



# DESENVOLVIMENTO WEB BACK END

AULA 4



Prof. Rafael Moraes

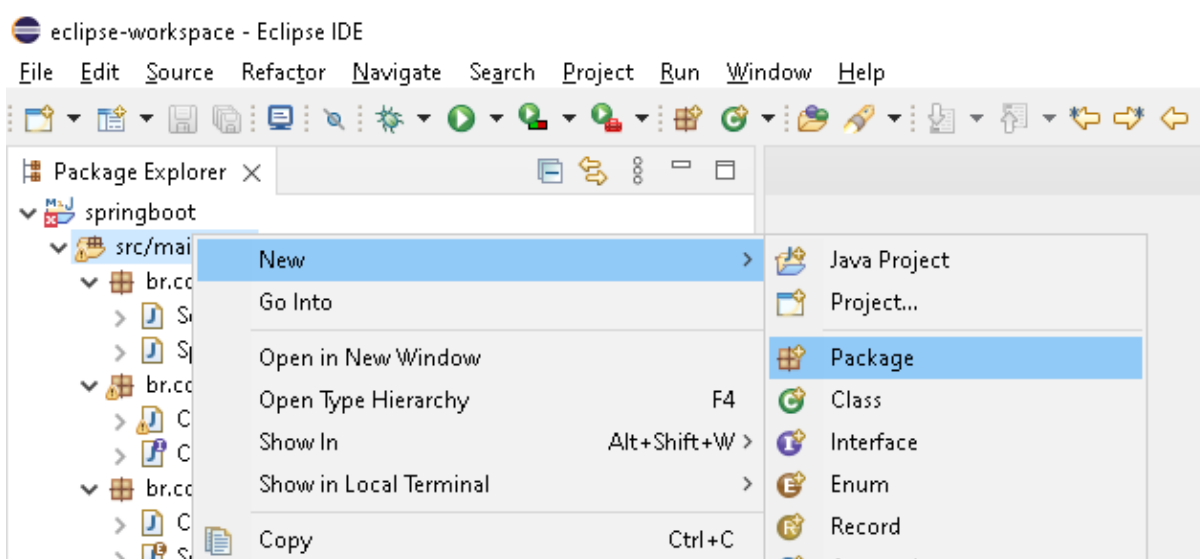
## CONVERSA INICIAL

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:

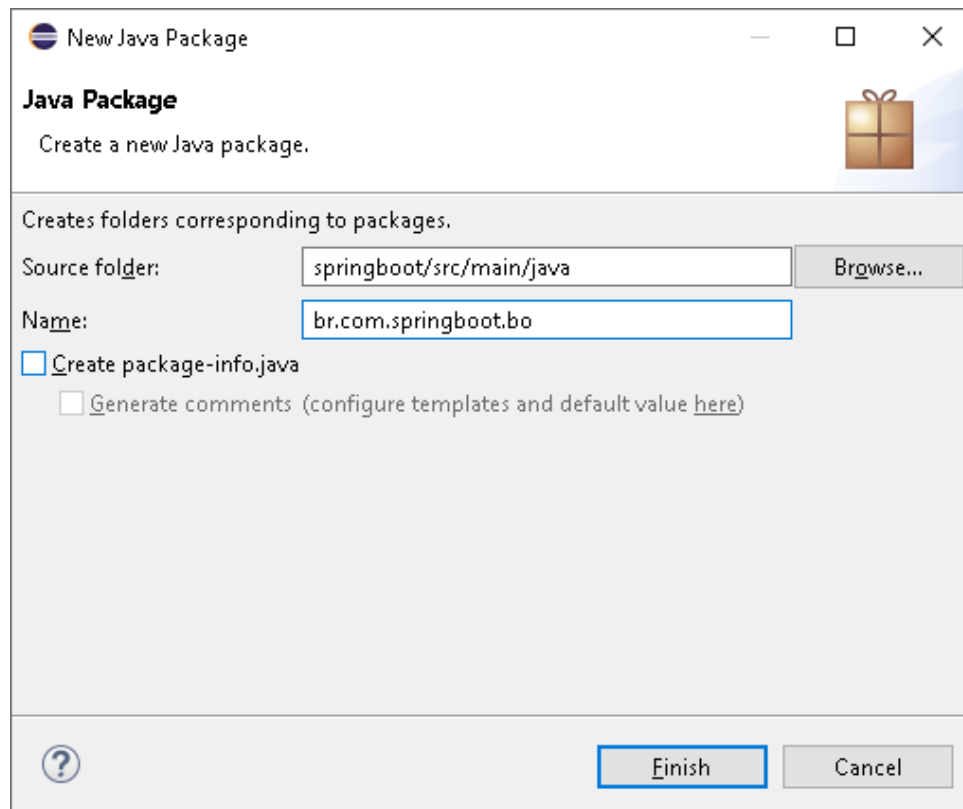
Figura 1 – Menu para criação do pacote



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

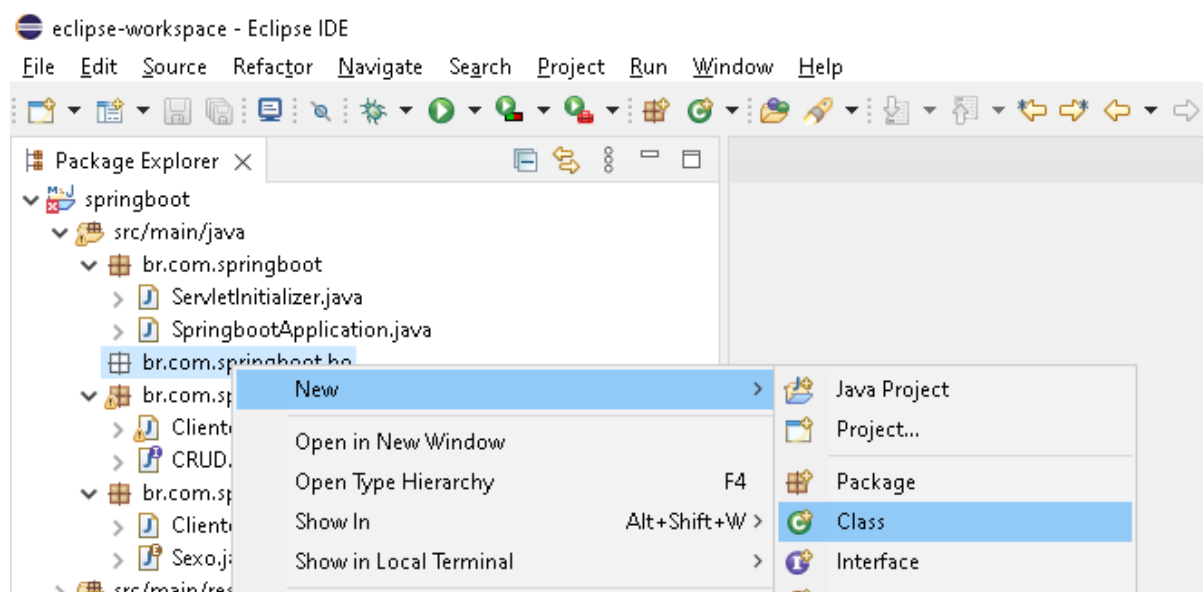


Figura 2 – Tela de criação do pacote



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

**New Java Class**

**Java Class**  
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

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`.

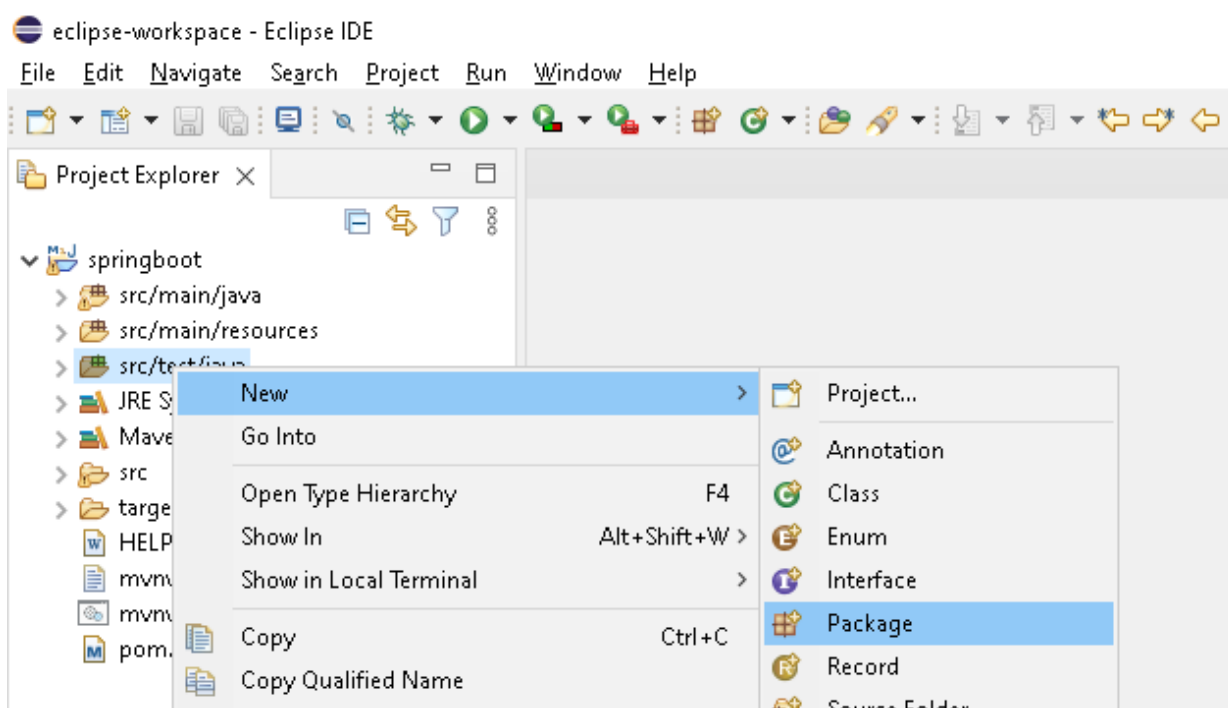
Também foi criado o método `inativa`, o qual será responsável por inativar o cadastro de um determinado cliente. Em algumas situações, devemos optar por inativar um cadastro em vez de excluí-lo. Ao inativar, tornamos o cadastro invisível dentro do sistema, dando a percepção para o usuário de que o cadastro 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:

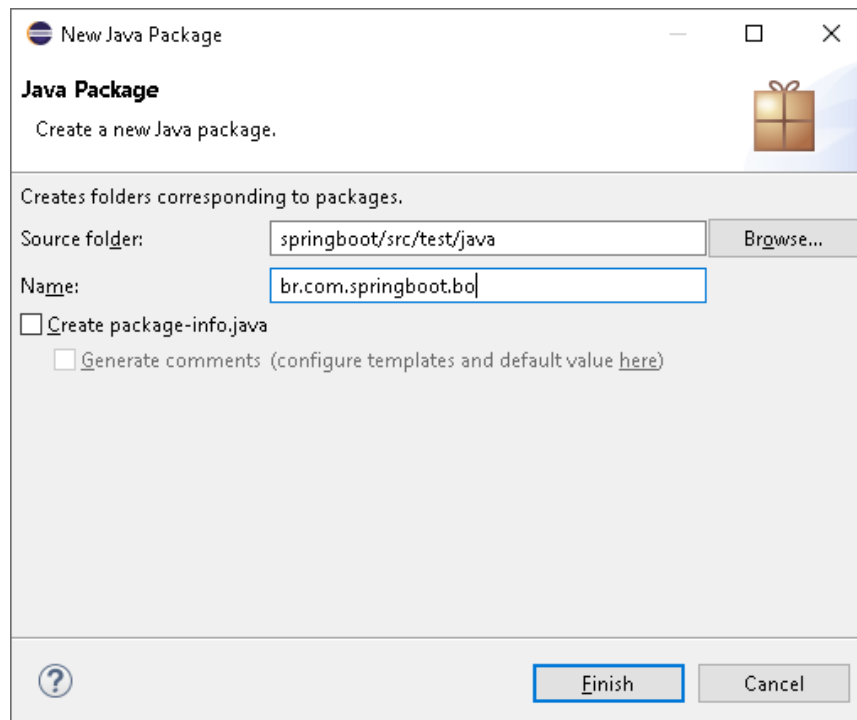
Figura 5 – Menu para criação do pacote



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

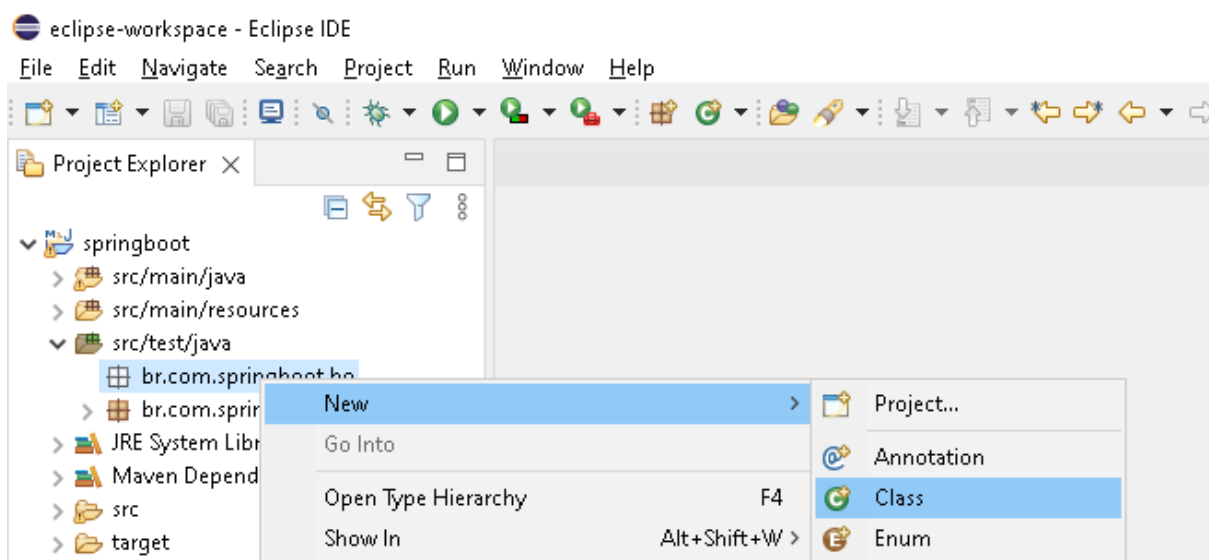


Figura 6 – Criação do pacote `br.com.springboot.bo`



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:

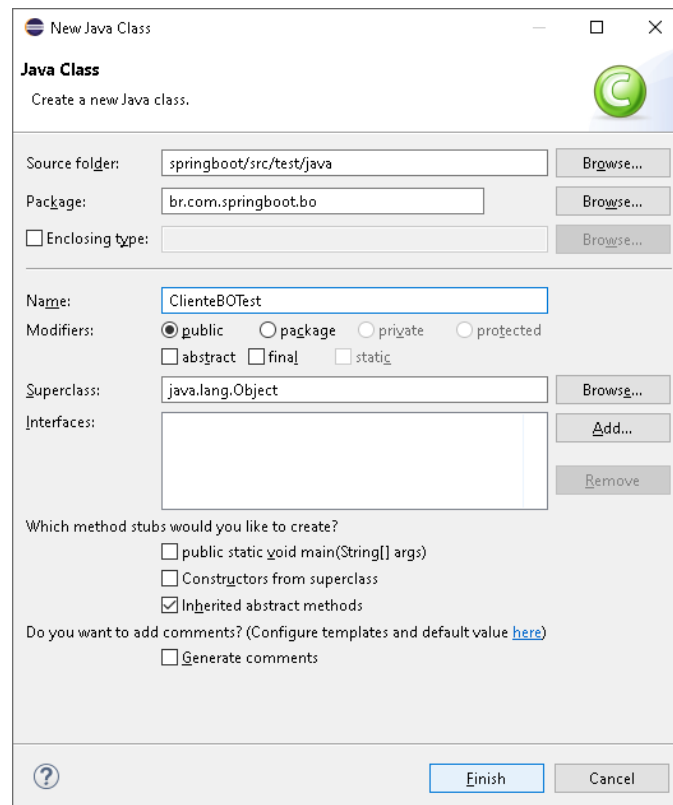
Figura 7 – Menu para criação da classe



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



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("413333333");
    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 `pesquisaPorId` 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 `remove` 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

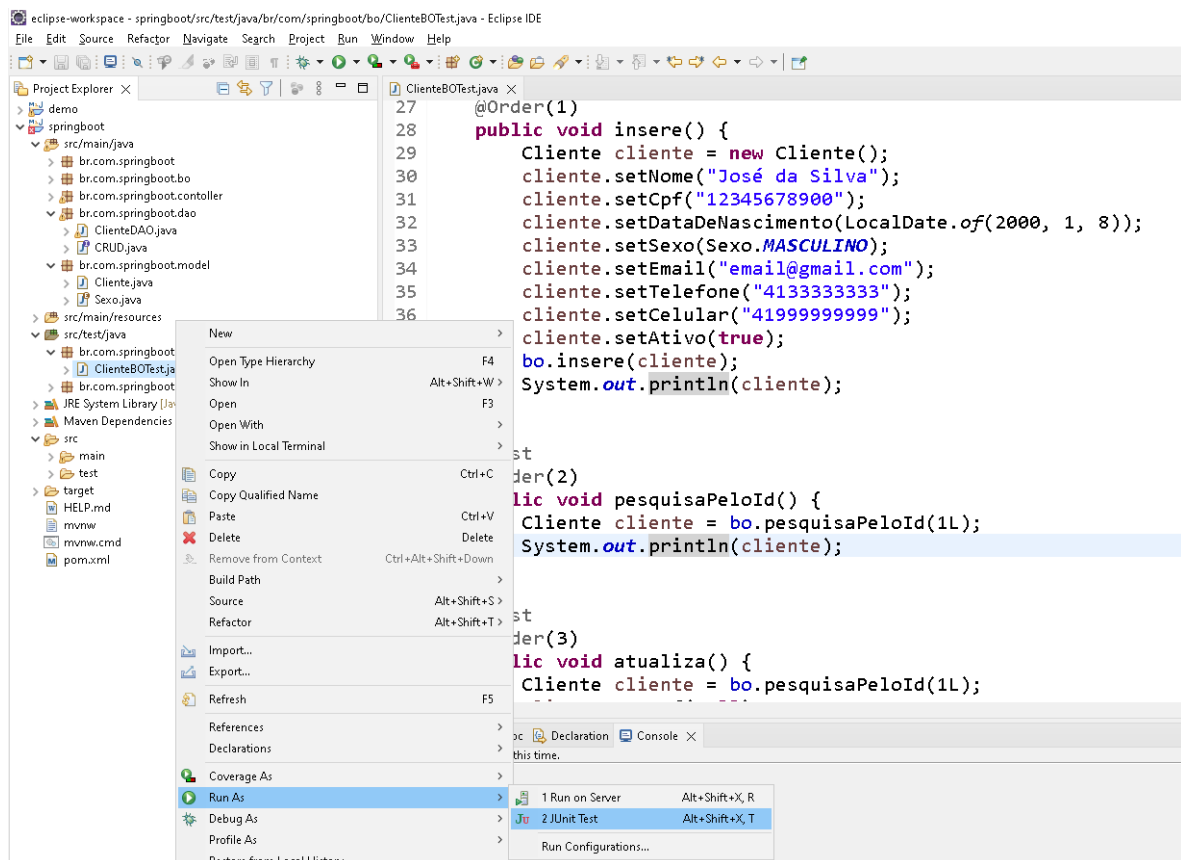
Antes de iniciarmos os testes, vamos adicionar o método `toString` à classe `Cliente`. Esse método irá formatar o objeto sempre que tentarmos convertê-lo para `String`. Dessa forma, podemos utilizar o comando `system.out.print` para imprimir o conteúdo desse objeto. O método `toString` pode ser verificado 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

