



# **DESENVOLVIMENTO WEB BACK END**

AULA 1

Prof. Rafael Veiga de Moraes



## CONVERSA INICIAL

Antes de começar a codificação do sistema, o desenvolvedor deve entender as necessidades de seus clientes para que, a partir disso, possa prover uma solução robusta que atenda às suas demandas. Para isso, ele deve realizar a análise de requisitos do projeto. Os requisitos são divididos em dois grupos: os requisitos funcionais (RF) e os requisitos não funcionais (RNF).

Os *requisitos funcionais* especificam todas as regras de negócio do projeto, fornecendo uma visão para o desenvolvedor de todos os recursos que devem ser implementados no sistema. Já os *requisitos não funcionais* especificam como os requisitos funcionais serão implementados, definindo, assim, a infraestrutura do projeto. Para ficar mais claro esses conceitos, vamos elencar alguns requisitos, tendo como base um sistema de gerenciamento de contas a pagar e a receber.

### Requisitos funcionais

- Manter cadastro de usuário.
- Manter cadastro de clientes.
- Manter cadastro de fornecedores.
- Manter cadastro de contas a pagar.
- Manter cadastro de contas a receber.
- Manter cadastro de forma de pagamento.
- Emissão de Nota Fiscal eletrônica (NFe).
- Gerar relatório de contas a pagar e a receber pela data de emissão.
- Gerar relatório de contas a pagar e a receber pela data da baixa.

### Requisitos não funcionais

- O sistema deve ser desenvolvido em linguagem Java.
- O tempo máximo para processamento de uma requisição não pode exceder 10s.
- O sistema gerenciador de banco de dados utilizado pela aplicação será o MySql.
- A aplicação deve ser compatível com os navegadores: Safari, Chrome, Firefox e Edge.



- O sistema deverá utilizar o serviço web (WS – Web Service) dos correios para consulta dos endereços pelo CEP.

Anteriormente, citamos apenas alguns requisitos de um sistema de contas a pagar e a receber. Em um projeto comercial, iremos nos deparar com centenas ou até milhares de requisitos a serem implementados. Portanto, desenvolver uma aplicação *Web* é uma tarefa complexa que irá demandar um considerável tempo de desenvolvimento. Se implementar os requisitos funcionais exige um grande esforço, imagine se o desenvolvedor tivesse que se preocupar com a implementação dos requisitos não funcionais, como persistência em banco de dados, interpretação das requisições HTTP, gerenciamento de sessão, gerenciamento de *threads*, serviços *web* e enviar notificações por *e-mail*.

Certamente, a complexidade do projeto seria ainda maior e talvez até o tornasse inviável. Diante disso, o desenvolvedor deve buscar ferramentas que minimizem a complexidade do projeto para que ele foque apenas na codificação das regras de negócio da aplicação, aumentando a sua produtividade e reduzindo o custo de desenvolvimento do projeto.

Para isso, iremos abordar o *Spring Framework*, uma das ferramentas mais utilizadas para o desenvolvimento de aplicações Java *Web*. Antes de abordar este *framework* em específico, precisamos compreender alguns conceitos de arquitetura de sistemas e como funciona uma aplicação Java dentro da plataforma Java EE (*Enterprise Edition*).

## TEMA 1 – ARQUITETURA DE SISTEMAS

Para compreender a arquitetura de uma aplicação Java EE, iremos abordar nesse tópico alguns conceitos referentes a arquitetura de sistemas de um modo geral. A arquitetura de uma aplicação é dividida em duas categorias: arquitetura física e arquitetura lógica.



## 1.1. Arquitetura física

Corresponde à infraestrutura necessária para execução da aplicação e pode ser dividida em três camadas distintas: cliente, servidor Java EE e servidor EIS.

### 1.1.1 Cliente

O cliente (*Client Machine*) corresponde ao dispositivo ou software utilizado para acessar a aplicação. O acesso pode ser realizado através de um dispositivo móvel utilizando um aplicativo ou estação de trabalho por meio de um navegador ou sistema *desktop*.

### 1.1.2 Servidor

O servidor é a máquina responsável por prover os serviços à aplicação e realizar a comunicação com outros servidores. A seguir, listamos algumas das responsabilidades de um servidor:

- persistência em banco de dados;
- interpretação das requisições HTTP;
- gerenciamento de sessão;
- gerenciamento de *threads*;
- gerenciamento de carga;
- serviços *web*; e
- enviar notificações por *e-mail*.

Para que o servidor possa prover os serviços, é necessário que nele seja instalado um software que implemente esses serviços. Dentro da plataforma Java EE, temos dois tipos de softwares que realizam tal tarefa: servidor de aplicação e *servlet container*.

## 1.2 Sistemas de Informação Corporativos

Os Sistemas de Informação Corporativos (EIS – *Enterprise Information System*) é um servidor externo à aplicação que fornece algum tipo de serviço para a aplicação, sendo, na maioria das vezes, composto apenas pelo servidor de banco de dados (*Database Server*).



## 1.3 Arquitetura lógica

Uma das principais preocupações do desenvolvedor é com relação à manutenção do código do projeto, pois à medida que a aplicação vai crescendo, novas classes e serviços vão sendo incorporados ao sistema. Para diminuir a complexidade e manter o código do projeto bem estruturado, flexível e de fácil compreensão, é comum dividir a arquitetura da aplicação em diferentes partes lógicas denominadas *camadas*.

### 1.3.1 Modelo de uma camada

Também conhecido como aplicações monolíticas ou centralizadas, surge nos anos 1960 com o uso de banco de dados ainda bem rudimentar, perdurando até meados dos anos 1980. É caracterizado por todo o processamento da aplicação ser realizado de forma centralizada em computadores de grande porte (*mainframes*), nos quais estão alocados todos os recursos do sistema (interface, banco de dados e a implementação das regras de negócio). Os terminais de interação com os *mainframes* são conhecidos como terminais *burros* ou *mudos*, pois eles não armazenam e tampouco processam qualquer tipo de informação.

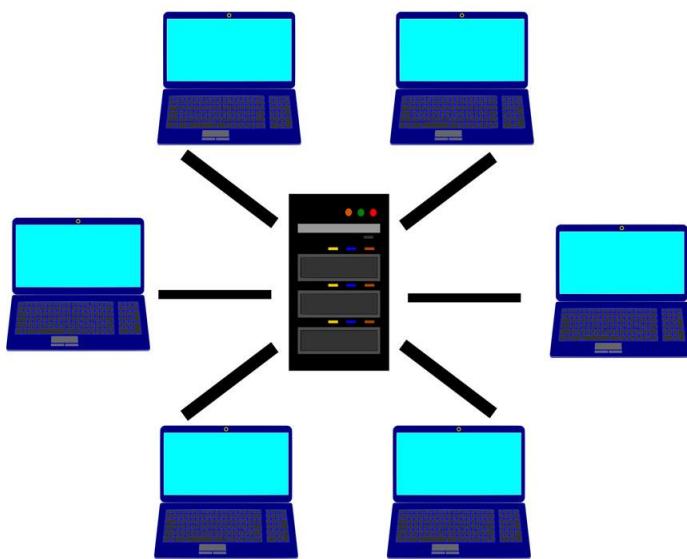
### 1.3.2 Modelo de duas camadas

Amplamente conhecido como modelo cliente-servidor, essa arquitetura predomina ao longo da década de 1980, devido às diversas transformações e inovações que ocorreram nesse período. Entre elas, destacam-se:

- surgimento das interfaces gráficas;
- a popularização dos computadores pessoais (PC – *Personal Computer*);
- a utilização de redes locais (LAN – *Local Area Network*); e
- a adoção de um SGBD (Sistema Gerenciador de Banco de Dados) para gerenciamento de dados da aplicação.



Figura 1 – Modelo cliente-servidor



Crédito: Snehalata/Shutterstock.

Uma aplicação cliente-servidor é dividida em duas camadas distintas.

- **Camada do cliente:** composta pelas estações de trabalho nas quais a aplicação está instalada, cuja finalidade é prover a interação entre o usuário e o software.
- **Camada do servidor:** representada por uma máquina dedicada a atender as requisições efetuadas pelo usuário, cuja finalidade é prover os dados para a aplicação. Nela está instalado o SGBD.

A implementação das regras de negócio da aplicação pode estar ou na camada do cliente ou na camada do servidor, porém é preferível implementá-las na camada do cliente a fim de evitar a sobrecarga de processamento no servidor. Essa solução é conhecida como *cliente gordo (fat-client)*.

### 1.3.3 Modelo multicamadas

A arquitetura de modelo multicamadas se consolida com a popularização da internet a partir da metade da década de 1990. Nela, as estações de trabalho se comunicam com o servidor de banco de dados através de uma camada intermediária, diferentemente do que ocorre no modelo cliente-servidor. Essa evolução arquitetônica tem como objetivo distribuir o processamento da aplicação a fim de evitar a sobrecarregá-lo de processamento sobre uma única



camada. Para uma aplicação ser considerada como multicamadas, é necessário que ela apresente ao menos três camadas específicas.

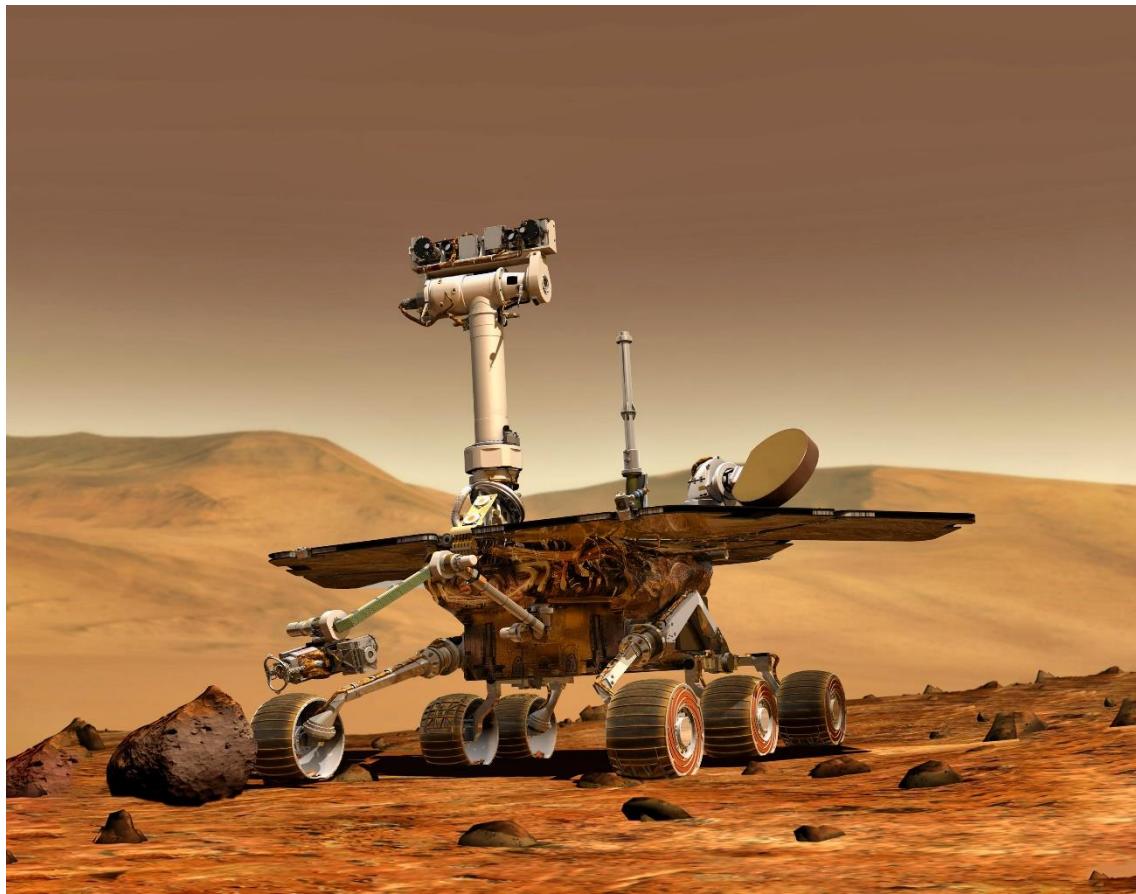
- **Camada de apresentação:** a camada de apresentação também é conhecida como Interface Gráfica com o Usuário (GUI – *Graphical User Interface*). Nela ocorre a interação entre o usuário e o software por meio de uma interface gráfica, que é acessada por meio de uma estação de trabalho. A camada de apresentação se comunica apenas com a camada de domínio da aplicação, na qual serão processadas todas as ações efetuadas pelo usuário através da interface do software.
- **Camada de domínio:** na camada de domínio da aplicação, estão implementadas todas as regras de negócio da aplicação. Ela se comunica tanto com a camada de apresentação, efetuando o processamento das requisições efetuadas pelo usuário, quanto com a camada de fonte de dados, realizando a comunicação com o banco de dados e/ou outros sistemas. A comunicação com as camadas de apresentação e de fonte de dados ocorre por meio do servidor Java EE.
- **Camada de fonte de dados:** a camada de fonte de dados é responsável pela troca de informações com outros sistemas, sendo, na maioria das vezes, responsável apenas pela comunicação da aplicação com o banco de dados. Nessa camada, encontramos os sistemas de informação corporativos (EIS – *Enterprise Information System*) que possuem a capacidade de trocar informações com outros sistemas, característica conhecida como *interoperabilidade*.

## TEMA 2 – PLATAFORMA JAVA EE

Atualmente, a linguagem Java é a plataforma de desenvolvimento mais utilizada no mundo e conta com mais de 9 milhões de desenvolvedores ao redor do globo. As aplicações desenvolvidas em Java estão presentes no nosso dia a dia, podemos encontrá-las executando em diversos dispositivos, como *smartphones*, *tablets*, *notebooks*, eletrodomésticos, televisores, além de jogos, *websites* e até mesmo em Marte.



Figura 2 – Robô *Spirit* controlado remotamente por uma aplicação Java



Crédito: Best-Backgrounds/Shutterstock.

A linguagem Java conta com quatro plataformas distintas, na qual cada plataforma provém um conjunto de especificações para o desenvolvimento de aplicações com propósitos bem definidos. Entre elas, iremos abordar a plataforma Java EE (*Enterprise Edition*), voltada para o desenvolvimento de aplicações corporativas de grande porte.

Conforme a documentação da Oracle, a plataforma Java EE fornece um poderoso conjunto de APIs a fim de reduzir o tempo e a complexidade de desenvolvimento do projeto, além de otimizar o desempenho da aplicação. Utilizando as especificações dessa plataforma, é possível desenvolver aplicações distribuídas, escaláveis, portáteis, transacionais que aproveitam a velocidade, a segurança e a confiabilidade da tecnologia do lado do servidor.

## 2.1 Arquitetura

A arquitetura de uma aplicação Java EE é baseada no modelo distribuído em multicamadas, composta por três camadas físicas e quatro camadas lógicas.



A **arquitetura física** é composta pelas seguintes camadas.

- Máquina do cliente (*Cliente Machine*): dispositivo pelo qual o usuário interage com a aplicação.
- Servidor Java EE (*Java EE Server*): máquina responsável por prover os serviços para a aplicação.
- Servidor de banco de dados (*Database Server*): máquina responsável pelo armazenamento de dados da aplicação.

A **arquitetura lógica** é composta pelas seguintes camadas.

- Camada do cliente (*Client Tier*): camada de apresentação da aplicação.
- Camada Web (*Tier Web*): camada responsável pelo processamento das requisições e das páginas Web.
- Camada de negócio (*Business Tier*): camada na qual estão implementadas as regras de negócio.
- Camada de sistemas de informação corporativos (*EIS Tier*): camada de fonte de dados da aplicação.

Além disso, uma aplicação Java EE é composta por componentes, unidade de software funcional independente que são executados dentro de uma aplicação Java EE. Entre eles, destacam-se:

- cliente de aplicativo (*Application client*) e miniaplicativo (*Applet*), componentes executados na camada do cliente;
- *servlets*, JavaServer Faces (JSF) e JavaServer Pages (JSP), componentes executados na camada Web do servidor Java EE; e
- enterprise JavaBeans (EJB), componente executado na camada de negócio do servidor Java EE.

