (Exception e) {
      e.printStackTrace();
   }
   return "redirect:/clientes";
}

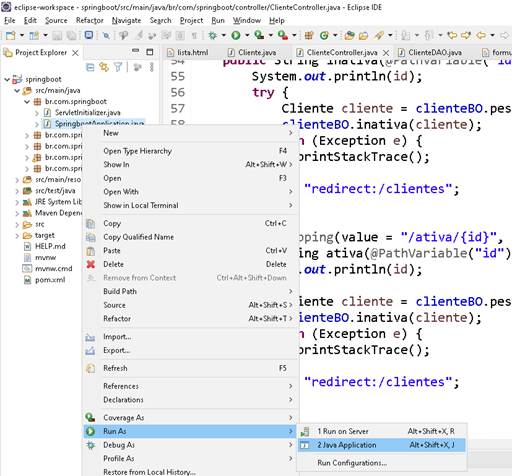

Repare que o retorno desse método não é um objeto do tipo ModelAndView, e sim uma String, indicando que, após inativar o registro, a aplicação será redirecionada para a tela de gerenciamento de cadastro do cliente. Vamos adotar essa mesma abordagem para o método salva, dessa forma, sempre que o usuário cadastrar ou alterar um cadastro, a aplicação será redirecionada para a tela de gerenciamento.

Quadro 11 – Método salva da classe ClienteController

Caixa de Texto: @RequestMapping(value = "", method=RequestMethod.POST)
public String salva(@ModelAttribute("cliente") Cliente cliente) {
   if (cliente.getId() == null)
      clienteBO.insere(cliente);
   else 
      clienteBO.atualiza(cliente);
   return "redirect:/clientes";
}


Ao finalizar a implementação dos métodos citados anteriormente, execute a aplicação clicando com o botão direito sobre a classe SpringbootApplication, localizada dentro do pacote br.com.springboot, e acesse o menu Run As > Java Application, conforme a figura a seguir:

Figura 11 – Iniciando a aplicação



Assim que a aplicação for iniciada, abra o navegador e acesse a url <http://localhost:8080/clientes> para visualizar a tela de gerenciamento de cadastro do cliente, conforme mostra a figura a seguir:

Figura 12 – Tela de gerenciamento de cadastro do cliente

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Na tela de gerenciamento de cadastro do cliente, navegue pelos links e verifique se a aplicação está efetuando as requisições solicitadas corretamente. Inative um cadastro utilizando o link Inativar e, na sequência, tente ativar o mesmo cadastro por meio do link Ativar. Note que a operação solicitada não foi efetuada, pois faltou implementar esse recurso.

Tomando como base a opção de inativar um cadastro, implemente a opção para ativar o cadastro. Lembre-se de que você terá que adicionar um método na classe ClienteController para atender a essa requisição e outro método na classe ClienteBO para realizar essa tarefa.

**TEMA 5 – UTILIZANDO O BOOTSTRAP**

Desenvolvido por um designer e desenvolvedor do Twitter em meados de 2010, o Bootstrap tornou-se um dos frameworks de front-end e projetos de código aberto mais populares do mercado. Utilizando suas bibliotecas, podemos desenvolver, de forma fácil e rápida, aplicações responsivas com um visual elegante. Para isso, devemos adicionar ao código HTML dois arquivos.

I. O Bootstrap é um framework de estilo utilizado no front-end V

II. A classe container do Bootstrap é utilizada para estilizar o menu da aplicação

III. Ao aplicar a classe table-hover do Bootstrap em uma tabela, a linha da tabela na qual o cursor está posicionado ficará destacada

* **Bootstrap.min.css**: folha de estilo com o código CSS para estilizar os componentes da tela. Esse arquivo deve ser adicionado dentro da tag head da página HTML.
* **Bootstrap.bundle.min.js**: arquivo de script para o correto funcionamento dos componentes visuais. Esse arquivo deve ser adicionado antes do fechamento da tag body da página HTML.