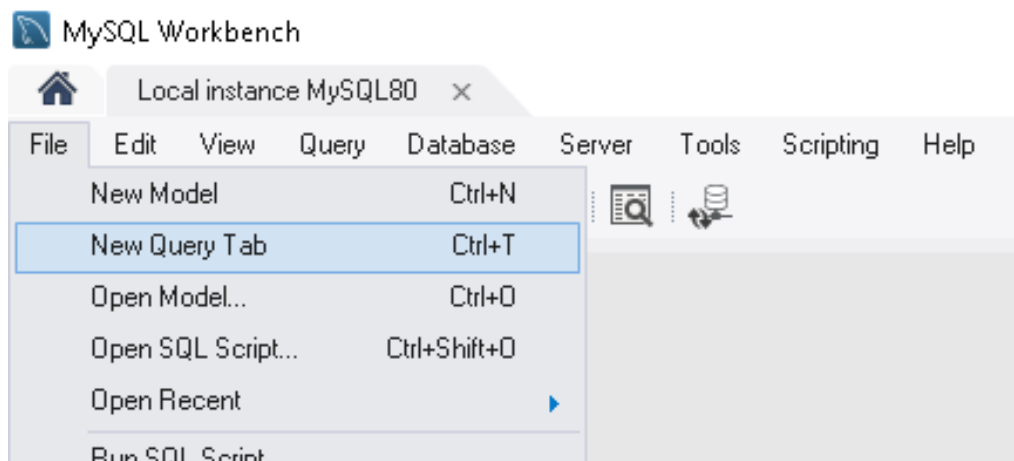


pela anotação @Order. O primeiro método a ser executado, conforme visto no tema anterior, é o método `insere`. Esse método irá inserir um registro na tabela `clientes` com as seguintes informações:

- 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*



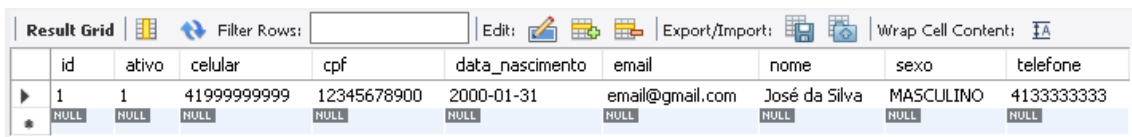
Na aba que foi aberta, informe os seguintes comandos SQL conforme o quadro 10. Substitua o termo `[base_dados]` pelo nome da base de dados que foi criada para armazenar os dados da aplicação. O primeiro comando será para conectar-se à base de dados. Por sua vez, o segundo comando será para listar todos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

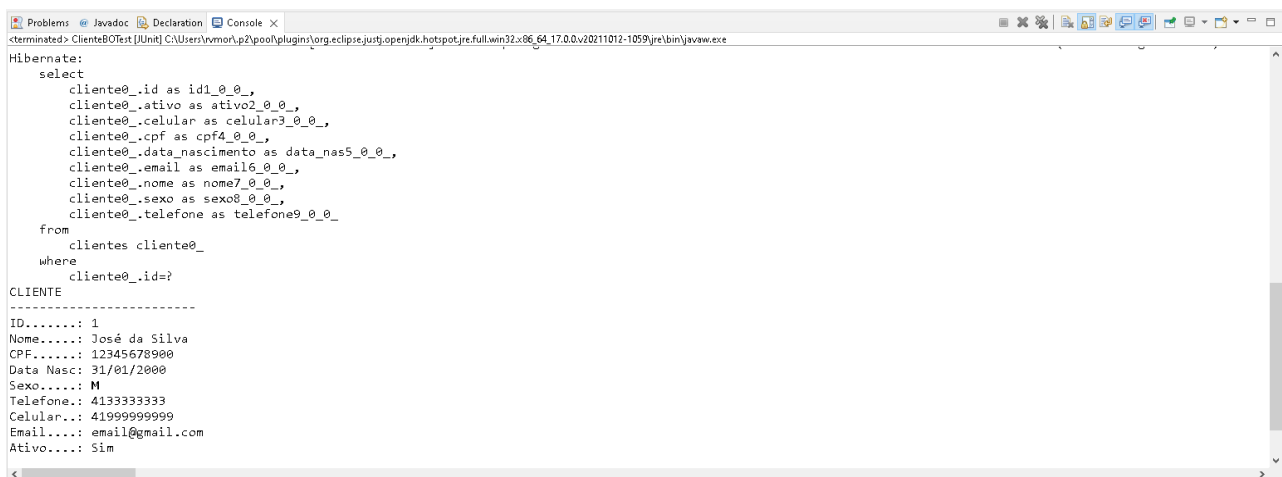
Figura 11 – Cadastro inserido



	id	ativo	celular	cpf	data_nascimento	email	nome	sexo	telefone
▶	1	1	41999999999	12345678900	2000-01-31	email@gmail.com	José da Silva	MASCULINO	4133333333
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

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`



```
Hibernate:
select
  cliente0_.id as id1_0_0_,
  cliente0_.ativo as ativo2_0_0_,
  cliente0_.celular as celular3_0_0_,
  cliente0_.cpf as cpf4_0_0_,
  cliente0_.data_nascimento as data_nas5_0_0_,
  cliente0_.email as email6_0_0_,
  cliente0_.nome as nome7_0_0_,
  cliente0_.sexo as sexo8_0_0_,
  cliente0_.telefone as telefone9_0_0_
from
  clientes cliente0_
where
  cliente0_.id=?

CLIENTE
-----
ID.....: 1
Nome.....: José da Silva
CPF.....: 12345678900
Data Nasc.: 31/01/2000
Sexo.....: M
Telefone..: 4133333333
Celular..: 41999999999
Email.....: email@gmail.com
Ativo.....: Sim
```