O ciclo de vida de cada componente é gerenciado por um *container*. No ambiente de uma aplicação Java EE, deparamo-nos com os seguintes *containers*:

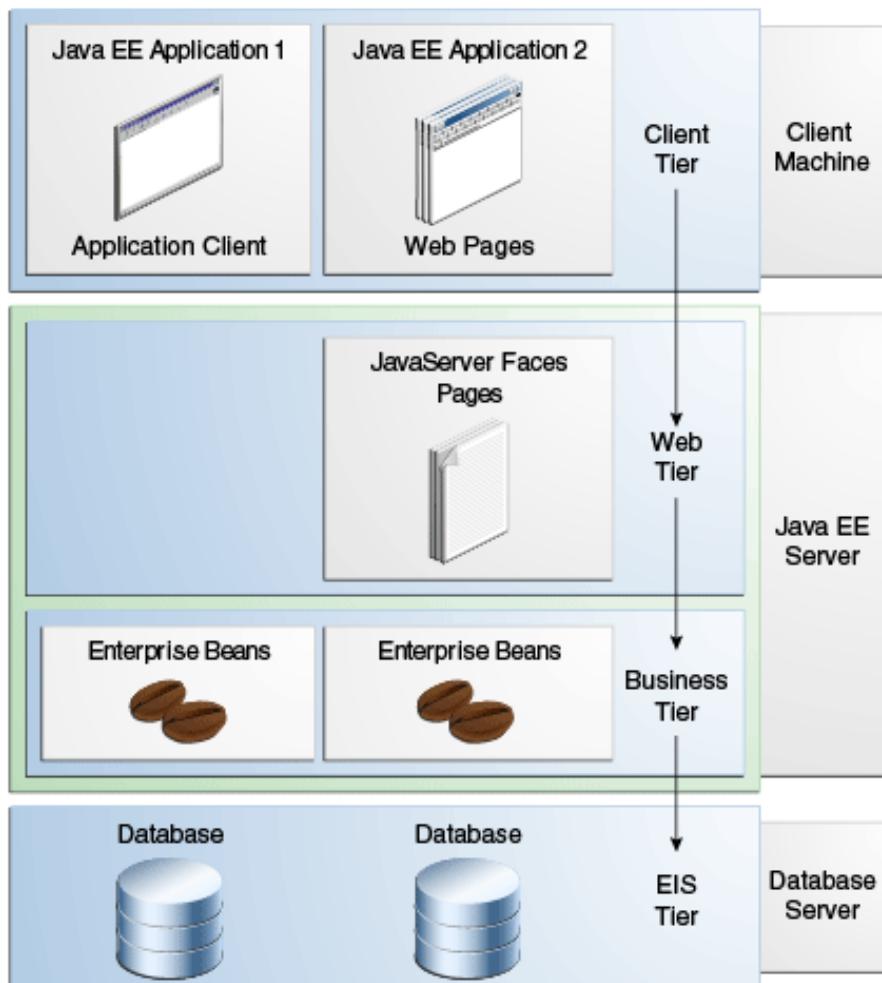
- *Applet Container* – gerencia o ciclo de vidas dos *applets*;
- *Application Client Container* – gerencia o ciclo de vida dos componentes do cliente de aplicativo;
- *Web Container* – gerencia o ciclo de vida dos componentes Web; e



- EJB Container – gerencia o ciclo de vida dos componentes de negócio.

A disposição das camadas e dos componentes de uma aplicação Java EE podem ser visualizados na figura a seguir, extraída da documentação oficial da Oracle.

Figura 3 – Arquitetura Java EE



Fonte: Oracle, [S.d.].

Nos próximos temas, serão abordados mais detalhadamente os elementos que compõem o ambiente de uma aplicação Java EE.

### TEMA 3 – AMBIENTE DA PLATAFORMA JAVA EE

Conforme mencionado anteriormente, a arquitetura da aplicação Java EE é distribuída em multicamadas, devendo conter ao menos três camadas lógicas: apresentação, domínio e fonte de dados. A seguir, serão abordados os elementos que compõem cada uma dessas camadas.



### 3.1 Camada de apresentação

A camada de apresentação da aplicação corresponde à camada do cliente (*Tier Client*), responsável por prover a interface gráfica da aplicação. A interface pode ser acessada por meio de uma página *Web* (*Web Page*) ou cliente de aplicativo (*Application Client*).

#### 3.1.1 Cliente *Web*

O cliente *Web* é composto por dois elementos.

- **Páginas *Web* dinâmicas (*Dynamic Web Pages*)**: contém a interface gráfica da aplicação fornecida pelos componentes de camada *Web* do servidor Java EE.
- **Navegador *Web* (*Web Browser*)**: responsável pelo processamento das páginas recebidas do servidor Java EE.

O cliente *Web* também é chamado de *cliente magro (thin client)*, pois, geralmente, não efetuam o processamento das regras de negócio da aplicação, delegando essa tarefa ao servidor da aplicação.

##### 3.1.1.1 Cliente de aplicativo

Um cliente de aplicativo (*Application Client*) é executado em uma máquina cliente que fornece ao usuário uma interface para que este possa realizar as suas tarefas, geralmente desenvolvida a partir da API *Swing* ou API *Abstract Window Toolkit* (AWT). Os clientes de aplicativo acessam diretamente os EJBs presentes na camada de negócios da aplicação, porém, se necessário, também podem estabelecer uma conexão HTTP com uma *servlet* que esteja sendo executada na camada *Web*. É importante destacar que os clientes de aplicativo desenvolvidos em outras linguagens de programação também podem interagir com uma aplicação Java EE.

##### 3.1.1.2 Applet

Criado para adicionar interatividade às páginas *Web* que antes eram totalmente estáticas, o *applet* (miniaplicativo) é um programa desenvolvido para executar uma tarefa específica dentro do contexto de outra aplicação. Como



exemplo de *applet*, podemos citar os *plugins*, programas que são instalados em um navegador com o objetivo de adicionar funcionalidades e recursos das páginas *Web*.

### 3.1.1.3 Camada de domínio

É responsável por prover os serviços necessários para o processamento de dados da aplicação. Esses serviços são disponibilizados pelo servidor Java EE (*Java EE Server*) através de um conjunto de componentes e APIs que contém a implementação das especificações da plataforma Java EE. Há dois tipos de servidores que implementam essas especificações dentro da plataforma Java EE.

- **Servidor de aplicação (*Application Server*)**: software que contém a implementação de todas as especificações da plataforma Java EE, sendo indicado para o desenvolvimento de aplicações de grande porte. Os servidores de aplicação destacam-se por fornecerem suporte para a criação de componentes *Web* e EBJ. Entre eles, destacam-se:
  - Apache Geronimo;
  - *GlassFish*;
  - JBoss;
  - JOracle *WebLogic*;
  - IBM *WebSphere*;
  - SAP *NetWeaver*; e
  - SAP *Web Application*.
- **Servlet container**: possui algumas limitações quando comparado ao servidor de aplicações, pois não fornece suporte para todas as especificações da plataforma Java EE, apenas para componentes *Web*, sendo uma ótima alternativa para o desenvolvimento de aplicações de pequeno e médio porte. Os *servlets containers* mais utilizados do mercado são:
  - Apache Tomcat; e
  - Jetty.

O domínio da aplicação está dividido em duas camadas com propósitos diferentes: camada *Web* e camada de negócio.



### 3.1.1.4 Camada Web

A camada *Web* é responsável por efetuar o processamento das requisições HTTP e das páginas Web. Nessa camada, encontram-se os seguintes componentes:

- **Servlets** – responsável por tratar as requisições HTTP solicitadas pelos clientes.
- **JavaServer Faces (JSF)** – tecnologia utilizada para desenvolvimento de páginas *Web* dinâmicas por meio de componentes pré-fabricados.
- **JavaServer Pages (JSP)** – tecnologia utilizada para o desenvolvimento das páginas *Web* dinâmicas por meio de *taglibs* ou *scriptlets*.

### 3.1.1.5 Camada de negócio

A camada de negócio é responsável pela implementação das regras de negócio da aplicação. Nessa camada, encontra-se o seguinte componente:

- **Enterprise JavaBean (EJB)** – tecnologia que permite o desenvolvimento rápido e simplificado de aplicações distribuídas, transacionais, seguras e portáteis.

## 3.1.2 Camada de sistemas de informação corporativos

*Camada de sistemas de informação corporativos* corresponde a camada de fonte de dados, responsável pela interação com servidores externos a aplicação, sendo, na maioria das vezes, composta apenas pelo servidor de banco de dados.

## 3.2 Containers

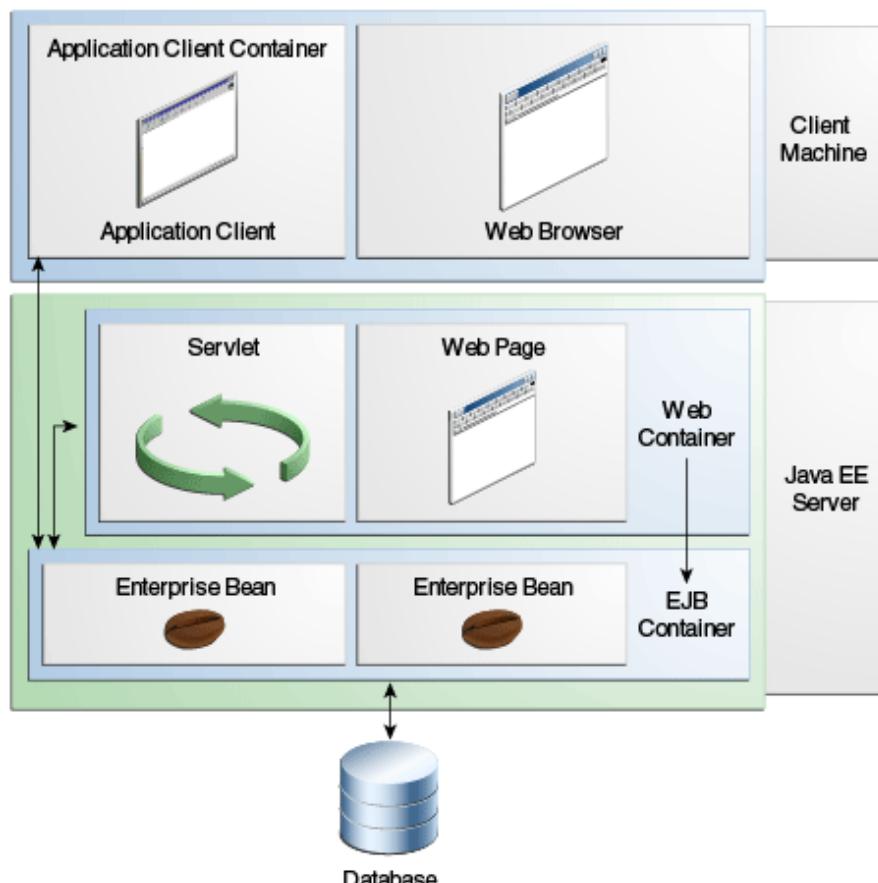
Os *containers* da plataforma Java EE são responsáveis por manter a comunicação entre componentes, APIs e demais serviços da aplicação, sendo eles gerenciados pelo software do fornecedor de servidor Java EE, que disponibilizam um container para cada tipo de componente. Os *containers* são divididos em quatro grupos.

- **Applet Container**: gerencia o ciclo de vidas dos *applets*.



- **Application Client Container:** gerencia o ciclo de vida dos componentes do cliente de aplicativo.
- **Web Container:** gerencia o ciclo de vida dos componentes Web.
- **EJB Container:** gerencia o ciclo de vida dos componentes de negócio.

Figura 4 – Componentes da plataforma Java EE



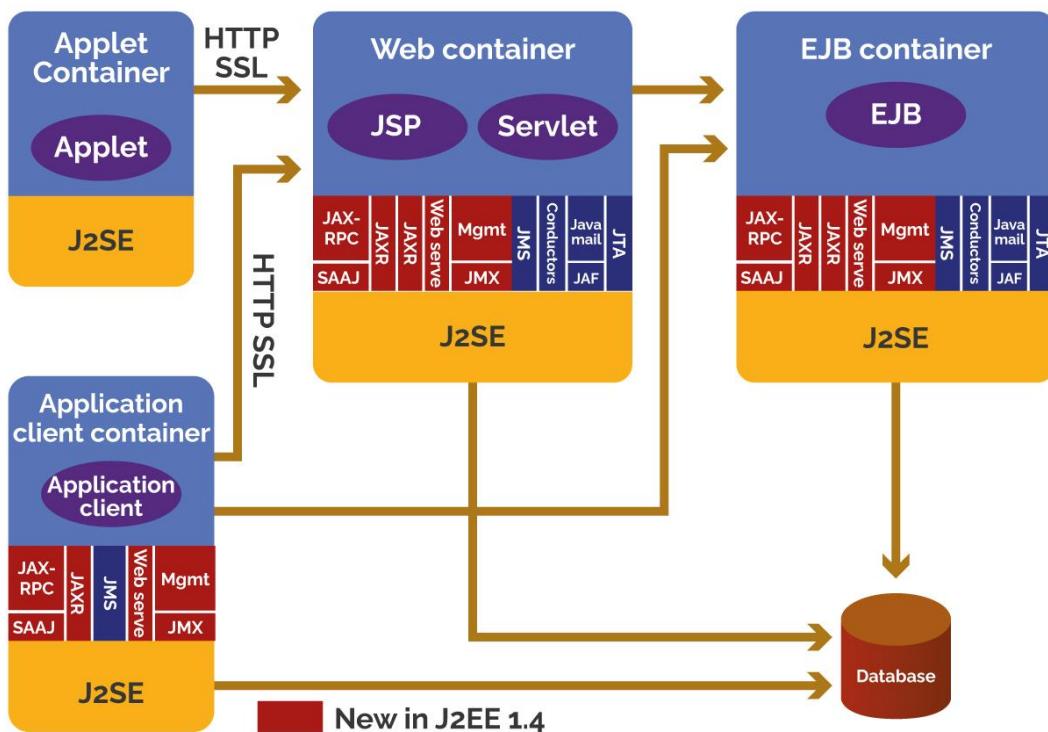
Fonte: Oracle, [S.d.].

Os *containers* fornecem diversos serviços para os componentes. Serão listados alguns deles:

- persistência de dados (JDBC – *Java Database Connection*);
- controle transacional (JTA – *Java Transaction API*);
- segurança (JAAS – *Java Authentication and Authorization Service*);
- envio de notificações por e-mail (*JavaMail*); e
- conectividade (RMI – *Remote Method Invocation*).



Figura 5 – Containers da plataforma Java EE



## TEMA 4 – COMPONENTES DA PLATAFORMA JAVA EE

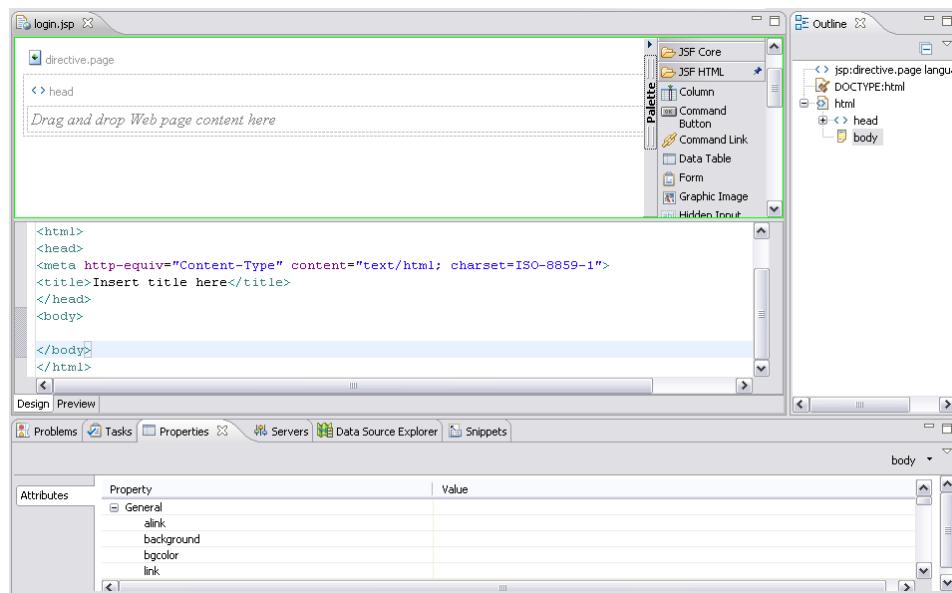
Neste tema, serão abordados os componentes da plataforma Java EE e suas funcionalidades.

### 4.1 JavaServer Faces

A JSF (*JavaServer Faces*) é um componente *Web* da plataforma Java EE utilizado para o desenvolvimento da interface gráfica da aplicação. Essa tecnologia permite criar uma página *Web* de forma rápida e prática, pois fornece um conjunto de componentes pré-fabricados que podem ser adicionados a um formulário. Cada componente possui um conjunto de eventos que permitem ao desenvolvedor saber qual ação foi executada pelo usuário e implementar a rotina pertinente ao evento executado.



Figura 6 – Interface do Eclipse para criação de páginas JSF



Na figura anterior, temos a interface do Eclipse para a criação de páginas JSF. Nessa tela, o desenvolvedor arrasta os componentes pré-fabricados como botões, tabelas, links, imagens, entre outros componentes para o corpo da página, criando, assim, de forma rápida, a interface de uma página Web.

## 4.2 JavaServer Pages

A JSP (*Java Server Pages*) é outro componente *Web* da plataforma Java EE utilizado para o desenvolvimento da interface gráfica da página *Web*. Essa tecnologia permite desenvolver páginas dinâmicas, embutindo código Java dentro do código HTML por meio dos *scriptlets* (bloco de comandos compreendidos entre as tags “`<%`” e “`%>`”) ou das *taglibs* da biblioteca JSTL. No quadro a seguir, é exibido o código de uma página JSP utilizando *scriptlets* para exibir uma informação na tela.