Vamos adicionar os dois arquivos tanto na página de cadastro do cliente quanto na página de gerenciamento de cadastro do cliente por meio dos links da CDN fornecidos pelo próprio Bootstrap.

Portanto, nos arquivos HTML referentes a essas páginas, adicione o código do quadro 12 dentro da tag head e o código do quadro 13 antes do fechamento da tag body.

Quadro 12 – Folha de estilo CDN



Quadro 13 – Arquivo de script CDN



Adicionados os arquivos às páginas, basta aplicar aos componentes HTML às classes do Bootstrap para estilizá-los. Por meio do site [<https://getbootstrap.com/>](https://getbootstrap.com/), você tem acesso à documentação desse framework, em que são apresentados diversos exemplos de estilos e o seu respectivo código HTML. Na figura a seguir, temos uma tela estilizada com as classes do Bootstrap e, no quadro 14, o seu código HTML, ambos extraídos da documentação do framework.

Figura 13 – Formulário estilizado com Bootstrap

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Quadro 14 – Código fonte do formulário estilizado com Bootstrap

Caixa de Texto: <form>
  <div class="mb-3">
    <label for="exampleInputEmail1" class="form-label">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1" aria-describedby="emailHelp">
    <div id="emailHelp" class="form-text">We'll never share your email with anyone else.</div>
  </div>
  <div class="mb-3">
    <label for="exampleInputPassword1" class="form-label">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1">
  </div>
  <div class="mb-3 form-check">
    <input type="checkbox" class="form-check-input" id="exampleCheck1">
    <label class="form-check-label" for="exampleCheck1">Check me out</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>


Veja que, de forma simples, conseguimos estilizar a interface de uma página HTML, bastando para isso importar as bibliotecas do framework e vincular as classes do Bootstrap aos componentes da tela.

**5.1 FORMULÁRIO DE CADASTRO DO CLIENTE**

No arquivo referente ao formulário de cadastro do cliente, vamos adicionar algumas classes aos componentes que compõem essa tela. Dentre elas, destacam-se:

Tabela 2 – Classes aplicadas ao formulário de cadastro

|  |  |
| --- | --- |
| **Classe** | **Descrição** |
| **container** | Classe que centraliza o formulário com espaçamentos laterais |
| **mb** | Classe que define o espaçamento entre elementos |
| **form-check** | Classe para estilizar o elemento que contém a label e o campo de entrada do tipo checkbox |
| **form-switch** | Classe para estilizar o checkbox no formato switch |
| **form-check-label** | Classe para estilizar a label do checkbox |
| **form-check-input** | Classe para estilizar um campo de entrada do tipo checkbox |
| **form-control** | Classe para estilizar um campo de entrada do tipo texto |
| **row** | Define o elemento com uma linha, tornando possível definir o tamanho e demais elementos que compõem essa linha |
| **col** | Define o tamanho de um elemento dentro de uma linha |
| **btn** | Deixa o elemento com formato de um botão, podendo essa classe ser aplicada a links |
| **btn-primary** | Deixa o botão na cor azul (botão principal da tela) |

No quadro a seguir, temos o código fonte do formulário de cadastro do cliente utilizando as classes do Bootstrap, que irão resultar na interface vista na Figura 14.

Quadro 15 – Código fonte do formulário estilizado com Bootstrap





Figura 14 – Interface do formulário de cadastro do cliente

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

**5.2 TELA DE GERENCIAMENTO DE CADASTRO DO CLIENTE**

No arquivo referente à tela de gerenciamento de cadastro do cliente, além das classes citadas anteriormente, vamos adicionar outras classes, das quais destacam-se:

Tabela 3 – Classes aplicadas na tela de gerenciamento de cadastro do cliente

|  |  |
| --- | --- |
| **Classe** | **Descrição** |
| **table** | Classe para estilizar uma tabela |
| **table-hover** | Destaca a linha da tabela na qual o cursor está posicionado |
| **btn-secundary** | Deixa o botão na cor cinza (botão secundário da tela) |
| **btn-sm** | Reduz o tamanho do botão |

No quadro a seguir, temos o código fonte da tela de gerenciamento de cadastro do cliente utilizando as classes do Bootstrap, que irão resultar na interface vista na figura 15.