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

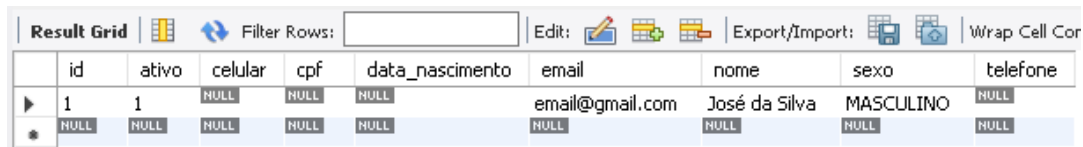
- CPF: será atribuído o valor nulo;
- Data de nascimento: será atribuído o valor nulo;
- Telefone: será atribuído o valor nulo;



- Celular: será atribuído o valor nulo.

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

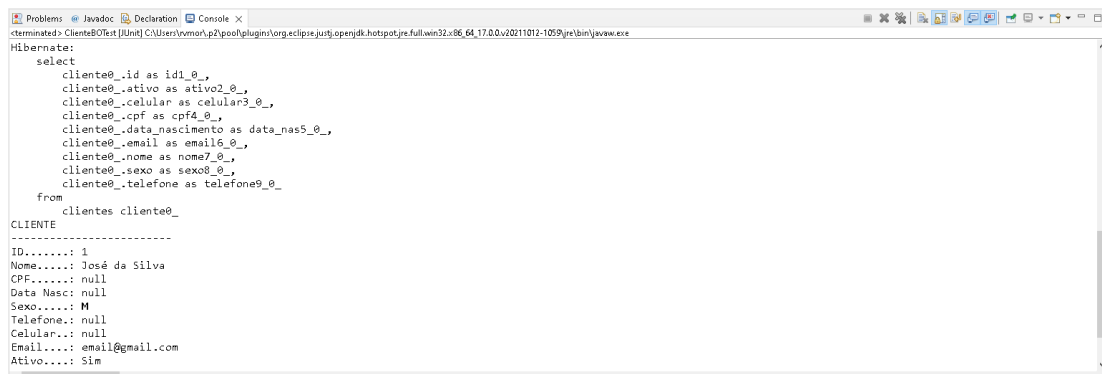
Figura 13 – Cadastro alterado



	id	ativo	celular	cpf	data_nascimento	email	nome	sexo	telefone
▶	1	1	NULL	NULL	NULL	email@gmail.com	José da Silva	MASCULINO	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

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



```

Hibernate:
select
  cliente0_.id as id1_0_,
  cliente0_.ativo as ativo2_0_,
  cliente0_.celular as celular3_0_,
  cliente0_.cpf as cpf4_0_,
  cliente0_.data_nascimento as data_nas5_0_,
  cliente0_.email as email6_0_,
  cliente0_.nome as nome7_0_,
  cliente0_.sexo as sexo8_0_,
  cliente0_.telefone as telefone9_0_
from
  clientes cliente_
CLIENTE
-----
ID.....: 1
Nome.....: José da Silva
CPF.....: null
Data Nasc.: null
Sexo.....: M
Telefone..: null
Celular...: null
Email.....: email@gmail.com
Ativo.....: Sim
```

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

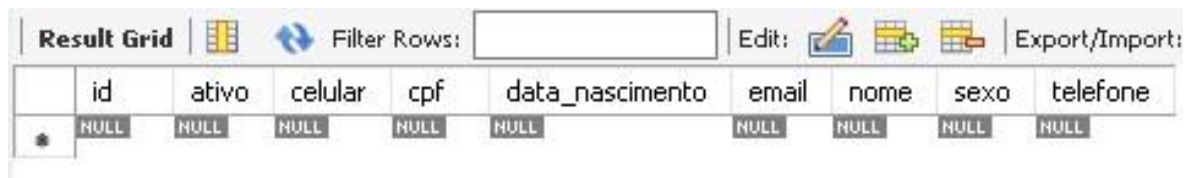


	id	ativo	celular	cpf	data_nascimento	email	nome	sexo	telefone
▶	1	0	NULL	NULL	NULL	email@gmail.com	José da Silva	MASCULINO	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL



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:

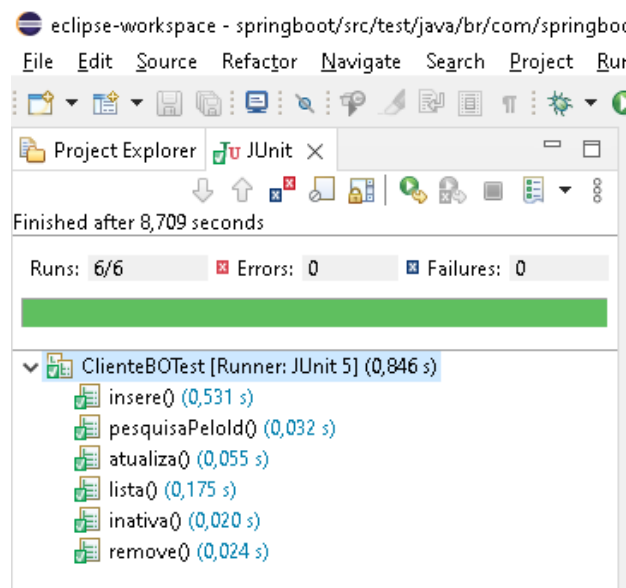
Figura 16 – Registro removido



	id	ativo	celular	cpf	data_nascimento	email	nome	sexo	telefone
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

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.