Quadro 1 – Código fonte de uma JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
    <title>Universidade</title>
</head>
<body>
    <h1>
        <% out.println("Olá Mundo!"); %>
    </h1>
</body>
</html>
```



Diferentemente da especificação JSF, a JavaServer Pages requer do desenvolvedor um conhecimento mais apurado para o desenvolvimento da interface da página Web, uma vez que ele terá que implementar e estilizar os seus próprios componentes. Para isso, ele deve conhecer as três linguagens listadas a seguir.

- **HTML**: linguagem de marcação utilizada para inserir conteúdo e definir a estrutura da página Web.
- **JavaScript**: linguagem de script utilizada para implementar os eventos dos componentes HTML.
- **CSS**: linguagem de estilo utilizada para alterar os elementos visuais dos componentes HTML.

### 4.3 *Servlets*

A *servlet* é um componente da plataforma Java EE que permite ao desenvolvedor efetuar o processamento das operações realizadas pelo usuário dentro da aplicação. Para cada operação efetuada no cliente, será enviada ao servidor uma requisição HTTP com as informações necessárias para o seu processamento. Para isso, o desenvolvedor deverá implementar uma *servlet* para atender tal requisição e retornar uma resposta para o cliente, com as informações referentes ao seu processamento.

Baseado nas informações contidas na resposta do servidor, é possível identificar se a requisição HTTP foi processada com sucesso ou se ocorreu algum erro durante o seu processamento. Com isso, a aplicação consegue dar um *feedback* adequado para o usuário a respeito da ação efetuada. Uma *servlet* deve implementar a interface HttpServlet e tanto a requisição como a resposta HTTP pode ser acessada através dos objetos das classes HttpServletRequest e HttpServletResponse, respectivamente. No quadro a seguir, é exibido o código de uma *servlet*.



## Quadro 2 – Código fonte de uma servlet

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {
    private static final long serialVersionUID = 1L;

    // Requisição HTTP método GET
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Bloco de comandos
    }

    // Requisição HTTP método POST
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Bloco de comandos
    }
}
```

## 4.4 Enterprise Java Beans

O Enterprise Java Beans (EJB) é um componente da camada de negócio que permite o desenvolvimento rápido e simplificado de aplicações distribuídas, transacionais, seguras e portáteis dentro da plataforma Java EE. Os EJBs estão divididos em duas categorias: *Session Beans* e *Message-driven Beans*. A configuração dos EJB pode ser realizada por meio de anotações (*Annotations*) ou por meio do arquivo web.xml, conhecido como *Deployment Descriptor*.

### 4.4.1 Message-Driven Bean

O *bean* do tipo *Message-Driven* (MDB) permite que as aplicações Java EE processem mensagens de forma assíncrona. Esse tipo de *bean* pode implementar qualquer tipo de mensagem, sendo mais comum o processamento de mensagens do tipo JMS (Java Message Service). Um *Message-Driven Bean* implementa a *interface* *MessageListener* e é identificado pela anotação `@MessageDriven`, na qual deve ser especificado o JNDI do JMS destino, por meio do atributo `mappedName`. Sempre que esse *bean* for invocado, será executado o método `onMessage`, que contém a lógica de negócios a ser processada pela aplicação.



### Quadro 3 – Código fonte de um *Message-Driven Bean*

```
@MessageDriven(mappedName = "destino")
public class MeuMessageBean implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            // processamento da mensagem
        } catch (JMSException ex) {
            // tratamento das exceções
        }
    }
}
```

#### 4.4.2 Session Beans

Os *Session Beans* implementam as regras de negócio da aplicação. Existem três tipos de *Session Beans* que são classificados de acordo com o seu ciclo de vida: *Stateless*, *Stateful* e *Singleton*.



#### 4.4.2.1 Stateless Session Bean

O *bean* sem estado de sessão (*Stateless Session Bean*) possui um ciclo de vida correspondente ao tempo de execução do método invocado. Esse tipo de *bean* é utilizado na implementação de classes cujos métodos não exigem o armazenamento de dados entre uma requisição e outra. Isso significa dizer que o *Stateless Session Bean* não mantém um estado conversacional com o cliente. As classes sem estado de sessão são identificadas com a anotação @Stateless, conforme podemos verificar no quadro a seguir.

Quadro 4 – Código fonte de um *Stateless Session Bean*

```
@Stateless
public class Calculadora {
    public float adicao(float operador1, float operador2) {
        return operador1 + operador2;
    }

    public float subtracao(float operador1, float operador2) {
        return operador1 - operador2;
    }

    public float multiplicacao(float operador1, float operador2) {
        return operador1 * operador2;
    }

    public float divisao(float operador1, float operador2) {
        return operador1 / operador2;
    }
}
```

#### 4.4.2.2 Stateful Session Bean

Diferentemente do *Stateless Session Bean*, o *bean* com estado de sessão (*Stateless Session Bean*) mantém o estado conversacional com o cliente. É importante destacar que um *bean* do tipo *Stateful* não pode ser compartilhado entre clientes. Um exemplo clássico de *Stateful Session Bean* é o carrinho de compras de um *e-commerce*. Ao adicionar um produto ao carrinho de compras, a aplicação deve manter o seu estado para que o usuário possa continuar navegando pelo site e adicionar novos produtos ao carrinho de compras. As classes com estado de sessão são identificadas com a anotação @Stateful, conforme podemos verificar no quadro a seguir.



Quadro 5 – Código fonte de um *Stateful Session Bean*

```
@Stateful
public class CarrinhoDeCompras {

    List<Produto> produtos;

    public CarrinhoDeCompras () {
        produtos = new ArrayList<Produto>();
    }

    public void adicionaProduto(Produto produto) {
        produtos.add(produto);
    }

    public void removeProduto(Produto produto) {
        produtos.remove(produto);
    }

    @Remove
    public void finalizaCompra() {
        produtos = null;
    }
}
```

#### 4.4.2.3 *Singleton Session Bean*

O *bean* do tipo *Singleton* mantém o estado de conversação de um *bean* durante o tempo de execução da aplicação. Esse *Session Bean* é muito similar ao *bean* do tipo *Stateful*, com a diferença que o *bean* do tipo *Singleton* possui apenas uma única instância, a qual pode ser acessada simultaneamente por diversos clientes. No quadro a seguir, temos um EJB do tipo *Singleton* que nos permite identificar a quantidade de acessos realizados na aplicação.

Quadro 6 – Código-fonte de um *Singleton Session Bean*

```
@Singleton
public class ContadorDeAcessos {
    int contador;

    public void incrementar() {
        ++contador;
    }

    public void getAcessos() {
        return contador;
    }

    public void reiniciar() {
        contador = 0;
    }
}
```



## TEMA 5 – SERVIÇOS E BIBLIOTECAS

Neste tema, serão abordadas as principais especificações e bibliotecas da plataforma Java EE.

### 5.1. JavaServer Pages Standard Tag Lib

A JSTL (*JavaServer Pages Standard Tag Lib*) é uma biblioteca composta por um conjunto de *tags* que podem ser utilizadas para o desenvolvimento da JSP como uma alternativa ao *scriptlet*, tornando o código da página mais legível e enxuto. A JSTL é composta por cinco pacotes, conforme ilustra a tabela a seguir.

Tabela 1 – Pacotes da JSTL

Pacote	Prefixo	Funcionalidade
<b>JSTL Core</b>	c	Comandos condicionais, iterativos e de atribuição
<b>JSTL fmt</b>	fmt	Formatação de dados
<b>JSTL sql</b>	sql	Persistência de dados
<b>JSTL xml</b>	xml	Manipulação e criação de documentos XML
<b>JSTL functions</b>	Fn	Processamento de <i>strings</i> e coleção de dados

Cada pacote JSTL conta com um conjunto de *tags* que estão agrupadas de acordo com as suas funcionalidades. Na tabela a seguir, elencaremos algumas das tags que compõem o pacote JSTL Core e suas respectivas funções.

Tabela 2 – Tags do pacote JSTL Core

Tag	Função
<b>choose, when</b> <b>e otherwise</b>	Utilizadas para analisar várias alternativas para uma condição específica, formam uma estrutura semelhante ao comando <i>switch</i> em Java.
<b>if</b>	Utilizada para exibir um determinado conteúdo na JSP caso a condição analisada seja verdadeira.
<b>forEach</b>	Utilizada para realizar a iteração de uma coleção.
<b>out</b>	Utilizada para exibir uma informação na tela.
<b>set</b>	Utilizada para atribuir um valor a uma variável.



Algumas *tags* exigem o preenchimento de atributos, espécie de variáveis internas da *tag*, para que elas possam ser executadas, como no caso da tag *out*. Na tabela a seguir, serão listados os atributos da *tag out*.

Tabela 3 – Atributos da *tag out*

Atributo	Descrição	Obrigatório
<b>value</b>	Utilizado para exibir uma informação.	Sim
<b>default</b>	Utilizado para exibir uma informação quando o valor do atributo <i>value</i> for igual a nulo.	Não
<b>escapeXML</b>	Utilizado para verificar se os caracteres especiais '&', '<' e '>' do atributo <i>value</i> devem ser convertidos em suas respectivas entidades HTML. O valor padrão desse atributo é verdadeiro.	Não

As *tags* JSTL são identificadas dentro de uma JSP pelo prefixo do pacote seguido de dois pontos mais a tag do pacote. No código a seguir, foi utilizada a *tag out* do pacote JSTL Core para exibir uma informação na tela. A informação a ser exibida foi informada no atributo *value* da *tag out*, como podemos observar no quadro a seguir.

Quadro 7 – JSP com *taglib*

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
    <head>
        <title>Universidade</title>
    </head>
    <body>
        <c:out value="Olá mundo com JSTL!" />
    </body>
</html>
```

## 5.2 Expression Language

A *Expression Language* (EL) permite ao desenvolvedor acessar dinamicamente os dados dos componentes Java Beans e criar expressões lógicas e aritméticas. Além disso, através da EL, é possível acessar diversas informações da página como cookies, cabeçalhos, parâmetros, variáveis de



escopo e diversos outros atributos por meio dos chamados objetos implícitos. A EL possui a seguinte sintaxe:

#### Quadro 8 – Sintaxe de uma EL

```
${ expressão }
```

No quadro a seguir, temos o código de uma JSP com EL que exibe uma mensagem de boas-vindas para o usuário logado no sistema.

#### Quadro 9 – Código JSP com EL

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <title>Universidade</title>
    </head>
    <body>
        <h1>Seja bem-vindo ${usuario.nome}</h1>
    </body>
</html>
```

### 5.3 Java Persistence API

A JPA (Java Persistence API) é um serviço para realizar a persistência de dados da aplicação através do Mapeamento Objeto-Relacional (ORM – *Object-Relational Mapping*). Para isso, devemos utilizar as anotações do pacote javax.persistence.

No Quadro 9, temos um exemplo do Mapeamento Objeto-Relacional por meio do JPA, na qual temos as seguintes anotações.

- **@Entity**: especifica que a classe é uma entidade, portanto, os dados dessa classe serão armazenados em uma determinada tabela no banco de dados. Caso o nome da classe seja diferente do nome da tabela, basta informar no atributo *name* dessa anotação o nome da respectiva tabela.
- **@Id**: especifica quais são os atributos que compõem a chave primária da entidade.
- **@GeneratedValue**: especifica como será gerado o valor do atributo-chave da classe. Neste caso, foi adotada estratégia GenerationType.AUTO, na



qual o provedor de persistência criará o identificador de forma automática de acordo com o SGBD adotado.

- `@Column`: especifica que o dado do atributo será armazenado em uma determinada coluna da tabela. Caso o nome do atributo da classe seja diferente do nome da coluna, basta informar no atributo `name` dessa anotação o nome da respectiva coluna.
- `@Temporal`: especifica que o atributo é do tipo temporal: data, hora ou timestamp (data e hora).

Na documentação da JPA, você encontra mais informações a respeito das anotações e seus respectivos atributos.

Quadro 10 – Código fonte de uma classe com JPA

```
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Pessoa {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false)
    private String nome;
    private String cpf;
    @Temporal(TemporalType.DATE)
    @Column(name = "dataNasc", nullable = false)
    private Date dataNascimento;
    @Column

    // declarar os getters e setters
}
```

## 5.4 JavaMail

As aplicações Java EE utilizam a API JavaMail para o envio de notificações por e-mail. No Quadro 11, é exibida a classe JavaMail que implementa esse serviço, adotando como exemplo uma conta do Gmail. Para que essa classe possa funcionar corretamente, deve-se definir o conteúdo das seguintes variáveis.



- *E-mail*: informar o *e-mail* que fará o envio das notificações.
- Senha: informar a senha do *e-mail* que fará o envio das notificações.

Quadro 11 – Código-fonte de uma classe com JavaMail

```
import java.util.Properties;
import javax.mail.Address;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class ServicoDeEmail {
    public static void enviaNotificacao(String destinatario, String assunto, String conteudo) {
        Properties props = new Properties();

        // Dados do remetente
        String email = "", senha = "";

        // Configuração do servidor
        props.put("mail.smtp.host", "smtp.gmail.com");
        props.put("mail.smtp.socketFactory.port", "465");
        props.put("mail.smtp.socketFactory.class",
        "javax.net.ssl.SSLSocketFactory");
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.port", "465");

        // Autenticação
        Session session = Session.getDefaultInstance(props,
            new javax.mail.Authenticator() {
                protected PasswordAuthentication getPasswordAuthentication(){
                    return new PasswordAuthentication(email, senha);
                }
            });
        try {
            // Envio da notificação por e-mail
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress(email));
            Address[] toUser = InternetAddress.parse(destinatario);

            message.setRecipients(Message.RecipientType.TO, toUser);
            message.setSubject(assunto);
            message.setText(conteudo);

            Transport.send(message);
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}
```



## 5.5 Java Beans

Java *Beans* é uma classe extremamente simples que tem como objetivo encapsular e abstrair uma entidade do sistema. Para que uma classe seja considerada um *bean*, ela deve atender aos seguintes requisitos:

- implementar a interface *Serializable*;
- possuir apenas o construtor padrão; e
- seus atributos são acessados apenas pelos métodos *get* e *set*.

No quadro a seguir, temos um exemplo de uma classe do tipo Java *Bean*.

Quadro 12 – Código fonte de um Java Bean

```
import java.util.Date;

public class Pessoa implements java.io.Serializable {
    private String cpf;
    private String nome;
    private Date dataDeNascimento;

    public Pessoa() {
        this.nome = null;
        this.cpf = null;
        this.date = null;
    }

    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public Date getDataDeNascimento() {
        return dataDeNascimento;
    }
    public void setDataDeNascimento(Date dataDeNascimento) {
        this.dataDeNascimento = dataDeNascimento;
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

## 5.6 Java Transaction API

A JTA (Java Transaction API) fornece uma interface padrão de alto nível para demarcação de transações por meio do pacote javax.transaction. Uma



transação consiste em uma sequência de operações que são executadas em conjunto, como se fosse uma única unidade lógica. Ela segue o conceito ACID, acrônimo para as quatro propriedades de uma transação: atomicidade, consistência, isolamento e durabilidade. A transação pode ser especificada tanto em nível de classe quanto em nível de método por meio da anotação `@Transactional`.

Quadro 13 – Transação em nível de classe

```
@Transactional  
public class Bean {  
    // Definição da classe  
}
```

Quadro 14 – Transação em nível de método

```
public class Bean {  
    public void metodo1() {  
        // Bloco de comandos  
    }  
  
    @Transactional  
    public void metodo2() {  
        // Bloco de comandos  
    }  
}
```

## 5.7 Java Database Connectivity API

A JDBC (Java Database Connectivity) é uma API que fornece um conjunto de classes e interfaces para que a aplicação possa realizar a persistência de dados através de um *driver* específico para o SGBD desejado, devendo este ser importado ao projeto.



## Quadro 15 – Classe de conexão com JDBC

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {
    public static Connection createConnection() throws SQLException{
        String connectionString = "jdbc:mysql://localhost:3306/[catalogo]";
        String usuario = "";
        String senha = "";

        Connection conexao = null;
        conexao = DriverManager.getConnection(connectionString, usuario,
        senha);

        return conexao;
    }
}
```

Adotando o MySQL como SGBD da aplicação, no Quadro 12, temos a classe que estabelece a conexão com o banco de dados, devendo-se definir as seguintes variáveis.

- *ConnectionString*: substituir o termo [catalogo] pelo nome da base de dados que se deseja acessar.
- Usuário: informar o usuário de acesso ao banco de dados.
- Usuário: informar a senha do usuário de acesso ao banco de dados.

Para executar os comandos DML (*Data Manipulation Language*), como *insert*, *update*, *delete* e *select*, iremos utilizar o objeto *PreparedStatement* e para obter o resultado da consulta a interface *ResultSet*. No Quadro 16, temos o exemplo de um código referente ao processo de inserção de dados. Para isso, deve-se instanciar o objeto *PreparedStatement* por meio do método *prepareStatement* do objeto de conexão, passando por parâmetro o comando SQL a ser executado. Para executar o comando SQL, basta invocar o método *execute* do objeto instanciado anteriormente.