Quadro 16 – Código fonte da tela de gerenciamento estilizado com Bootstrap



Caixa de Texto:     <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="${cliente.dataDeNascimento}"></td>
          <td th:text="${cliente.cpf}"></td>
          <td>
            <a class="btn btn-sm btn-secondary"
             th:href="@{/clientes/edita/{id}(id=${cliente.id})}">Editar</a>
          </td>
          <td>
          <a th:href="@{/clientes/ativa/{id}(id=${cliente.id})}" 
             class="btn btn-sm btn-secondary"
             th:if="${cliente.ativo == false}" >Ativar</a>
          <a th:href="@{/clientes/inativa/{id}(id=${cliente.id})}"
             class="btn btn-sm btn-secondary"
             th:unless="${cliente.ativo == false}" >Inativar</a>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</body>
</html>



Figura 15 – Interface da tela de gerenciamento de cadastro do cliente

Tabela

Descrição gerada automaticamente

**FINALIZANDO**

Nesta aula, aprendemos a desenvolver os controladores da aplicação, componente responsável por atender às requisições efetuadas pelo usuário. Além disso, desenvolvemos também a tela de gerenciamento de cadastro do cliente, fornecendo recursos para que o usuário possa realizar um cadastro, editar os dados cadastrais de um determinado cliente, além de ativar/inativar um cadastro específico. Ademais, adicionamos as bibliotecas do Bootstrap às páginas HTML com o intuito de torná-las mais atrativas comercialmente e recursivas, fazendo com que a interface gráfica da aplicação se adapte ao dispositivo no qual ela está sendo executada.

**LINGUAGEM DE PROGRAMAÇÃO**

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


Ao adicionar as anotações referentes a validação dos atributos da classe Cliente, deve-se adicionar a mensagem que será exibida caso o conteúdo do atributo seja inválido. Para isso, deve-se utilizar a propriedade *message*, conforme podemos observar nas anotações @NotNull, @CPF e @NotBlack no

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 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

Interface gráfica do usuário, Texto, Aplicativo

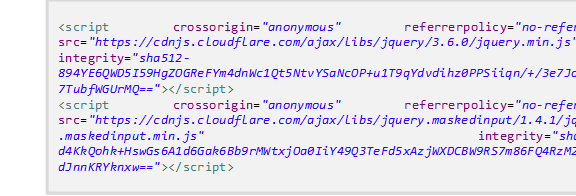
Descrição gerada automaticamente

**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



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

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

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 tarefas 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();
   }
   return "redirect:/clientes";
}


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

Tabela

Descrição gerada automaticamente

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

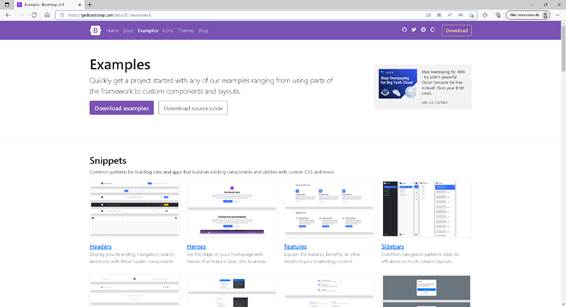
**TEMA 4 – CRIANDO A TELA INICIAL DO SISTEMA**

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.

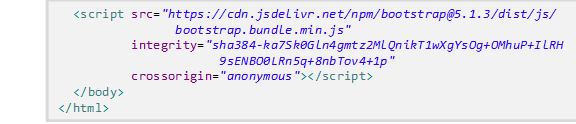
Figura 4 – Templates do Bootstrap



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





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

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

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

Uma imagem contendo Interface gráfica do usuário

Descrição gerada automaticamente

**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,

I. A classe de configuração de segurança da aplicação deve ser uma extensão da classe SecurityWebConfigurerAdapter. **VERDADEIRO**

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.