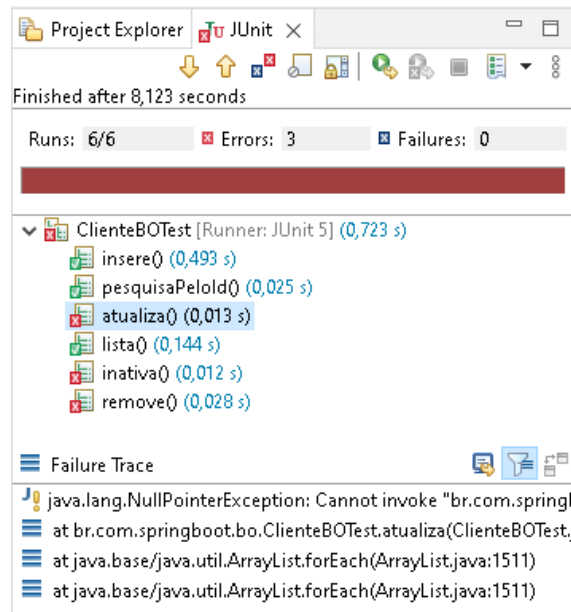
Figura 17 – Aba JUnit



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



## 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:

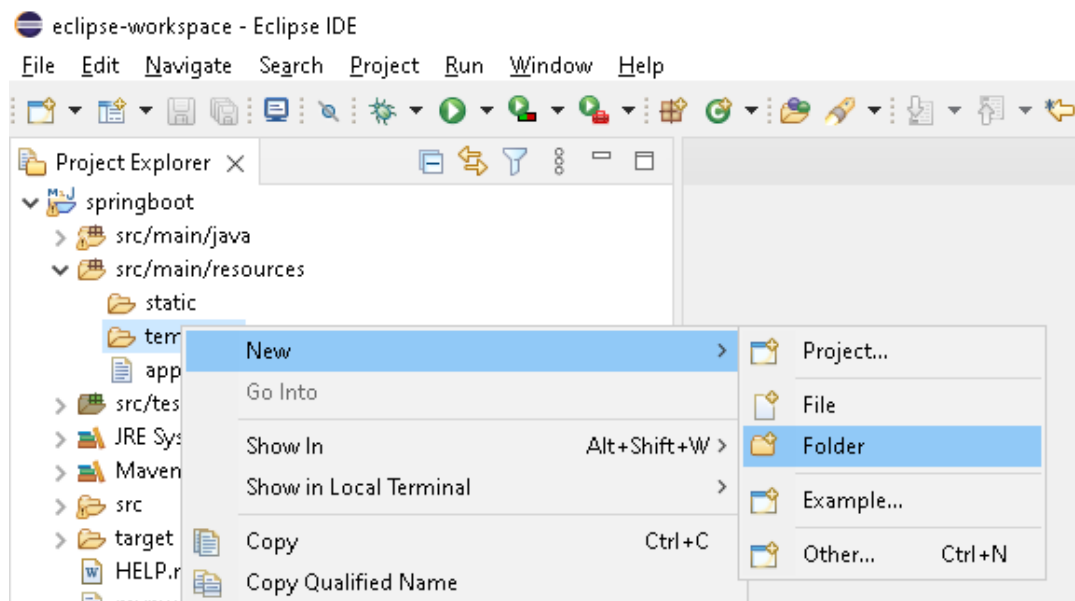
### Quadro 11 – Dependência do Thymeleaf

```
<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:

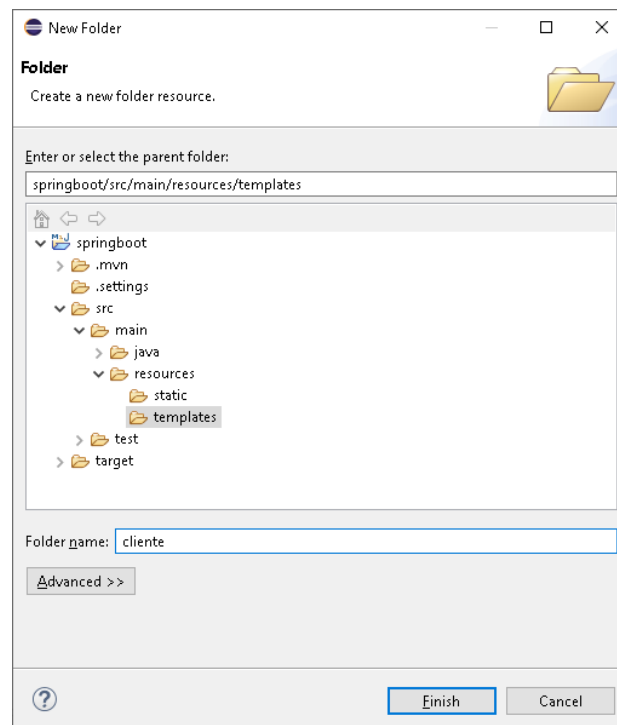


Figura 19 – Menu para criar uma pasta



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.

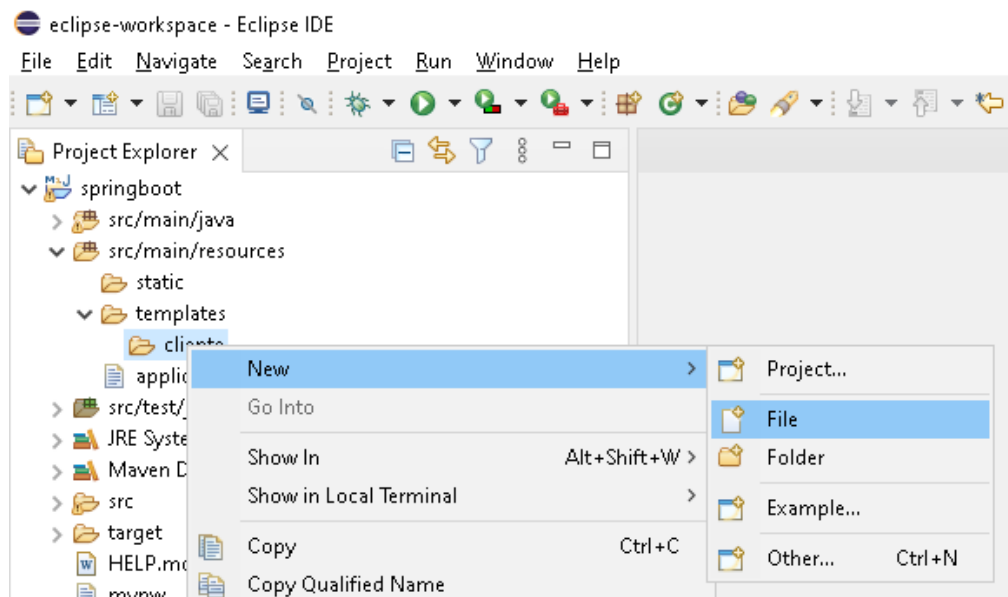
Figura 20 – Tela de criação da pasta



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.

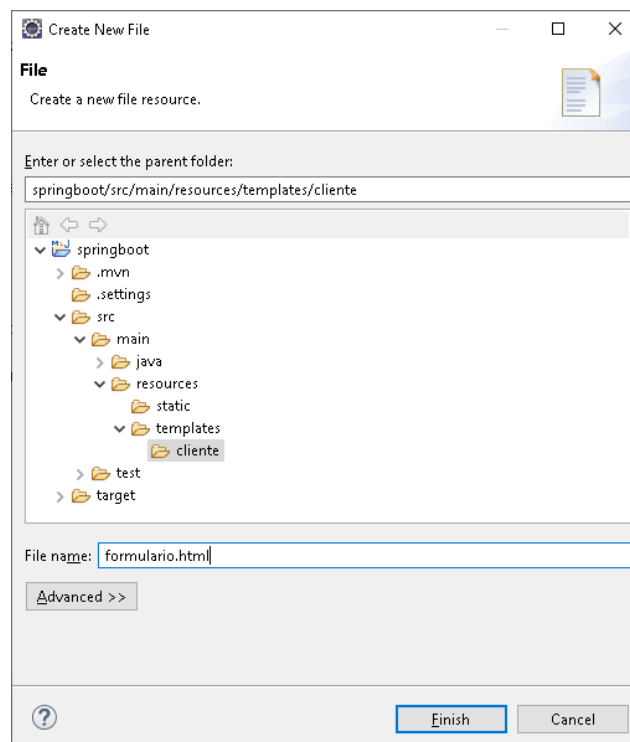


Figura 21 – Adicionando um arquivo à pasta



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



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
<b>&lt;!DOCTYPE html&gt;</b>	Indica que a página foi desenvolvida utilizando HTML5
<b>&lt;html&gt;</b>	Representa a raiz do documento HTML
<b>&lt;head&gt;</b>	Adiciona metadados que serão processados pelo navegador
<b>&lt;body&gt;</b>	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

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

HTML - Campos de entrada

Arquivo | C:/Users/rvmor,

## Formulário

Checkbox ☐

Texto

Data dd/mm/aaaa

Combobox Tipo 1

Salvar

## 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

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>

  </head>
  <body>

  </body>
</html>
```

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

The screenshot shows a web browser window with the title 'Sistema de Estoque'. The address bar shows 'Arquivo | C:/Users/rv'. The main heading is 'Dados do Cliente'. Below the heading, there is a form with the following fields:

- Registro ativo ☐
- Nome
- CPF
- Data de Nascimento  (with a calendar icon)
- Sexo  (with a dropdown arrow)
- Telefone
- Celular
- E-mail
- Salvar

## 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.



---

## REFERÊNCIAS

BECHTOLD, S. et al. **JUnit Junit 5 User Guide**. Disponível em: <<https://junit.org/junit5/docs/current/user-guide/>>. Acesso em: 9 mar. 2022.

SPRING. **Spring Testing the Web Layer**. Disponível em: <<https://spring.io/guides/gs/testing-web/>>. Acesso em: 9 mar. 2022.

THYMELEAF. **Thymeleaf Tutorial**: Thymeleaf + Spring Tutorial. Disponível em: <<https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html>>. Acesso em: 9 mar. 2022.