#### Quadro 16 – Código-fonte referente à inserção de dados

```
String query = "INSERT INTO pessoa ("  
    + "    nome,"  
    + "    cpf,"  
    + "    dataNascimento"  
    + ") VALUES ("  
    + "    'Rafael','"  
    + "    '99988877766','"  
    + "    '2021-10-04'"  
    + ")";  
  
PreparedStatement ps = conexao.prepareStatement(query);  
ps.execute();
```

Para efetuar a consulta, também deve-se instanciar o objeto `PreparedStatement`, porém, dessa vez, será invocado o método `executeQuery`, que irá retornar os registros da consulta em variável `ResultSet`, conforme podemos visualizar no Quadro 17.



Quadro 17 – Código-fonte referente à consulta de dados

```
conexao = ConnectionFactory.createConnection();

String sql = "SELECT id, nome FROM pessoa";
PreparedStatement ps = conexao.prepareStatement(sql);

ResultSet rs = ps.executeQuery();
while(rs.next()){
    int codigo = rs.getInt("id");
    String nome = rs.getString("nome");
    System.out.printf("Código %d: %s\n",codigo, nome);
}
```

Os registros da consulta podem ser acessados por meio de um laço de repetição utilizando o método *next* do objeto *ResultSet*.

## FINALIZANDO

Nesta aula, abordamos a plataforma Java EE voltada para o desenvolvimento de aplicações corporativas de grande porte. Ela fornece uma série de APIs que facilitam o processo de desenvolvimento de uma aplicação, de modo que o desenvolvedor não precise se preocupar em desenvolver os requisitos não funcionais, focando apenas na implementação dos requisitos funcionais.

Além disso, foram abordados todos os elementos que compõem as diferentes camadas do ambiente de uma aplicação Java EE, enfatizando, principalmente, a camada do servidor da aplicação, tendo em vista que iremos implementar e utilizar algumas das especificações que são executadas nessa camada.

Futuramente, iremos abordar o *Spring Framework*, objeto de estudo desse curso, abordando as suas especificações e traçando comparativos com as especificações da Java EE, justificando o porquê de ele ser um dos *frameworks* de desenvolvimento Java mais utilizados do mercado.



## REFERÊNCIAS

CNES. **Java Runs remote-controlled Mars rover**: Java runs remote-controlled Mars rover – CNET. set. 2021.

ECLIPSE. **WTP Tutorials – JavaServer Faces Tools Tutorial**: WTP Tutorials – JavaServer Faces Tools Tutorial. Disponível em: <[eclipse.org](http://eclipse.org)>. Acesso em: 17 dez. 2021.

ORACLE. **Java Platform, Micro Edition (Java ME)**. 2021. Disponível em: <<https://www.oracle.com/java/technologies/javameoverview.html>>. Acesso em: 17 dez. 2021.

ORACLE. **Introduction to Java EE**. 2021. Disponível em: <<https://javaee.github.io/tutorial/overview001.html>>. Acesso em: 17 dez. 2021.

ORACLE. **Obtenha informações sobre a Tecnologia Java**. Disponível em: <<https://www.java.com/pt-BR/about/>>. Acesso em: 17 dez. 2021.

ORACLE. **Onde obter informações técnicas sobre o Java**. 2021. Disponível em: <[https://www.java.com/pt-BR/download/help/techinfo\\_pt-br.html](https://www.java.com/pt-BR/download/help/techinfo_pt-br.html)>. Acesso em: 17 dez. 2021.

ORACLE. **Java™ EE 8 Specification APIs**. 2021. Disponível em: <<https://javaee.github.io/javaee-spec/javadocs/>>. Acesso em: 17 dez. 2021.

ORACLE. **JavaServer Pages Standard Tag Library 1.1 Tag Reference**: Overview (TLDDoc Generated Documentation). Disponível em: <[oracle.com](http://oracle.com)>. Acesso em: 17 dez. 2021.

ORACLE. **Overview of the EL**: overview of the EL. Disponível em: <[javaee.github.io](https://javaee.github.io)>. Acesso em: 17 dez. 2021.

ORACLE. **Enterprise JavaBeans (EJBs)**: enterprise JavaBeans (EJBs). Disponível em: <[oracle.com](http://oracle.com)>. Acesso em: 17 dez. 2021.

ORACLE. **J2EE 1.4 APIs**: J2EE 1.4 APIs. Disponível em: <[oracle.com](http://oracle.com)>. Acesso em: 17 dez. 2021.



# DESENVOLVIMENTO WEB

## BACK END

AULA 2

Prof. Rafael Veiga de Moraes



## CONVERSA INICIAL

O *Spring* é indiscutivelmente o *framework* de desenvolvimento Java mais utilizado no mercado para o desenvolvimento de aplicações corporativas. Ele é composto por um conjunto de projetos que fornecem uma variedade de serviços. Esse *framework* surgiu como uma alternativa a plataforma Java EE no começo dos anos 2000, na época, denominado J2EE devido aos gargalos de desempenho das aplicações por conta da utilização dos EJBs e a sua complexa configuração.

Atualmente, ele fornece uma solução completa para o desenvolvimento de aplicações corporativas, estando presente no nosso dia a dia, seja em aplicações de *streaming*, compras on-line e inúmeras outras soluções inovadoras. Além disso, o *Spring* conta com um conjunto abrangente de extensões e bibliotecas de terceiros que permitem aos desenvolvedores criarem diferentes tipos de aplicações.

Nesta aula, iremos abordar as principais características do *Spring framework* e iniciar o processo de configuração do ambiente de desenvolvimento. Para isso, serão apresentados os softwares necessários para a construção da aplicação e abordados conceitos referentes à arquitetura de uma aplicação *web*, para, futuramente, darmos início ao processo de desenvolvimento.

## TEMA 1 – INTRODUÇÃO AO SPRING FRAMEWORK

### 1.1 História

Em outubro de 2002, Rod Johnson escreveu o livro *Expert One-on-One J2EE Design and Development*, pela editora Wrox, no qual ele elencou diversas deficiências na estrutura de componentes da plataforma Java EE. Na época, foi denominada J2EE, o que tornavam o processo de desenvolvimento de aplicações corporativas um tanto complexo.

Nesse livro, Rod Johnson propõe uma solução mais simples que se baseia na utilização de classes do tipo POJO (*Plain Old Java Objects*) e a Inversão de Controle (*IoC – Inversion of Control*) por meio da Injeção de Dependência (*DI – Dependency Injection*). Utilizando esses conceitos, Rod Johnson demonstrou em seu livro como implementar uma aplicação de reserva



de assentos on-line escalonável e de alta *performance* sem a utilização dos EJBs. O livro fez um enorme sucesso, tanto que, após muitos anos de seu lançamento, ainda apresenta conceitos relevantes no desenvolvimento de aplicações para os dias atuais.

Como boa parte do código-fonte da aplicação era reutilizável, muitos desenvolvedores começaram a utilizá-lo em suas próprias aplicações. Devido a essa enorme aceitação, os desenvolvedores Juergen Hoeller e Yann Caroff entraram em contato com Rod Johnson para que juntos desenvolvessem um projeto *open-source* baseado nos conceitos abordados no livro. Em 2003, deram início ao projeto *Spring*, do inglês *Spring*, que significa “primavera”, nome proposto por Yann, representando um novo começo após o “inverno” da plataforma J2EE.

Em 2003, foi lançada a primeira versão do *Spring Framework*, que foi rapidamente adotada por diversos desenvolvedores. No ano seguinte, Rod Johnson, Juergen Hoeller, Keith Donald e Colin Sampaleanu fundaram a empresa Interface21, cujo foco estava voltado para a consultoria, treinamento e suporte do *Spring*. Desde o lançamento da primeira versão, o *framework* continuou evoluindo e adquirindo novos adeptos, tanto que, em 2006, com a versão 2.0, o *Spring* ultrapassou a marca de 1 milhão de *downloads*.

No ano de 2007, a empresa Interface21 foi rebatizada para *SpringSource*, que, dois anos mais tarde, foi adquirida pela VMWare por 420 milhões de dólares. Atualmente, o *Spring Framework* é gerenciado pela empresa Pivotal, uma *joint venture* criada pelas empresas VMWare e EMC a partir de um investimento da General Electric.

## 1.2 Características

Conforme vimos anteriormente, o *Spring* se destacou no mercado por prover uma solução mais simples que a plataforma J2EE, por meio da injeção de dependência e da utilização de classes POJO em vez dos EJBs. Aqui, abordaremos essas duas características do *Spring Framework*.

### 1.2.1 Injeção de dependência

A inversão de controle é um padrão de desenvolvimento muito utilizado em projetos orientados a objeto, visando diminuir o acoplamento entre as classes



da aplicação. Ela consiste basicamente em tirar a responsabilidade de uma classe de instanciar objetos de outras classes das quais ela depende, tendo em vista que as funcionalidades dessas outras classes podem posteriormente vir a ser alteradas.

Isso traz inúmeros benefícios na manutenibilidade do projeto, pois deixa o código-fonte mais legível, facilitando a sua interpretação e melhora a distribuição das responsabilidades das classes que compõem a aplicação. Para entendermos esse padrão de desenvolvimento, vamos analisar a classe *Aplicacao* apresentada no quadro a seguir.

Quadro 1 – Exemplo de forte acoplamento sem IoC

```
public class Aplicacao {  
    private Tarefa tarefa;  
  
    public void executaTarefa() {  
        this.tarefa = new Tarefa();  
        tarefa.executa();  
    }  
}
```

Note que a classe *Aplicacao* possui um vínculo direto com a classe *Tarefa*, o que caracteriza um forte acoplamento, visto que o atributo *tarefa* é instaciado dentro do método *executaTarefa* da classe *Aplicacao*. De acordo com as boas práticas de programação orientada a objetos especificados no princípio SOLID, o forte acoplamento entre classes fere a regra do princípio da responsabilidade única (SRP – *Single Responsibility Principle*).

Uma das soluções possíveis para eliminar o forte acoplamento entre as classes seria, em vez de instanciar o objeto dentro do método *executaTarefa*, criar um *setter* para atribuir um objeto já instanciado ao atributo *tarefa*, tirando essa responsabilidade da classe *Aplicacao*. A solução para isso pode ser observada no Quadro 2, no qual foi inserido o método *setTarefa* de forma a eliminar o forte acoplamento entre as classes.

Quadro 2 – Exemplo de baixo acoplamento com IoC

```
public class Aplicacao {  
    private Tarefa tarefa;  
  
    public setTarefa(Tarefa tarefa) {  
        this.tarefa = tarefa;  
    }  
  
    public void executaTarefa () {  
        tarefa.executa();  
    }  
}
```



O *Spring Framework* se destacou no mercado justamente por automatizar o processo de inversão de controle por meio da injeção de dependência, dessa forma, o desenvolvedor não precisa se preocupar em instanciar um determinado objeto, ele pode deixar essa atribuição a encargo do próprio *framework*. No quadro a seguir, podemos visualizar a injeção de dependência para a classe *Aplicacao* por meio da anotação `@Autowired`.

Quadro 3 – Exemplo de baixo acoplamento com DI

```
public class Aplicacao {  
    @Autowired  
    private Tarefa tarefa;  
  
    public void executaTarefa () {  
        tarefa.executa();  
    }  
}
```

Dessa forma, o desenvolvedor não precisa se preocupar em gerenciar o ciclo de vida de uma classe, o próprio *Spring*, por meio do seu contêiner de IoC, faz isso de forma automática, sabendo o exato momento de quando instanciar e destruir o objeto.

### 1.2.2 Classes POJO

Uma classe do tipo POJO se caracteriza por conter um construtor padrão e possuir apenas os métodos de *getter* e *setter* para acessar os seus atributos. No quadro a seguir, temos um exemplo de uma classe do tipo POJO.

Quadro 4 – Exemplo de uma classe do tipo POJO

```
public class Pessoa {  
    private long id;  
    private String nome;  
    private String cpf;  
  
    public Pessoa() {  
        this.id = 0;  
        this.nome = "";  
        this.cpf = "";  
    }  
  
    public long getId() {  
        return id;  
    }  
  
    public void setId(long id) {  
        this.id = id;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}
```



Um dos grandes problemas da plataforma Java EE no começo dos anos 2000 era o uso dos EJBs, que, embora provessem uma solução robusta, a sua configuração era complexa e exigia muito processamento, tornando-se o principal gargalo de desempenho da aplicação.

Diante disso, o *Spring* surgiu com uma ótima alternativa a plataforma Java EE, já que também provia os recursos necessários para o desenvolvimento de aplicações corporativas Java, porém com um melhor desempenho e uma configuração mais simples, visto que, em vez dos EJBs, foram adotadas classes do tipo POJO para a implementação das regras de negócio da aplicação.

## TEMA 2 – CONFIGURANDO O AMBIENTE DE DESENVOLVIMENTO

Antes de codificarmos a nossa aplicação, precisamos previamente baixar e instalar as ferramentas de desenvolvimento pertinentes ao projeto que será desenvolvido. Para isso, iremos listar os softwares que você deverá instalar na sua máquina para que possamos dar sequência ao curso.

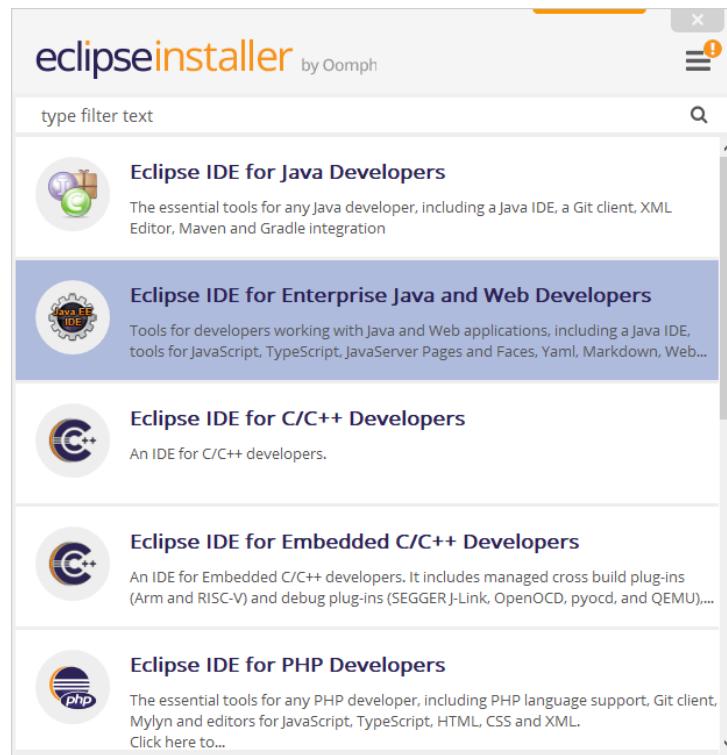
### 2.1 IDE

O primeiro software a ser instalado é o ambiente integrado de desenvolvimento (IDE – *Integrated Development Environment*), ferramenta de desenvolvimento que fornece funcionalidades que facilitam o processo de codificação e de gerenciamento dos recursos da aplicação. O *Spring Framework* é compatível com diversas IDEs, porém adotaremos o Eclipse como software de desenvolvimento da aplicação a ser desenvolvida, visto que é uma IDE poderosa, *open-source* e largamente utilizada no mercado.

O Eclipse pode ser baixado no site. Ao executar o instalador, serão listadas diversas versões do Eclipse, na qual o usuário deverá selecionar uma delas para dar sequência a instalação. Selecione a opção *Eclipse IDE for Enterprise Java and Web Developers* conforme mostrado na figura a seguir.

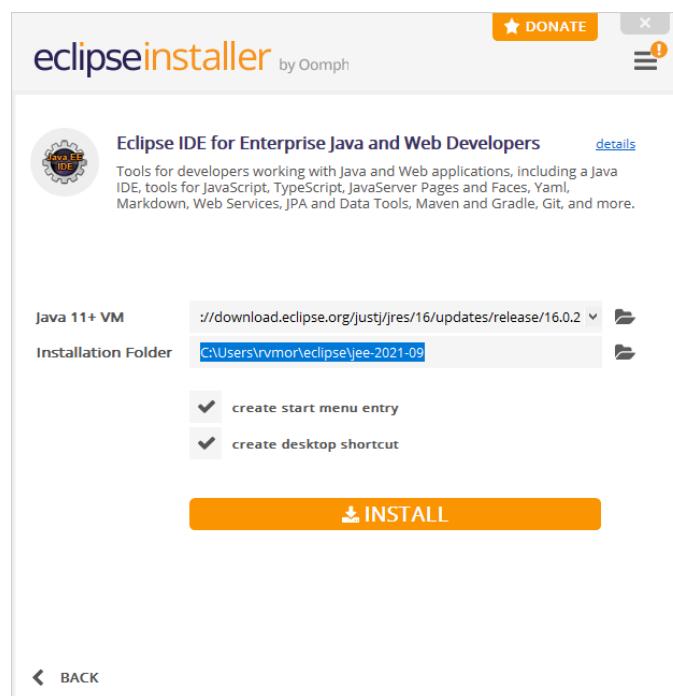


Figura 1 – Interface do instalador do Eclipse



Após selecionar a opção citada anteriormente, será exibida a tela de configuração da instalação, devendo o usuário informar o diretório no qual a IDE será instalada, conforme podemos observar na Figura 2. Configurado o diretório de instalação, basta clicar no botão *Install* para dar início ao processo de instalação do software.