II. A classe de configuração de segurança da aplicação deverá ser mapeada com as anotações @Configuration e @EnableWebSecurity. **VERDADEIRO**

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



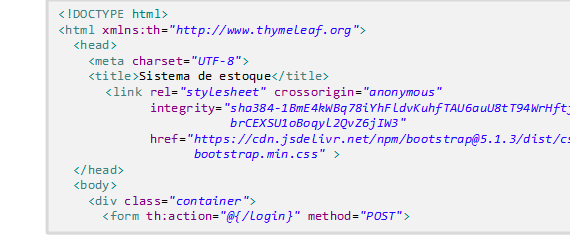
Concluídas as devidas configurações referentes à autenticação e permissão de acesso, deve-se implementar a tela de login da aplicação e criar um método dentro da classe AplicacaoController para exibi-la. Portanto, adicione à classe AplicacaoController o método login, conforme mostra o Quadro 20.

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



        <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" 
                 autofocos />
        </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

Interface gráfica do usuário, Aplicativo, Word

Descrição gerada automaticamente

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.

Questão 1/10 - Desenvolvimento Web - Back End

Com relação aos controladores, analise as sentenças a seguir:

I. Toda classe que implementa um controlador deve ser anotada com @Controller

II. A anotação @RequestMapping é responsável por configurar a url path das requisições que serão atendidas pelo controlador

III. Para que o controlador possa processar uma requisição do tipo POST, deve-se especificar esse método de requisição no atributo type da anotação @RequestMapping

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas  **Você acertou!**  AULA 05 - TEMA 01 |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 2/10 - Desenvolvimento Web - Back End

Dentre as alternativas a seguir, selecione a tag da linguagem HTML que permite adicionar um campo de seleção ao documento.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | option |
|  | B | select  **Você acertou!**  (Aula 04 – Tema 04) |
|  | C | input |
|  | D | form |
|  | E | label |

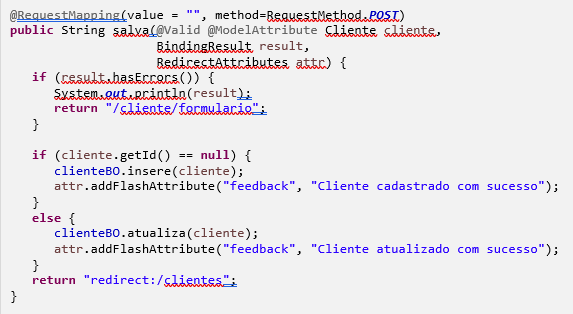
Questão 3/10 - Desenvolvimento Web - Back End

Para que o método de uma classe de teste seja executado pelo JUnit, este deve ser anotado com qual anotação dentre as alternativas a seguir?

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | @Autowired |
|  | B | @SpringBootTest |
|  | C | @Test  AULA 04 TEMA 2 |
|  | D | @JUnit |
|  | E | @Validation |

Questão 4/10 - Desenvolvimento Web - Back End

Baseado no código abaixo, podemos afirmar que a classe BindingResult está sendo utilizada para:  
  


Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Retornar uma mensagem de *feedback* para o usuário |
|  | B | Verificar se o objeto cliente possui algum erro de validação  **Você acertou!**   (Aula 06 – Tema 03) |
|  | C | Realizar a persistência de dados |
|  | D | Verificar se ocorreu algum erro durante o processo de persistência dos dados |
|  | E | Processar um requisição do tipo POST |

Questão 5/10 - Desenvolvimento Web - Back End



Adotando que a aplicação está sendo executada localmente na porta 8080, analise o código acima e julgue as sentenças a seguir:

I. O método novo da classe ClienteController será executado quando for efetuada uma requisição na url http://localhost:8080/clientes/novo do tipo GET

II. O método salva da classe ClienteController será executado quando for efetuada uma requisição na url http://localhost:8080/clientes do tipo POST

III. Após o método salva da classe ClienteController ser executado, a aplicação será redirecionada para a url http://localhost:8080/clientes/novo

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas  Aula 05 - Tema 02 |

Questão 6/10 - Desenvolvimento Web - Back End

Dentre as alternativas abaixo, assinale o framework de teste padrão de uma aplicação criada por meio do Spring Boot.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | JDBC |
|  | B | JPA |
|  | C | JTest |
|  | D | JSTL |
|  | E | JUnit  **Você acertou!**  (Aula 04 – Tema 03) |

Questão 7/10 - Desenvolvimento Web - Back End

Dentre as alternativas a seguir, selecione a anotação responsável por especificar a url que será processada por um determinado controller.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | @RequestMapping  **Você acertou!**  (Aula 05 – Tema 02) |
|  | B | @Url |
|  | C | @UrlPattern |
|  | D | @Http |
|  | E | @Controller |

Questão 8/10 - Desenvolvimento Web - Back End

A respeito do Spring Security, analise as sentenças a seguir:

I. A classe de configuração de segurança da aplicação deve ser uma extensão da classe SecurityWebConfigurerAdapter.

II. A classe de configuração de segurança da aplicação deverá ser mapeada com as anotações @Configuration e @EnableWebSecurity.

III. A classe HttpSecurity fornece os métodos necessários para efetuar a configuração das permissões de acesso de cada URL da aplicação.

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas  AULA 06 - TEMA 05 |

Questão 9/10 - Desenvolvimento Web - Back End



Adotando que a aplicação está sendo executada localmente na porta 8080, analise o código acima e julgue as sentenças a seguir:

I. Após realizar o logout, o usuário será redirecionado para a página de login

II. Os usuários com permissão de COLABORADOR têm permissão para acessar a url http://localhost:8080/nota-entrada

III. O método antMatchers é utilizado para especificar qual url será acessada mediante permissão de acesso

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas  **Você acertou!**  AULA 06 - TEMA 05 |

Questão 10/10 - Desenvolvimento Web - Back End

Com relação as anotações de validação, analise as sentenças a seguir:

I. A anotação @NotNull não permite que o atributo da classe seja nulo ou vazio

II. A anotação @Min especifica o mínimo de caracteres que um atributo do tipo String deve conter

III. A anotação @Max especifica o máximo de caracteres que um atributo do tipo String deve conter

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta  AULA 06 - TEMA 01 |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 1/10 - Desenvolvimento Web - Back End



Adotando que a aplicação está sendo executada localmente na porta 8080, analise o código acima e julgue as sentenças a seguir:

I. O método novo da classe ClienteController será executado quando for efetuada uma requisição na url http://localhost:8080/clientes/novo do tipo GET

II. O método salva da classe ClienteController será executado quando for efetuada uma requisição na url http://localhost:8080/clientes do tipo POST

III. Após o método salva da classe ClienteController ser executado, a aplicação será redirecionada para a url http://localhost:8080/clientes/novo

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas  **Você acertou!**  Aula 05 - Tema 02 |

Questão 2/10 - Desenvolvimento Web - Back End

Dentre as alternativas a seguir, selecione a tag da linguagem HTML que permite adicionar um campo de seleção ao documento.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | option |
|  | B | select  **Você acertou!**  (Aula 04 – Tema 04) |
|  | C | input |
|  | D | form |
|  | E | label |

Questão 3/10 - Desenvolvimento Web - Back End

Com relação a validação dos atributos de uma classe, assinale a alternativa correta com relação a anotação @NotBlank.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Não permite que o atributo seja nulo ou vazio  **Você acertou!**  Aula 06 – Tema 01 |
|  | B | Não permite que o valor do atributo seja vazio |
|  | C | Não permite que o valor do atributo seja nulo |
|  | D | Especifica o valor mínimo de um atributo do tipo numérico |
|  | E | Especifica que o atributo é constituído apenas por números |

Questão 4/10 - Desenvolvimento Web - Back End

O Thymeleaf é um *template engine* para projetos Java que facilita a criação de páginas HTML dinâmicas. Ao submeter um formulário desenvolvido por meio desse *template engine*, precisamos especificar a url para qual esse formulário será submetido. Isso pode ser especificado por meio do atributo:

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | th:object |
|  | B | th:url |
|  | C | th:action  **Você acertou!**  (Aula 04 – Tema 05) |
|  | D | th:method |
|  | E | th:controller |

Questão 5/10 - Desenvolvimento Web - Back End



Adotando que a aplicação está sendo executada localmente na porta 8080, analise o código acima e julgue as sentenças a seguir:

I. Após realizar o logout, o usuário será redirecionado para a página de login

II. Os usuários com permissão de COLABORADOR têm permissão para acessar a url http://localhost:8080/nota-entrada

III. O método antMatchers é utilizado para especificar qual url será acessada mediante permissão de acesso

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas  AULA 06 - TEMA 05 |

Questão 6/10 - Desenvolvimento Web - Back End

Para que o método de uma classe de teste seja executado pelo JUnit, este deve ser anotado com qual anotação dentre as alternativas a seguir?

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | @Autowired |
|  | B | @SpringBootTest |
|  | C | @Test  AULA 04 TEMA 2 |
|  | D | @JUnit |
|  | E | @Validation |

Questão 7/10 - Desenvolvimento Web - Back End

Dentre as alternativas abaixo, selecione aquela que corresponde a uma biblioteca de JavaScript.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | JQuery  **Você acertou!**  (Aula 06 – Tema 02) |
|  | B | Java |
|  | C | Spring |
|  | D | CSS |
|  | E | HTML |

Questão 8/10 - Desenvolvimento Web - Back End



Analisando a classe de teste acima, julgue as sentenças a seguir:

I. Após executar a classe ClienteBOTest, o JUnit irá executar os três métodos da classe e indicar quais métodos foram executados com sucesso e quais apresentaram erro durante a sua execução.

II. A anotação @ExtendWith integra o Spring com o JUnit, framework de teste unitário da linguagem Java, responsável por validar se os métodos estão funcionando corretamente.

III. A anotação @SpringBootTest, inicializa todo o contêiner da aplicação, desta forma podemos realizar a injeção de dependência do objeto de negócio, por meio da anotação @Autowired.

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas  AULA 04 - TEMA 01 |
|  | E | Todas as sentenças estão corretas |

Questão 9/10 - Desenvolvimento Web - Back End

Com relação ao Spring Security, analise as sentenças a seguir:

I. O Spring Security prove uma poderosa e altamente personalizável estrutura de autenticação e controle de acesso

II. O Spring Security é uma dependência da plataforma Java EE

III. O Spring Security protege a aplicação de diversos ataques como session fixation, clickjacking, CSRF (Cross-Site Request Forgery), entre outros

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas  **Você acertou!**  AULA 06 - TEMA 05 |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 10/10 - Desenvolvimento Web - Back End

Com relação as anotações de validação, analise as sentenças a seguir:

I. A anotação @NotNull não permite que o atributo da classe seja nulo ou vazio

II. A anotação @Min especifica o mínimo de caracteres que um atributo do tipo String deve conter

III. A anotação @Max especifica o máximo de caracteres que um atributo do tipo String deve conter

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta  **Você acertou!**  AULA 06 - TEMA 01 |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 1/10 - Desenvolvimento Web - Back End

Para que um objeto possa ser validado conforme as anotações da biblioteca javax.validation, ele deve ser anotado com qual das anotações a seguir?

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | @Valid  **Você acertou!**  (Aula 06 – Tema 01) |
|  | B | @BindingResult |
|  | C | @BeanValidation |
|  | D | @Error |
|  | E | @HasError |

Questão 2/10 - Desenvolvimento Web - Back End

A tag <td> da linguagem HTML é utilizada para:

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Adicionar uma linha à tabela |
|  | B | Adicionar uma coluna à tabela |
|  | C | Adicionar uma célula à tabela a linha da tabela  **Você acertou!**  (Aula 05 – Tema 03) |
|  | D | Adicionar um cabeçalho à tabela |
|  | E | Adicionar o título da tabela |