Figura 2 – Tela de instalação do Eclipse





## 2.2 JDK

Além da instalação da IDE, o desenvolvedor Java também precisa instalar o kit de desenvolvimento Java (JDK – *Java Development Kit*). O JDK é composto por diversos componentes necessários para o desenvolvimento, gerenciamento e monitoramento das aplicações a serem desenvolvidas em linguagem Java. Dentre eles destacam-se a seguir alguns.

- **JRE**: máquina virtual e bibliotecas.
- **Javac**: compilador da linguagem Java.
- **Javadoc**: ferramenta de documentação.
- **Jdb (Java Debugger)**: ferramenta de depuração.
- **APIs**: conjunto de serviços da linguagem Java.

### 2.2.1 JRE

Para que uma aplicação Java possa ser executada em uma determinada máquina, é necessário que nela esteja instalado o ambiente de execução Java (JRE – *Java Runtime Environment*). O JRE é *plug-in* composto pela máquina virtual Java (JVM – *Java Virtual Machine*) e a biblioteca padrão da plataforma Java.

#### 2.2.1.1 Máquina virtual Java

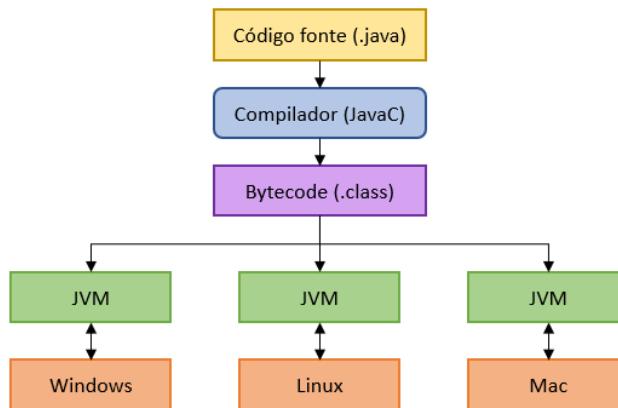
Uma das principais características da linguagem Java é a sua compatibilidade com diferentes sistemas operacionais. O componente que torna possível o desenvolvimento de aplicações Java independentes da plataforma é a máquina virtual Java (JVM – *Java Virtual Machine*).

Diferentemente de outras linguagens de programação, ao compilar o código-fonte de uma aplicação Java, não é gerado um arquivo executável, e sim um arquivo chamado *bytecode*. Ele é um arquivo intermediário gerado pelo compilador da linguagem Java, que será interpretado pela JVM e não contém qualquer vínculo com o sistema operacional do ambiente de desenvolvimento. Isso significa que o *bytecode* é um arquivo independente da plataforma, podendo ser executado em qualquer ambiente que tenha suporte a JVM, por exemplo, os sistemas operacionais Windows, Mac e Linux.



A JVM contém toda infraestrutura necessária para a execução de uma aplicação Java, convertendo o código em *bytecode* em código de máquina, compatível com a plataforma na qual ele está sendo executado, conforme nos mostra a figura a seguir.

Figura 3 – Funcionamento de uma aplicação Java



A JVM possui alguns módulos.

- **Loader:** responsável por carregar o *bytecode* na memória da máquina virtual.
- **Verificador:** analisará se o código em *bytecode* não contém erros.
- **Interpretador:** converterá o *bytecode* em um código de máquina compatível com a plataforma de execução.
- **Coletor de lixo:** fará o gerenciamento de memória da máquina virtual.
- **Compilador JIT (Just In Time):** responsável por otimizar a *performance* da aplicação durante a sua execução.

#### 2.2.1.2 Biblioteca padrão

A biblioteca padrão da linguagem Java contém um conjunto de pacotes que fornecem uma série de recursos necessários para a execução de qualquer aplicação desenvolvida em linguagem Java. Além disso, as bibliotecas auxiliam o programador no processo de desenvolvimento da aplicação, reduzindo a complexidade da implementação de diversas funcionalidades.

Supondo que a aplicação irá armazenar as informações do sistema em um banco de dados relacional, seria necessário que o programador desenvolvesse uma interface de comunicação com o banco de dados para



gerenciar os dados da sua aplicação. Porém, isso não é necessário, pois na biblioteca padrão da linguagem Java existe um pacote chamado *java.sql*, que contém a implementação de diversas classes e interfaces de comunicação com o banco de dados. Entre eles destacam-se os que se seguem.

- **Classe *DriverManager*:** utilizada para definir os parâmetros de acesso ao servidor de banco de dados: IP do servidor, porta de conexão, nome da fonte de dados, entre outros.
- **Interface *Connection*:** utilizada para estabelecer a conexão com o servidor de banco de dados.
- **Interface *ResultSet*:** utilizada para obter os resultados de uma *query* (consulta).
- **Interface *PreparedStatement*:** utilizada para executar um comando SQL.

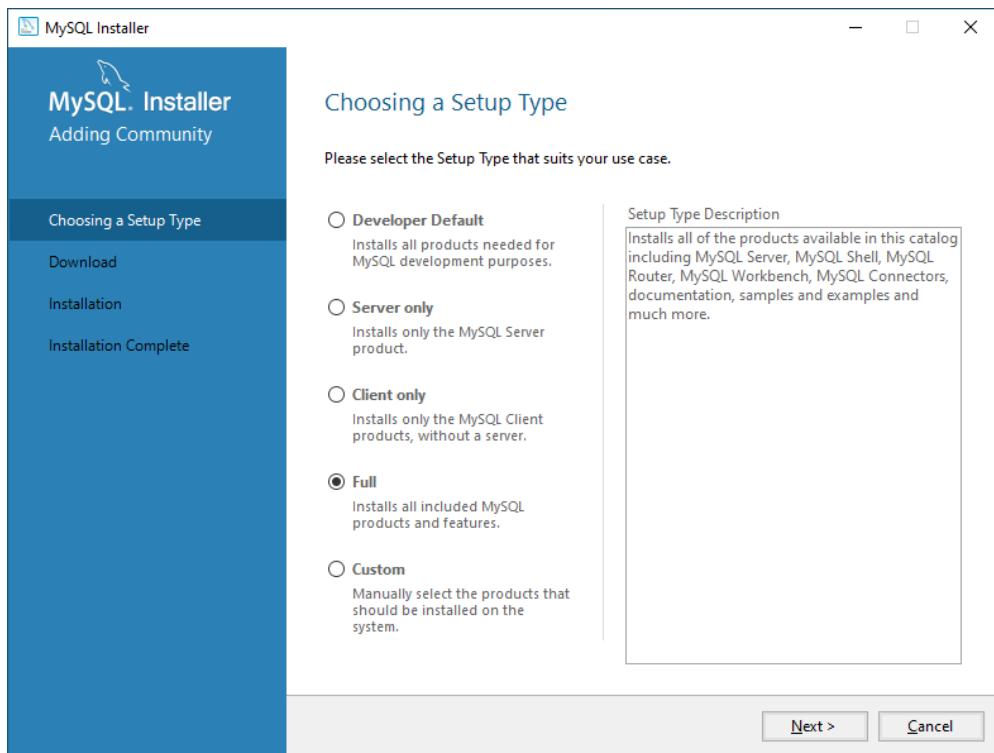
Além da biblioteca padrão, o desenvolvedor também pode adicionar ao projeto bibliotecas de terceiros, bastando para isso importar o arquivo JAR (*Java Archive*) do *framework* que será utilizado.

## 2.3 SGBD

O SGBD (Sistema Gerenciador de Banco de Dados) é mais uma ferramenta imprescindível para o desenvolvimento da aplicação, visto que necessitamos de um software que permita efetuar o armazenamento de dados do sistema. Entre os vários SGBDs disponíveis no mercado, vamos adotar o MySQL para o desenvolvimento da aplicação, tendo em vista que é uma ferramenta *open-source* robusta e utilizada por grandes corporações.

Para realizar a instalação do MySQL, é necessário baixar e instalar o MySQL *Workbench*, software que compreende o SGBD mais a interface gráfica para executarmos os comandos da Linguagem SQL. O instalador pode ser baixado por meio do site [dev.mysql](http://dev.mysql.com). Ao executar o instalador do MySQL *Workbench*, opte pelo tipo de instalação *Full* conforme a figura a seguir.

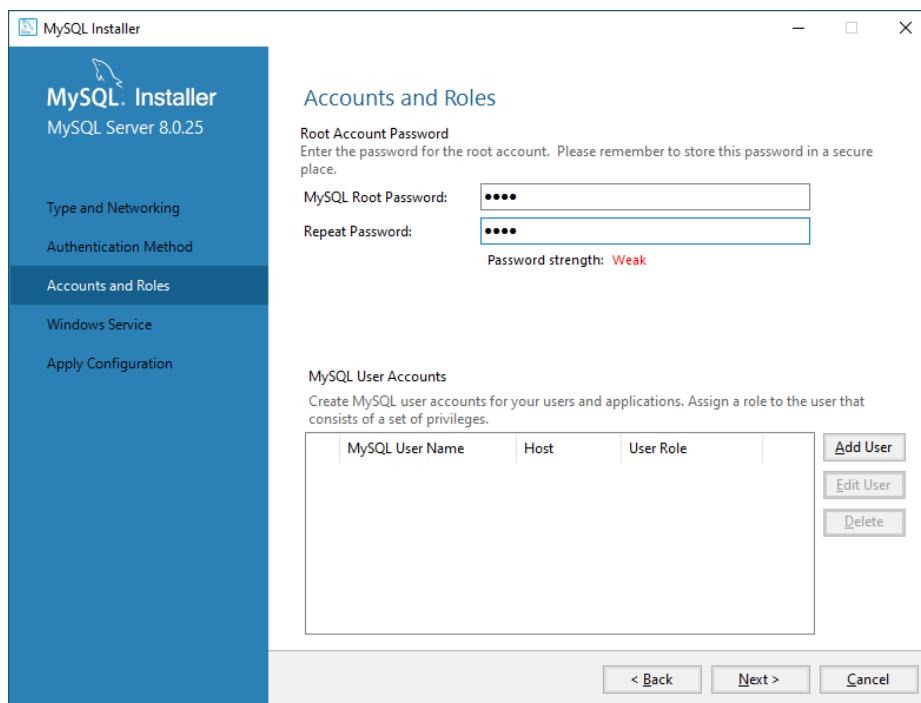
Figura 4 – Tela inicial do instalador MySQL Workbench



Durante a instalação, serão baixados os arquivos pertinentes ao tipo da instalação selecionada e, caso necessário, será solicitada a instalação de softwares complementares que podem ser baixados manualmente pelo próprio instalador. Boa parte do processo de instalação consiste em ir clicando no botão *Next*, porém duas telas merecem destaque. A primeira é a tela de cadastro *Accounts* e *Roles*, na qual é definida a senha do usuário administrador do MySQL, conforme podemos verificar na figura a seguir.



Figura 5 – Tela *Accounts and Roles* do instalador MySQL Workbench

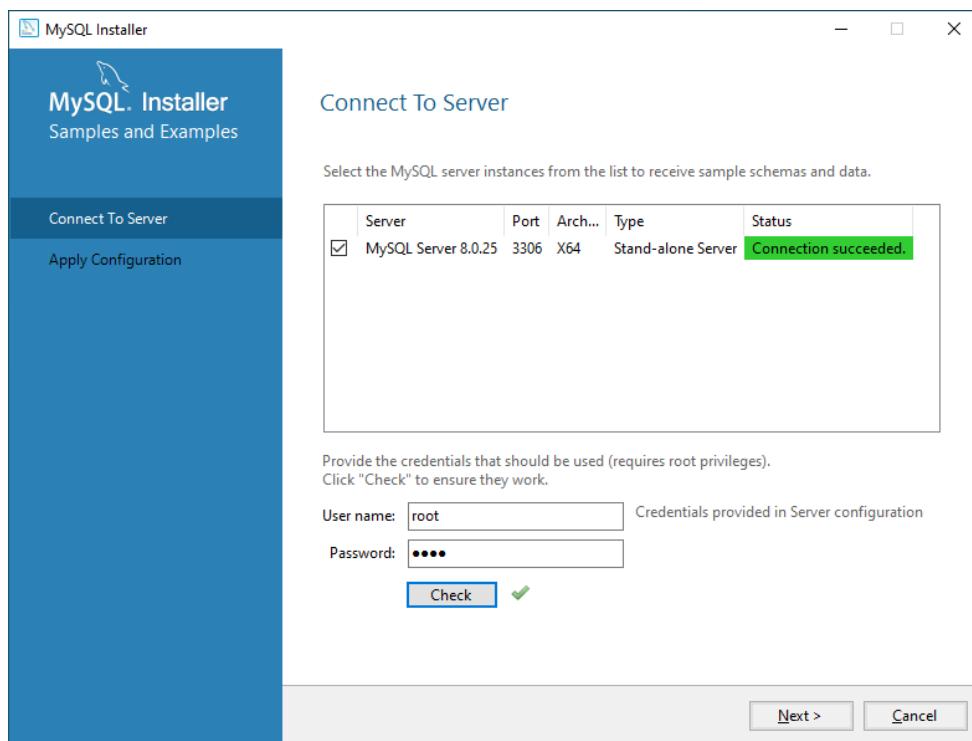


Cadastre uma senha de fácil recordação, pois, caso você venha a esquecê-la futuramente, terá que reinstalar o SGDB novamente, já que algumas funcionalidades do MySQL precisam de permissão de acesso do usuário administrador.

A segunda tela que merece destaque é a tela *Connect To Server*, nela podemos verificar se a instalação foi realizada com sucesso. Para isso, informe no campo *User name* o usuário administrador “root” sem aspas, no campo *Password*, a senha definida anteriormente na tela *Accounts and Roles* e, em seguida, clique no botão *Check*. Caso a instalação tenha sido realizada com sucesso, será exibida a mensagem *Connection succeeded* com o fundo verde na coluna *Status*, conforme pode ser visto na figura a seguir.



Figura 6 – Tela Connect To Server do instalador MySQL Workbench



## 2.4 Postman

É uma ferramenta que foi desenvolvida para auxiliar os desenvolvedores na criação e teste das APIs. Por meio dele, veremos as APIs que serão desenvolvidas durante o curso, tornando o processo de desenvolvimento mais rápido e objetivo, uma vez que está a todo momento abrindo um *browser* para realizar os testes. O Postman pode ser baixado no site e possui suporte para Windows, Mac ou Linux.

Figura 7 – Interface do Postman

The screenshot shows the Postman interface with the following details:

- Left Sidebar:** Collections (Twitter's Public Workspace), APIs (Twitter API v2), Mock Servers, Monitors, History.
- Request Details:**
  - Method: GET
  - URL: https://api.twitter.com/2/tweets/:id
  - Headers: None
  - Body: None
  - Tests: None
  - Settings: None
- Params:** A table with a single row for 'tweet.fields' set to 'attachments,author\_id,context\_annotations,conversation\_id,created\_at,entities,geo,id,in\_reply\_to\_user\_id,lang,non\_public\_metrics,possibly\_sensitive,promoted\_metrics,public\_metrics,referenced\_tweets,reply\_settings,source,text,withheld'.
- Path Variables:** A table with a single row for 'id' set to '1403216129661628420' with a note: 'Required. Enter a single Tweet ID.'
- Body:** Shows the JSON response from the API call:

```

1 {
2   "data": [
3     {
4       "id": "1403216129661628420",
5       "text": "Donovan Mitchell went down after a collision with Paul George toward the
6     end of Game 2. https://t.co/y9jhXh0LDN"
7   ]
8 }

```
- Documentation:** Shows the API endpoint https://api.twitter.com/2/tweets/:id with a note: 'This endpoint returns details about the Tweet specified by the requested ID.' It also mentions 'Full details, see the API reference for this endpoint.'
- Authorization:** Bearer token
- Request params:** A note: 'This request is using an authorization helper from collection Twitter API v2'
- Request fields:** A note: 'Comma-separated list of fields for the Tweet object.'
- Allowed values:** A list of fields: attachments,author\_id,context\_annotation,conversation\_id,created\_at,entities,geo,id,in\_reply\_to\_user\_id,lang,non\_public\_metrics,possibly\_sensitive,promoted\_metrics,public\_metrics,referenced\_tweets,reply\_settings,source,text,withheld
- Default values:** id;text
- OAuth1.0a User Context authorization:** A note: 'required if any of the following fields are included in the request: non\_public\_metrics,organic\_metrics,promo'
- View complete collection documentation →**

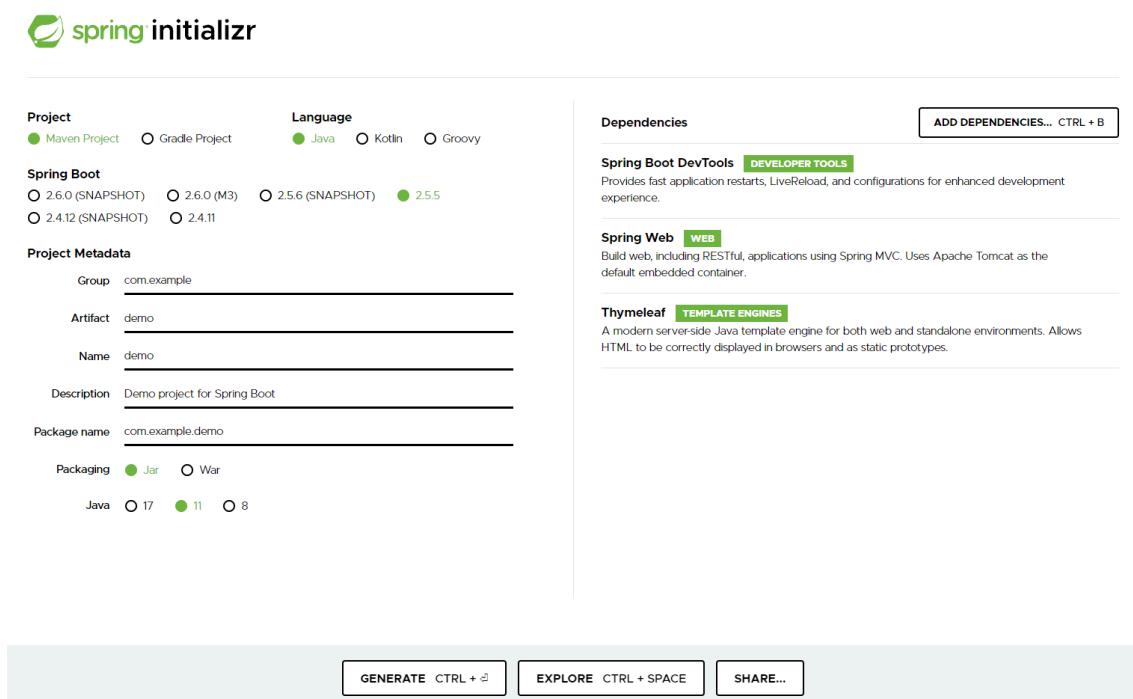
### TEMA 3 – SPRING BOOT

Começar um projeto do zero nem sempre é uma tarefa fácil, pois precisamos configurar diversos recursos da nossa aplicação antes de começarmos a codificação das regras de negócio. Esse processo basicamente consiste em baixar, instalar e configurar as bibliotecas, *frameworks* e serviços que serão utilizados no desenvolvimento da aplicação.