Questão 3/10 - Desenvolvimento Web - Back End

Com relação as tags da linguagem HTML, analise as sentenças a seguir:

I. A tag label adiciona uma etiqueta ao documento

II. A tag input adiciona um campo de entrada ao documento

III. A tag select adiciona as opções para um campo de seleção

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas  (Aula 04 - Tema 04) |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 4/10 - Desenvolvimento Web - Back End

Com relação ao Bootstrap, analise as sentenças a seguir:

I. O Bootstrap é um framework de estilo utilizado no front-end

II. A classe container do Bootstrap é utilizada para estilizar o menu da aplicação

III. Ao aplicar a classe table-hover do Bootstrap em uma tabela, a linha da tabela na qual o cursor está posicionado ficará destacada

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas  **Você acertou!**  AULA 05 - TEMA 05 |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 5/10 - Desenvolvimento Web - Back End

Dentre as alternativas a seguir, selecione a tag da linguagem HTML que permite adicionar um campo de seleção ao documento.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | option |
|  | B | select  **Você acertou!**  (Aula 04 – Tema 04) |
|  | C | input |
|  | D | form |
|  | E | label |

Questão 6/10 - Desenvolvimento Web - Back End

Dentre as alternativas abaixo, selecione o módulo do Spring que prove uma poderosa e altamente personalizável estrutura de autenticação e controle de acesso.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Spring Web |
|  | B | Spring Security  **Você acertou!**  (Aula 06 – Tema 05) |
|  | C | Spring Validation |
|  | D | Spring Authentication |
|  | E | Spring Access |

Questão 7/10 - Desenvolvimento Web - Back End



Adotando que a aplicação está sendo executada localmente na porta 8080 e que a classe na qual o método acima foi implementado está anotada com @RequestMapping(“/clientes”), analise o código referente ao método inativa e julgue as sentenças a seguir:

I. O método inativa pertence a uma classe do tipo Controller

II. Ao efetuar a requisição da url http://localhost:8080/clientes/inativa?id=1 o método inativa será executado

III. Após o método inativa ser executado, a aplicação será redirecionada para a url http://localhost:8080/clientes

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas  AULA 05 - TEMA 04 |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 8/10 - Desenvolvimento Web - Back End

Dentre as alternativas a seguir, selecione a linguagem de marcação de hipertexto utilizada para o desenvolvimento da interface gráfica das páginas Web por meio de tags.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | HTML  **Você acertou!**  (Aula 04 – Tema 04) |
|  | B | Java |
|  | C | JavaScript |
|  | D | Spring |
|  | E | CSS |

Questão 9/10 - Desenvolvimento Web - Back End

A respeito das anotações utilizadas na implementação do controlador, analise as sentenças a seguir:

I. A anotação @PathVariable é utilizada para obter o valor de um atributo que compõem o cabeçalho de uma requisição Web.

II. A anotação @RequestParam é utilizada para mapear uma requisição Web.

III. A anotação @ModelMap é utilizada para mapear o objeto que irá receber os dados do front-end

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta  **Você acertou!**  AULA 05 - TEMA 04 |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas |
|  | E | Todas as sentenças estão corretas |

Questão 10/10 - Desenvolvimento Web - Back End

Com relação a implementação de tabelas por meio da linguagem HTML, analise as sentenças a seguir:

I. A tag <tbody> define o cabeçalho da tabela e a tag <thead> define o corpo da tabela

II. A tag <tr> é utilizada para adicionar uma linha à tabela

III. A tag <td> é utilizada para adicionar uma célula à linha da tabela

De acordo com as sentenças apresentadas, assinale a alternativa correta.

Nota: 10.0

|  |  |  |
| --- | --- | --- |
|  | A | Nenhuma sentença está correta |
|  | B | Somente as sentenças I e II estão corretas |
|  | C | Somente as sentenças I e III estão corretas |
|  | D | Somente as sentenças II e III estão corretas  **Você acertou!**  AULA 05 - TEMA 03 |