Para agilizar esse processo moroso da configuração inicial de um projeto, o time de desenvolvimento do *Spring* disponibilizou o *Spring Boot*, uma ferramenta que permite ao desenvolvedor criar um projeto *Spring* de forma rápida e fácil, por meio do site.



Figura 8 – Tela de configuração do *Spring Boot*



Ao acessar o site anterior, será exibida a tela de configuração do projeto *Spring*, conforme podemos observar na figura anterior. A configuração do projeto está dividida em cinco grupos.

- **Project**: define a ferramenta de gerenciamento e automação de construção do projeto.
- **Language**: define a linguagem de programação do projeto.
- **Spring Boot**: define a versão do *Spring* que será utilizada no projeto.
- **Project Metadata**: são as informações referentes ao projeto.
- **Dependencies**: lista de dependências que serão utilizadas no projeto.

A seguir, veremos com mais detalhes como esses grupos devem ser configurados.

### 3.1 Project

No grupo *Project*, deve-se identificar qual será a ferramenta de gerenciamento e automação de construção do projeto, ou de forma mais informal, qual será a ferramenta de *build* (construção) do projeto. Nesse grupo, podemos optar entre duas ferramentas diferentes: Maven e Gradle.



Essas ferramentas serão responsáveis por gerenciar as dependências do projeto, ou seja, cabe a elas baixar, instalar e configurar as bibliotecas, frameworks e serviços que serão utilizados no desenvolvimento da aplicação. Além disso, outra responsabilidade delas é a geração do arquivo de *build* do projeto, para que possamos realizar o processo de *deploy* (implantação) da aplicação.

### 3.2 Language

No grupo *Language*, deve-se identificar qual será a linguagem de programação que será utilizada para o desenvolvimento do projeto. Nesse grupo, temos três opções: Java, Kotlin e Groovy. Entre elas, iremos adotar a linguagem Java.

### 3.3 Spring Boot

No grupo *Spring Boot*, deve-se identificar qual será a versão do *Spring* que será utilizada para o desenvolvimento do projeto. Para isso, devemos compreender como se dá a nomenclatura das versões a fim de selecionar a mais adequada para o projeto que virá a ser desenvolvido. Note que algumas versões estão identificadas com M3, *SNAPSHOT* ou apenas pelo número da versão. As versões estão divididas em três grupos.

- **GA (General Availability)**: são as versões que foram lançadas ao público (versão estável) e nunca sofrerá qualquer alteração.
- **PRE**: são as versões de pré-lançamento e são identificadas pela letra M (acrônimo para marco). As versões de pré-lançamento têm como objetivo disponibilizar os novos recursos do *Spring* para que os desenvolvedores possam testá-los. Como as alterações estão em processo de validação (versão instável), essas versões podem apresentar erros durante a execução do projeto e não devem ser utilizadas para o desenvolvimento de um projeto comercial.
- **SNAPSHOT**: são atualizadas frequentemente e contém as alterações mais recentes do *framework* e, assim como as versões de pré-lançamento, não devem ser utilizadas para o desenvolvimento de um projeto comercial.



Para exemplificar esse processo, o ciclo de vida da versão 1.0.0 seguiria o seguinte processo.

- **1.0.0 SNAPSHOT:** essa versão é atualizada frequentemente com as alterações mais recentes.
- **1.0.0 M1:** sempre que é atingido um marco de desenvolvimento é lançada uma versão de pré-lançamento. A cada marco de desenvolvimento, o identificador do marco é incrementado em uma unidade, dessa forma, o primeiro marco de desenvolvimento é identificado pela sigla M1, o segundo por M2 e assim sucessivamente.
- **1.0.0 GA:** a partir do momento que foi lançado o último marco e a versão instantânea está completa e não apresenta qualquer erro, a versão é disponibilizada para o público. A partir desse momento, qualquer erro que for identificado e novas alterações serão desenvolvidas em uma nova versão.

### 3.4 Project Metadata

No grupo *Project Metadata*, serão definidas as características do projeto e contém os seguintes campos.

- **Group:** define o nome do grupo.
- **Artifact:** define o nome do artefato.
- **Name:** define o nome do projeto.
- **Description:** descrever o objetivo do projeto.
- **Package name:** define o nome do pacote principal do projeto.
- **Packaging:** define como o projeto será compactado.
- **Java:** define a versão do Java.

### 3.5 Dependencies

As dependências do projeto são referentes a bibliotecas, *frameworks* e serviços que serão utilizados no desenvolvimento do projeto. Sempre que quisermos adicionar algum desses recursos, basta adicionarmos a dependência no arquivo pom.xml do projeto. Adicionada a dependência, esta será baixada e adicionada ao projeto de forma automática pela ferramenta de gerenciamento de construção do projeto (Maven ou Gradle) escolhida no item *Project*.



Na tela de configuração do *Spring Boot*, são listadas as dependências mais utilizadas pelos desenvolvedores no item *Dependencies*. Nesse item, já podemos vincular ao projeto algumas dessas dependências, tornando o processo de configuração do projeto bem mais fácil e prático. A seguir, serão listadas algumas das dependências disponíveis para o desenvolvedor.

- ***Spring Boot DevTools***: ferramenta que contém configurações para uma experiência de desenvolvimento aprimorada e destaca-se principalmente por reiniciar a aplicação automaticamente a cada alteração no código fonte.
- ***Spring Web***: permite a criação de aplicações *web* trazendo por padrão o  *servlet container* Apache Tomcat já configurado para prover os serviços da aplicação.
- ***Spring Session***: fornece uma API para a implementação e gerenciamento das informações da sessão do usuário.
- ***Spring Web Services***: fornece suporte para a criação de *Web Services* do tipo *SOAP*.
- ***Thymeleaf***: ferramenta alternativa as especificações JSF e JSP para desenvolvimento das páginas *web*.
- ***JDBC API***: API que permite efetuar a conexão com banco de dados relacional.
- ***Spring Data JPA***: especificação para realizar a persistência de dados, usando JPA com *Spring Data* e *Hibernate*.
- ***Spring Security***: ferramenta para autenticação de usuários altamente personalizável para controle de acessos da aplicação.



## TEMA 4 – CRIANDO UM PROJETO COM O SPRING BOOT

Agora que sabemos como configurar um projeto pelo *Spring Boot*, vamos criar o nosso projeto para iniciarmos o processo de implementação, utilizando o framework *Spring*. O primeiro passo é acessarmos o *Spring Boot* pelo *link* apresentado anteriormente.

Ao acessar a página de configuração do projeto, adotaremos estas configurações conforme a figura a seguir.

Figura 9 – Configuração do projeto utilizando o *Spring Boot*

The screenshot shows the Spring Initializr web application. It has two main sections: 'Project' and 'Dependencies'. In the 'Project' section, 'Maven Project' is selected under 'Language' (Java). Under 'Spring Boot', version '2.5.6 (SNAPSHOT)' is selected. In the 'Dependencies' section, 'Spring Boot DevTools' is selected under 'DEVELOPER TOOLS'. Below it, 'Spring Web' is selected under 'WEB'. At the bottom, there are three buttons: 'GENERATE' (CTRL + D), 'EXPLORE' (CTRL + SPACE), and 'SHARE...'.

Project	Language
<input checked="" type="radio"/> Maven Project	<input checked="" type="radio"/> Java
<input type="radio"/> Gradle Project	<input type="radio"/> Kotlin
	<input type="radio"/> Groovy

Spring Boot	
<input type="radio"/> 2.6.0 (SNAPSHOT)	<input type="radio"/> 2.6.0 (M3)
<input type="radio"/> 2.5.6 (SNAPSHOT)	<input checked="" type="radio"/> 2.5.5
<input type="radio"/> 2.4.11	<input type="radio"/> 2.4.12 (SNAPSHOT)

Project Metadata	
Group	br.com.springboot
Artifact	springboot
Name	springboot
Description	Demonstração do framework Spring
Package name	br.com.springboot
Packaging	<input type="radio"/> Jar <input checked="" type="radio"/> War
Java	<input type="radio"/> 17 <input checked="" type="radio"/> 11 <input type="radio"/> 8

Dependencies ADD DEPENDENCIES... CTRL + B

**Spring Boot DevTools** DEVELOPER TOOLS  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

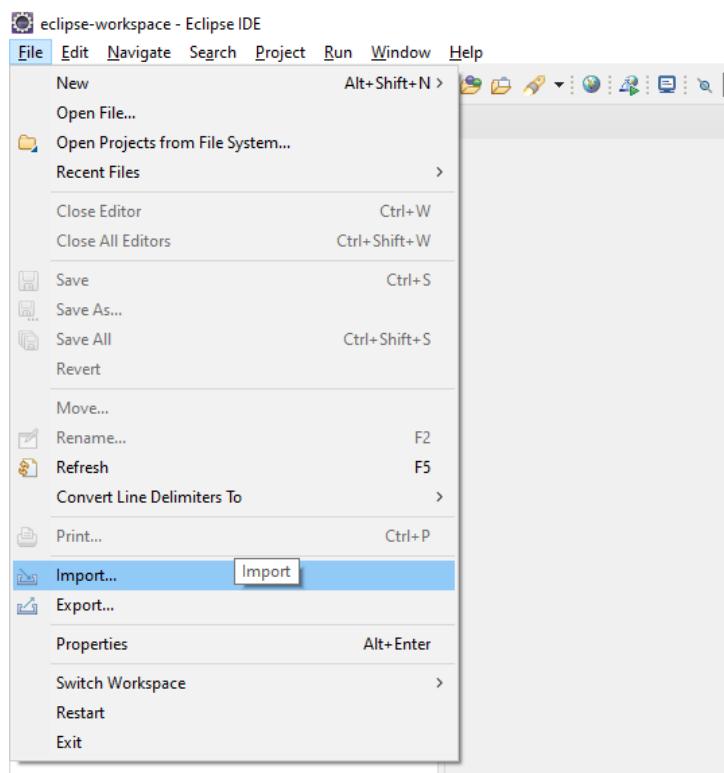
GENERATE CTRL + D EXPLORE CTRL + SPACE SHARE...

Após efetuar a parametrização anterior, basta clicar no botão *Generate* ou pressione a combinação de teclas *Ctrl + Enter* para baixar o arquivo de configuração do projeto. Será gerado um arquivo zip com o nome do projeto, o qual deve ser extraído para posteriormente ser importado na IDE que será adotada para o desenvolvimento da aplicação. Como iremos utilizar o Eclipse como IDE, demonstraremos passo a passo como realizar a importação do projeto por meio dessa ferramenta de desenvolvimento.

Inicializando o Eclipse, importaremos o arquivo de configuração do projeto utilizando a tela de importação pelo menu “*File > Import*”, conforme podemos visualizar na Figura 10.

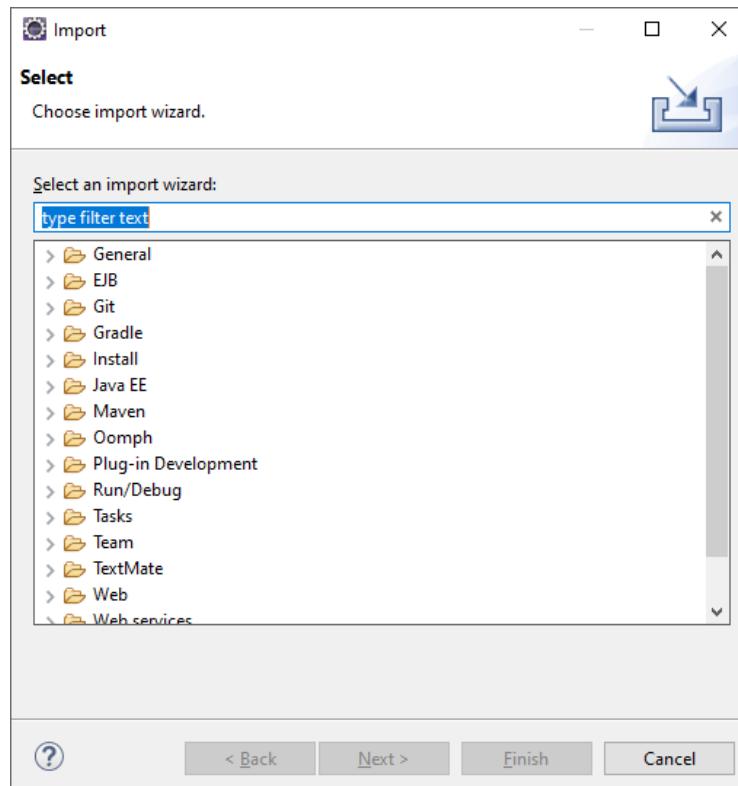


Figura 10 – Menu para importação de um projeto já existente no Eclipse



Ao acessar o menu citado anteriormente, será exibida a tela de importação conforme ilustra a Figura 11.

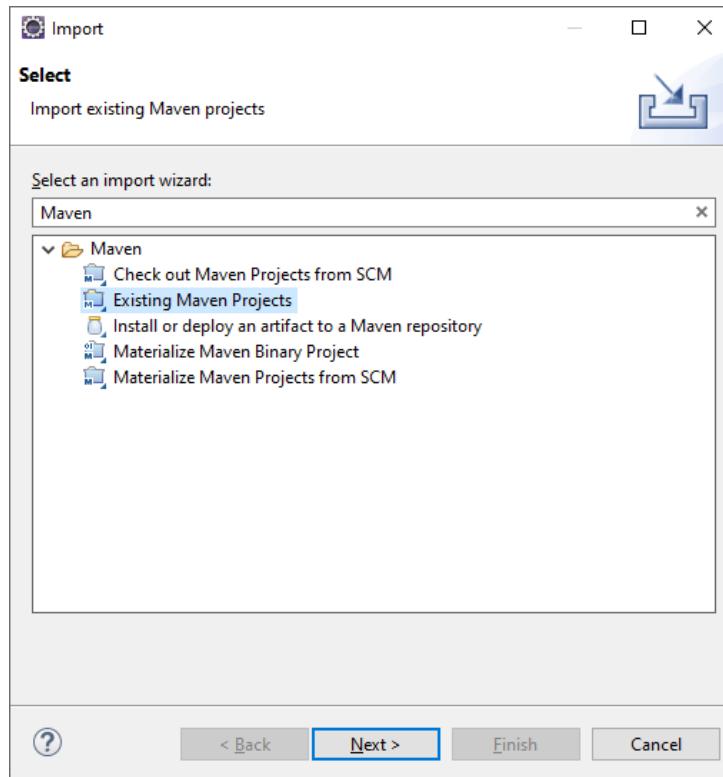
Figura 11 – Tela de importação do Eclipse





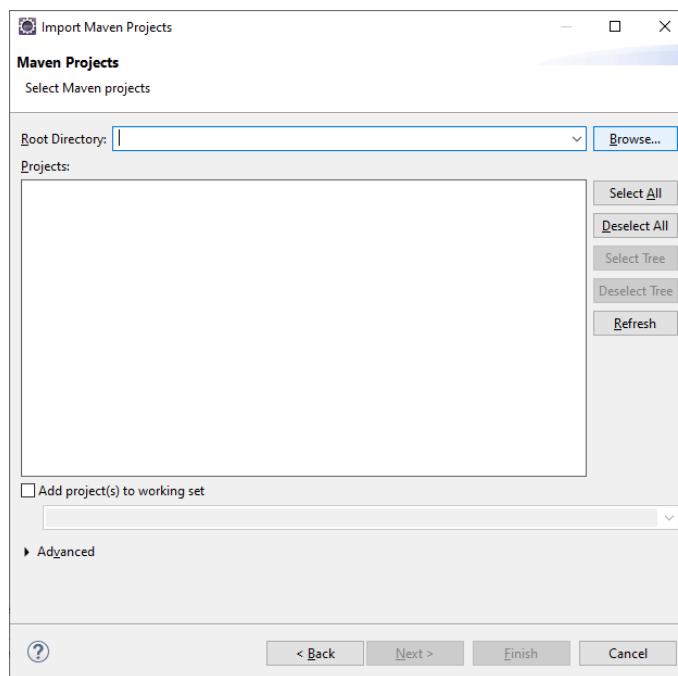
No campo texto digite, “Maven”, sem aspas, selecione a opção *Existing Maven Projects* e clique no botão *Next*, conforme a figura a seguir.

Figura 12 – Tela de importação do Eclipse para projetos Maven



Ao realizar a operação anterior, será exibida a tela de importação de um projeto Maven, conforme podemos observar na Figura 13.

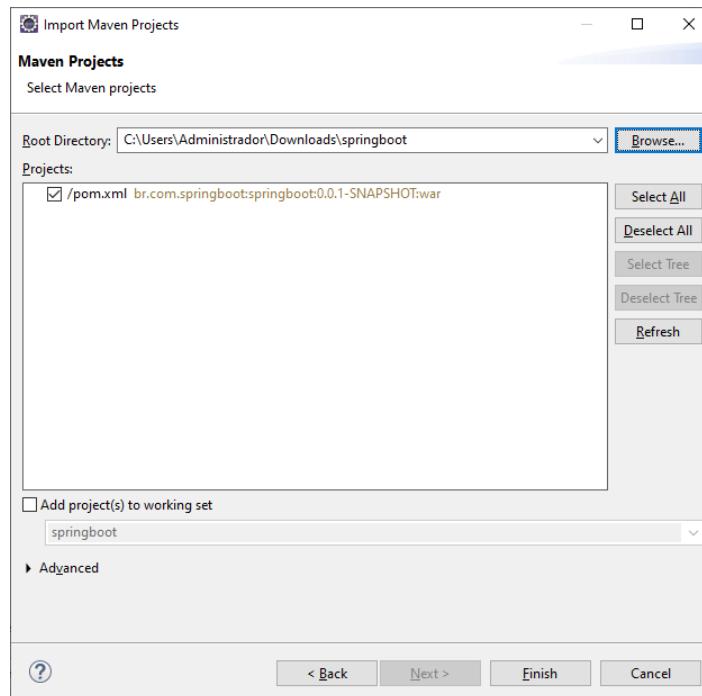
Figura 13 – Importação de um projeto Maven





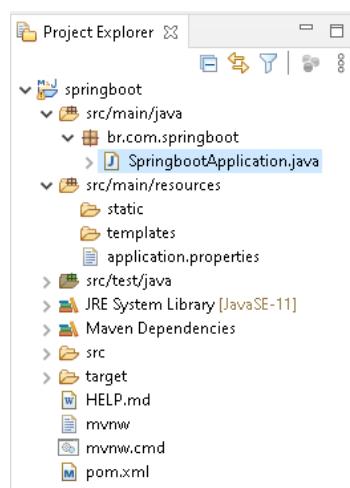
Nessa tela, deve ser informado o diretório no qual se encontra o arquivo de configuração do projeto, aquele que geramos pelo *Spring Boot*. Utilize o botão *Browse* para realizar tal operação, após selecionado o diretório, clique no botão *Finish*.

Figura 14 – Importação do projeto *Spring Boot*



Após efetuar a operação citada anteriormente, o Eclipse iniciará o processo de importação do projeto. Realizada a importação do projeto, precisamos compreender como está organizada a estrutura do projeto. Vamos expandir o projeto clicando duas vezes sobre o nome do projeto na aba *Project Explorer* no canto superior esquerdo, conforme podemos observar na Figura 15.

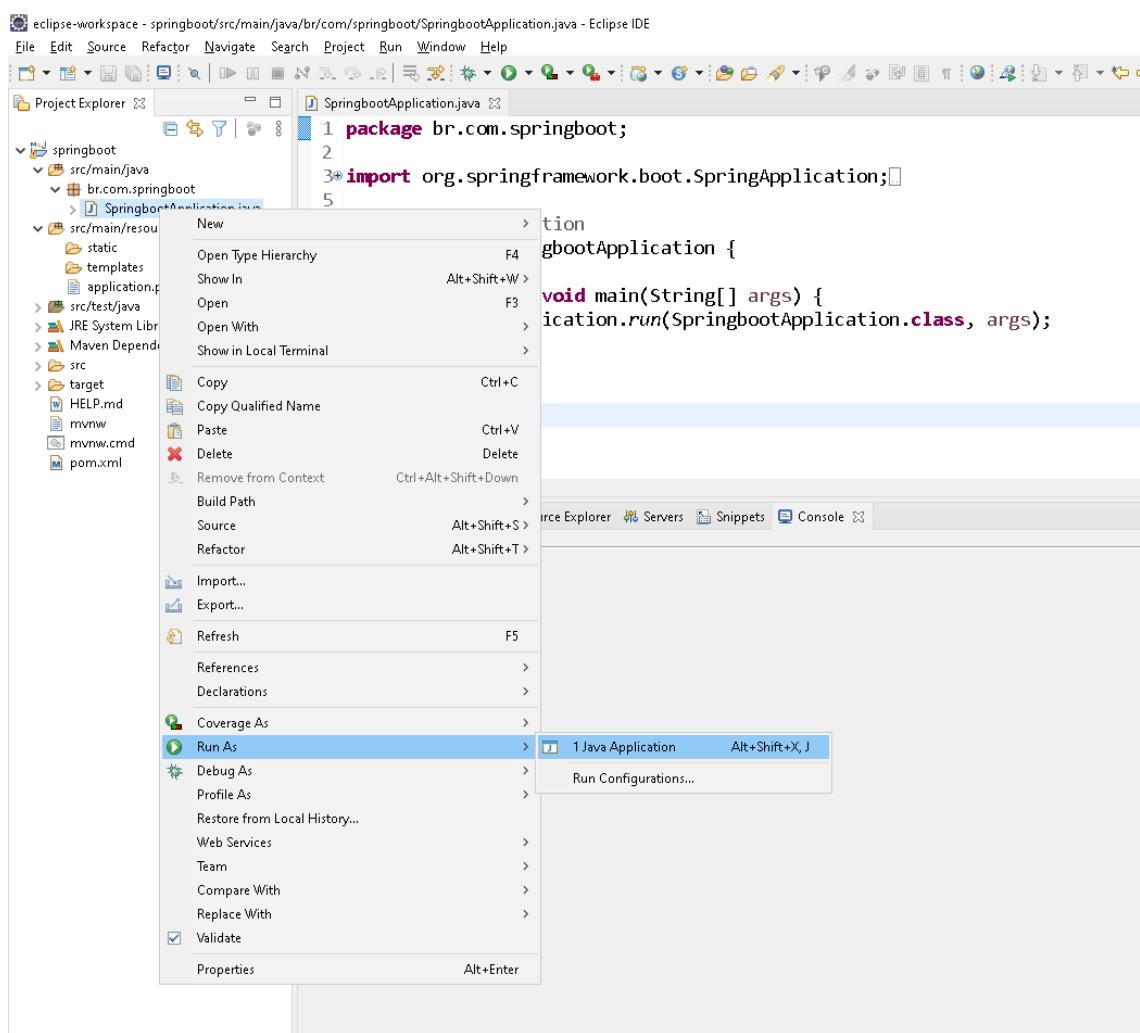
Figura 15 – Estrutura de um projeto *Spring*





O projeto é composto por diversas pastas e arquivos, localizar a pasta `src/main/java` e vá expandindo os nós até encontrar o arquivo `SpringbootApplication`, classe responsável por iniciar a aplicação. Para verificarmos se o projeto foi importado com sucesso e não contém nenhum erro, vamos executá-lo clicando com o botão direito sobre a classe `SpringbootApplication` e selecionando o menu `Run as > Java Application` conforme a figura a seguir.

Figura 16 – Inicializando a aplicação Spring



Como o Spring Boot já traz o Tomcat configurado por padrão, esse `Servlet Container` irá iniciar a aplicação, e o `log` referente a esse processo será exibido no console do Eclipse. Caso a inicialização tenha ocorrido com sucesso, no `log`, além de não apresentar qualquer mensagem de erro, informará a porta na qual a aplicação está rodando, indicando que a aplicação já está apta para ser utilizada. Essa informação pode ser encontrada na penúltima linha do `log` do tipo "Tomcat started on port 8080 (http) with context path".

Figura 17 – Aplicação iniciada com sucesso

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows a project named "springboot" containing "src/main/java" with a file "SpringbootApplication.java".
- Code Editor:** Displays the code for "SpringbootApplication.java":

```
package br.com.springboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringbootApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

- Console:** Shows the application logs starting with:

```
2021-11-02 20:44:41.792 INFO 8872 --- [ restartedMain] br.com.springboot.SpringbootApplication : Starting SpringbootApplication using Java 16.0.1 on DESKTOP-64_16_0_1.v20210520-1205
2021-11-02 20:44:41.793 INFO 8872 --- [ restartedMain] br.com.springboot.SpringbootApplication : No active profile set, falling back to default profiles: def
2021-11-02 20:44:41.797 INFO 8872 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-02 20:44:41.807 INFO 8872 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : For additional web related logging consider setting the "log
2021-11-02 20:44:43.569 INFO 8872 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-02 20:44:43.599 INFO 8872 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-02 20:44:43.599 INFO 8872 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.54]
2021-11-02 20:44:43.701 INFO 8872 --- [ restartedMain] o.a.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-02 20:44:43.701 INFO 8872 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1793
2021-11-02 20:44:44.265 INFO 8872 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : LiveReload server is running on port 3529
2021-11-02 20:44:44.361 INFO 8872 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-02 20:44:44.377 INFO 8872 --- [ restartedMain] br.com.springboot.SpringbootApplication : Started SpringbootApplication in 3.181 seconds (JVM running
```

- Maven:** Shows the Maven build output.

## TEMA 5 – CONCEITOS WEB

Neste tema, abordaremos alguns conceitos necessários para que possamos iniciar o processo de codificação de uma aplicação web utilizando o *Spring Framework*. Para isso, é necessário compreender dois conceitos fundamentais, como funciona o protocolo HTTP e o arquitetura MVC.

### 5.1 Protocolo HTTP

Permite a troca de mensagens entre máquinas distintas por meio de hipertexto, sendo HTTP um acrônimo para *HyperText Transfer Protocol*, ou seja, Protocolo de Transferência de Hipertexto. Esse protocolo atua dentro da camada de aplicação do modelo OSI e é por meio dele que conseguimos navegar pela web, acessando os mais variados sites disponíveis na rede de internet.

Quando acessamos um determinado site, por exemplo o Google, abrimos o navegador e digitamos o endereço <http://www.google.com.br>. Esse endereço é o que chamamos de URL (*Uniform Resource Locator*) do site. O navegador, por sua vez, comprehende esse conteúdo e faz uma requisição para o servidor <www.google.com.br>, responsável por atender as requisições pertinentes a essa URL. A URL representa um recurso com o qual deseja-se acessar, podendo ser uma imagem, arquivo, vídeo ou página web.



Agora que sabemos o que é uma URL, precisamos entender como ela é composta e, para isso, vamos adotar o seguinte endereço: <<http://www.gov.br/mec/pt-br>>. Essa URL é composta por três elementos.

- **Protocolo:** o prefixo anterior aos “://” da URL identifica o protocolo de comunicação que será adotado para troca de dados entre as máquinas. Nesse caso, estamos utilizando o protocolo HTTP, mas poderíamos utilizar outros protocolos como FTP, TCP etc.
- **Host:** identifica o servidor responsável por fornecer os recursos de um domínio, na URL o *host* está compreendido entre os “://” e a primeira “/” após os “://”, sendo o *host* da URL <[www.gov.br](http://www.gov.br)>. O navegador para acessar o recurso dessa URL irá converter o seu *host* em um endereço de IP por meio do DNS *Lookup*, a fim de identificar o servidor responsável pelo processamento da requisição efetuada pelo usuário.
- **URL Path:** identifica o recurso que será acessado pelo usuário, na URL é identificado logo após o *host*, no endereço adotado como exemplo, a URL Path é /mec/pt-br. Nesse caso, o recurso a ser acessado é uma página web, mas poderia ser uma imagem caso a URL Path fosse </images/ico-mec.png/>.

### 5.1.1 Métodos

Os recursos gerenciados pela aplicação podem ser manipulados de diversas formas por meio de uma URL. Quando uma requisição é efetuada pelo cliente, além da URL, precisamos informar também o método de manipulação do recurso. Na tabela a seguir, são elencados os principais métodos do protocolo HTTP e suas respectivas funcionalidades.

Tabela 1 – Métodos do protocolo HTTP

Método	Funcionalidade
<b>GET</b>	Consultar os dados de um recurso.
<b>POST</b>	Criar um recurso.
<b>PUT</b>	Atualizar os dados de um recurso.
<b>PATCH</b>	Atualizar parcialmente um recurso.
<b>DELETE</b>	Remover os dados de um recurso.



<b>HEAD</b>	Consultar os cabeçalhos da resposta.
<b>OPTIONS</b>	Identificar quais manipulações podem ser efetuadas em um recurso.

Para realizar o CRUD (*Create, Read, Update and Delete*) de uma determinada entidade, ou seja, implementar as operações de inserção, consulta, atualização e remoção de uma entidade no banco de dados, devemos criar as URLs pertinentes a cada uma dessas operações e associá-las a um determinado método. Na tabela a seguir, listamos as URL Path e seus respectivos métodos para exemplificar o *CRUD* dos usuários conforme o padrão *REST*.

Tabela 2 – Modelo *REST*

Método	URL Path	Operação
<b>GET</b>	/usuarios	Retorna todos os usuários do sistema.
<b>GET</b>	/usuarios/id	Retorna os dados do usuário com base no seu id.
<b>POST</b>	/usuarios	Cadastra um usuário.
<b>PUT</b>	/usuarios/id	Atualiza os dados de um usuário com base no seu id.
<b>DELETE</b>	/usuarios/id	Remove um usuário com base no seu id.

Note que, embora tenhamos URL Paths repetidas, o que define a operação a ser realizada é o método informado na requisição HTTP.

### 5.1.2 Porta

Para que a aplicação saiba quais são as requisições que ela deve processar, é necessário que na URL seja informado o número da porta. O Tomcat por padrão atende as requisições da aplicação na porta 8080, portanto, para consultar todos os usuários cadastrados no sistema, utilizariammos a requisição GET 8080/usuarios.

### 5.1.3 Query String

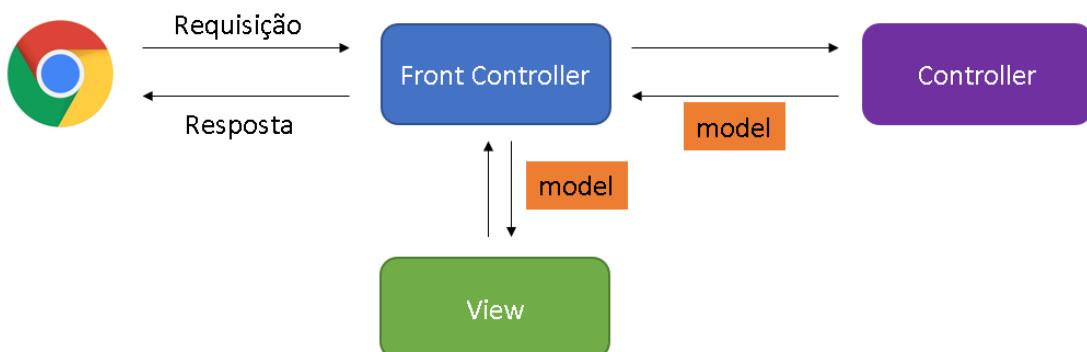
Permite-nos passar informações adicionais na URL por meio do operador “?” para que a aplicação possa realizar um processamento mais apurado de uma determinada requisição. Supondo que desejamos saber todos os usuários ativos no sistema, poderíammos montar a requisição para GET 8080/usuarios?ativo=true.



## 5.2 Arquitetura MVC

Antes de iniciarmos o processo de codificação da nossa aplicação, precisamos compreender a arquitetura de uma aplicação *web*. Um dos padrões de arquitetura mais utilizados para o desenvolvimento de aplicações *web* é o padrão arquitetural MVC, que é um acrônimo para o termo em inglês *Model* (Modelo), *View* (Visão) e *Controller* (Controlador).

Figura 18 – Padrão MVC do *Spring*



Na figura anterior, temos uma visão do padrão MVC implementado pelo *Spring*, exibindo os três componentes que compõem esse modelo arquitetural (*Model*, *View* e *Controller*) com a adição do *Front Controller*. Por meio de um navegador, será exibida a interface gráfica da aplicação, na qual o usuário realizará uma determinada tarefa. Ao realizar tal tarefa, será enviada ao servidor da aplicação uma requisição HTTP para que esta possa ser processada.

Essa requisição será recebida pelo *Front Controller*, componente responsável por receber as requisições dos clientes e encaminhá-las para os *Controllers* pertinentes, que, por sua vez, realizarão o processamento da requisição efetuada pelo cliente. Caso esse processamento exija algum tipo de interação com o banco de dados, o *Controller* irá realizar essa operação e retornar os dados para o *Front Controller* por meio de um *Model*, que irá conter todas as informações pertinentes à requisição solicitada.

Ao receber o *Model*, o *Front Controller* irá encaminhá-lo a *View*, responsável por realizar o processamento da página *web*, uma vez que, para o seu desenvolvimento, utilizamos componentes (JSP, JSF, entre outros). Portanto, a *View* realizará esse processamento, adicionando os dados providos pelo *Model* ao componente utilizado para o desenvolvimento da página *web*,



---

devolvendo para o *Front Controller* somente o código HTML da página web processada.

Assim que a *View* retornar a página web para o *Front Controller*, este irá retornar para o navegador uma resposta HTTP que fará o seu processamento, neste caso, exibindo a página web com os dados pertinentes à requisição efetuada anteriormente. A partir disso, inicia-se um novo ciclo de requisições por parte do usuário e, para cada nova solicitação, o mesmo fluxo de trabalho será efetuado para resolução das requisições.

## FINALIZANDO

Nesta aula, aprendemos a configurar o nosso ambiente de desenvolvimento, instalando as ferramentas necessárias para a construção de uma aplicação utilizando o *Spring Framework*. Após configurado o ambiente de desenvolvimento, por meio do *Spring Boot*, criamos o um projeto *Spring* de forma fácil e rápida sem a necessidade de configurarmos tudo de forma manual.

Além disso, também vimos como importar esse projeto dentro do Eclipse e os conceitos pertinentes para o desenvolvimento de uma aplicação web, de como montar uma API REST de requisições HTTP e como é o fluxo de processamento dessa requisição dentro da aplicação web baseado no padrão MVC.



---

## REFERÊNCIAS

APACHE. **What is a Maven?**: Maven – Introduction. Disponível em: <apache.org>. Acesso em: 19 dez. 2021.

ECLIPSE **About the Eclipse Foundation**. The Eclipse Foundation, 2021.

MySQL **About MySQL**. MySQL, 2021.

ORACLE. **Java Platform Standard Edition 17 Development Kit**: JDK 17. Disponível em: <oracle.com>. Acesso em: 19 dez. 2021.

POSTMAN. **What is Postman?**. Postman API Platform, 2021.

SNYK. **JVM Ecosystem report 2018**: About our Platform and Application. JVM Ecosystem, 2021.

SPRING. **Initializr**. Spring Initializr, 2021.

SPRING. **Why Spring?**. Spring, 2021.

SPRING. **Web MVC Framework**: 17. Web MVC framework. Disponível em: <spring.io>. Acesso em: 19 dez. 2021.



# **DESENVOLVIMENTO WEB**

## **BACK END**

AULA 3

Prof. Rafael Veiga de Moraes



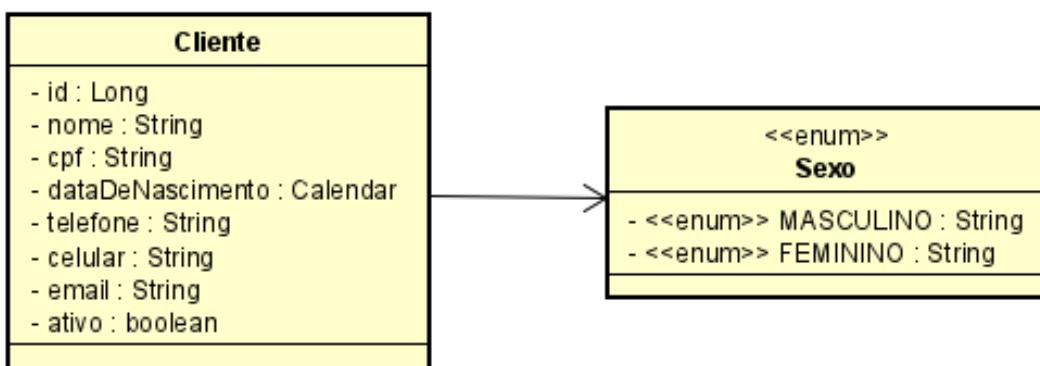
## CONVERSA INICIAL

Uma aplicação Web é dividida em duas partes: *back-end* e *front-end*, na qual a primeira é responsável pela implementação das regras de negócio e a segunda por prover a interface gráfica da aplicação. Nesta aula, daremos início ao processo de desenvolvimento de um sistema de controle de estoque, focando inicialmente no *back-end*, mais especificamente na criação dos objetos e na sua interação com o banco de dados.

### TEMA 1 – CRIANDO A MODEL

Vamos iniciar o desenvolvimento da nossa aplicação pela tela de cadastro de Cliente. Para isso, devemos inicialmente criar a classe que irá conter os seus dados. Conforme especificado no diagrama de classes da aplicação, devemos implementar a classe Cliente conforme nos mostra a figura abaixo.

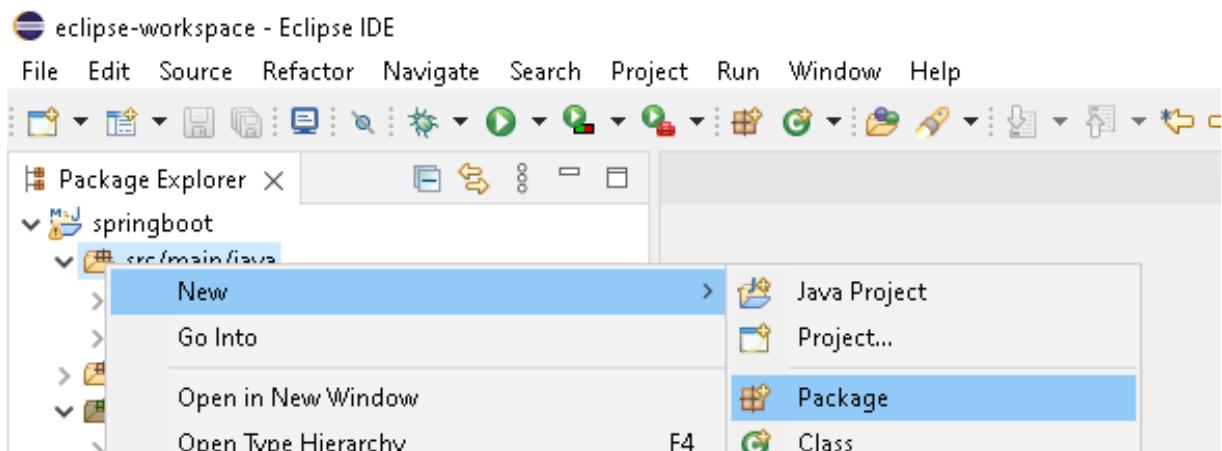
Figura 1 – Classe Cliente



Como iremos criar diversas classes durante o processo de desenvolvimento da aplicação, devemos agrupá-las em pacotes de acordo com as suas responsabilidades. Quanto às classes que irão representar as entidades do banco de dados, como, por exemplo, a classe Cliente, iremos manter dentro do pacote *br.com.springboot.model*. Para criá-lo, basta clicar com o botão direito do mouse sobre a pasta *src/main/java* na raiz do projeto e selecionar o menu *New > Package*, conforme a figura a seguir.

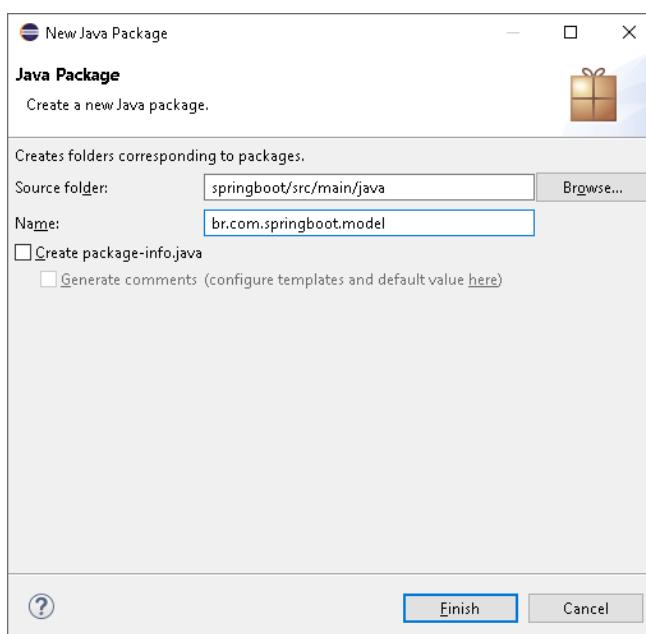


Figura 2 – Menu Package



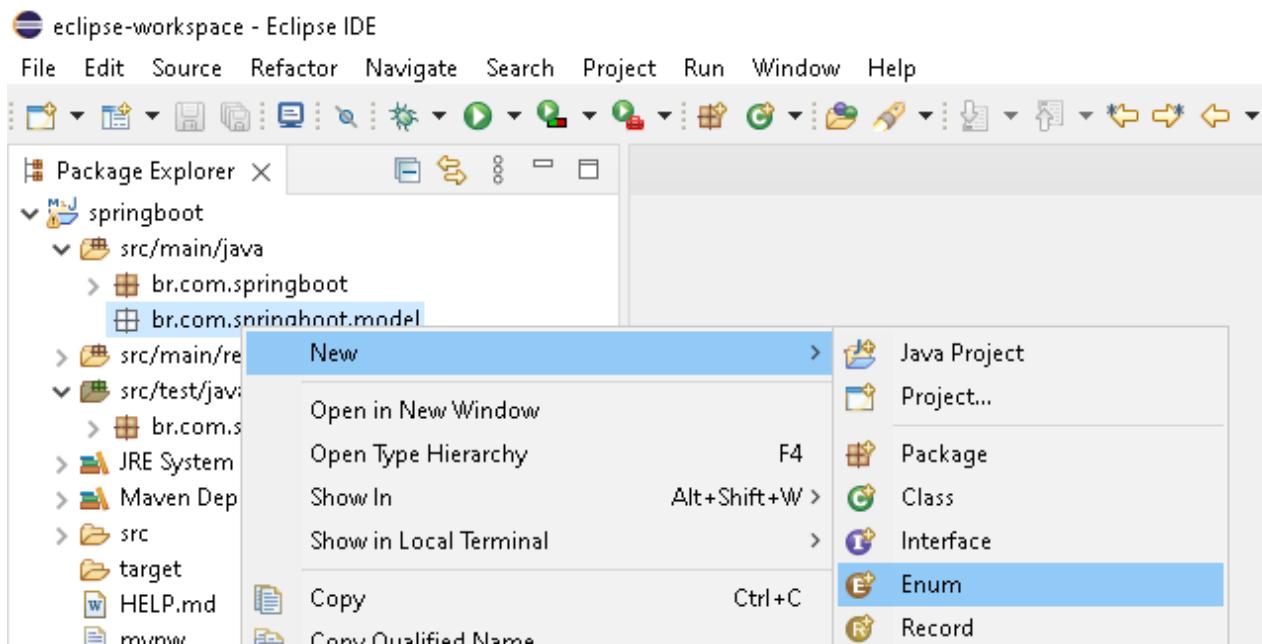
Ao clicar sobre a opção *Package* (Pacote), será aberta a tela para a criação do pacote. Nela, devemos informar o nome do pacote que desejamos criar no campo *Name*, conforme exemplificado na Figura 3.

Figura 3 – Tela de cadastro do pacote



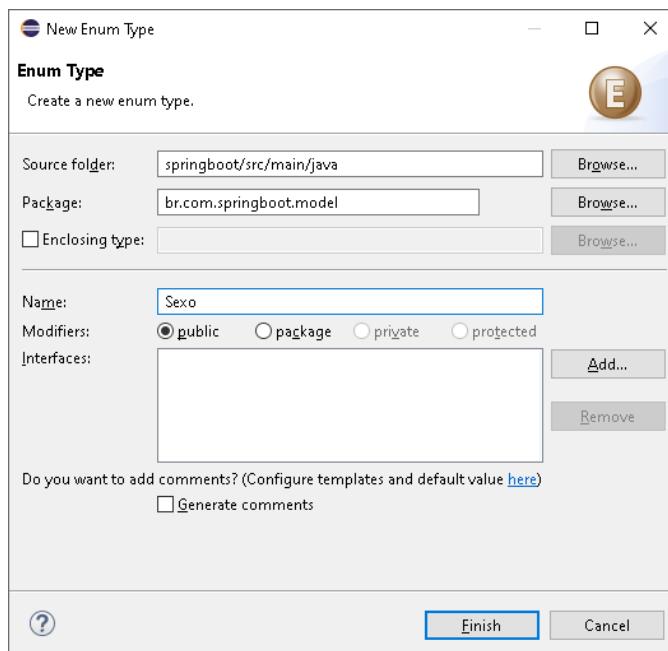
Informado o nome do pacote no campo *Name*, clique no botão *Finish*. Criado o pacote, vamos implementar a enumeração Sexo e a classe Cliente, iniciando a codificação pela primeira. Como a enumeração compõe a classe Cliente, ela também será adicionada ao pacote *br.com.springboot.model*. Para adicioná-la a esse pacote, basta clicar com o botão direito sobre ele e selecionar o menu *New > Enum*, conforme podemos visualizar na Figura 4.

Figura 4 – Menu Enum



Ao clicar sobre a opção *Enum* (Enumeração), será aberta a tela para a criação da enumeração. Nela devemos informar o nome da enumeração que desejamos criar no campo *Name*, conforme exemplificado na Figura 5.

Figura 5 – Tela de cadastro da enumeração



Informado o nome da enumeração no campo *Name* clique no botão *Finish*. A enumeração Sexo irá conter apenas dois valores (Masculino e Feminino) e sua implementação pode ser vista no quadro a seguir.



## Quadro 1 – Implementação da enumeração Sexo

```
package br.com.springboot.model;

public enum Sexo {
    MASCULINO("Masculino"),
    FEMININO("Feminino");

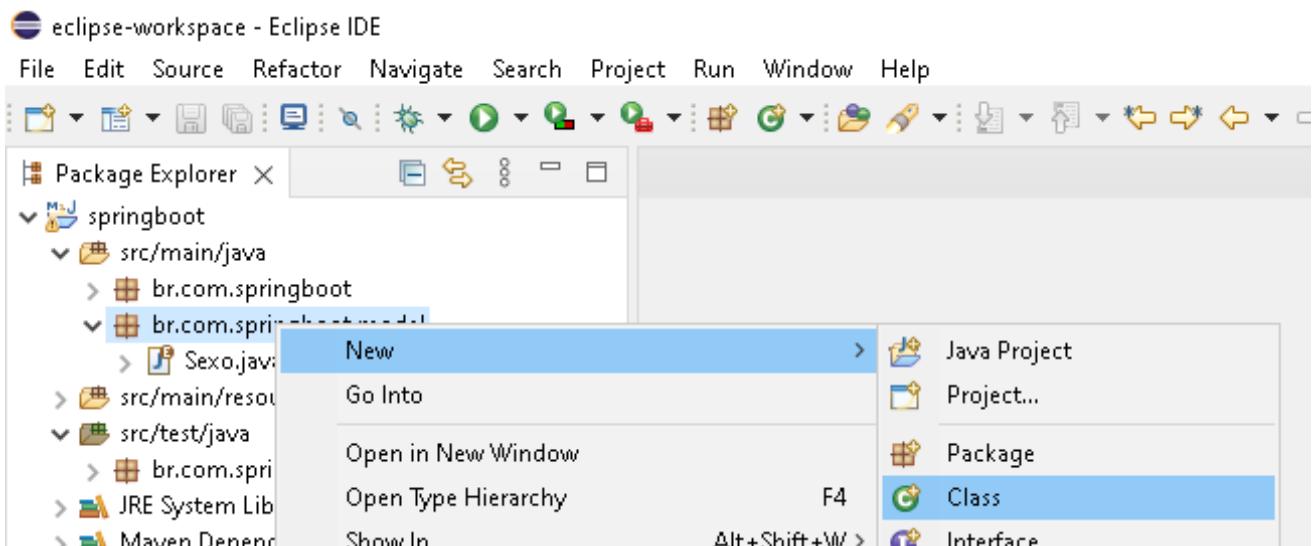
    private String descricao;

    Sexo(String descricao) {
        this.descricao = descricao;
    }

    public String getDescricao() {
        return this.descricao;
    }
}
```

Finalizada a enumeração Sexo, podemos agora iniciar a implementação da classe Cliente. Para 5ria-la, clique com o botão direito sobre o pacote *br.com.springboot.model* e selecione o menu *New > Class*, conforme a Figura 6.

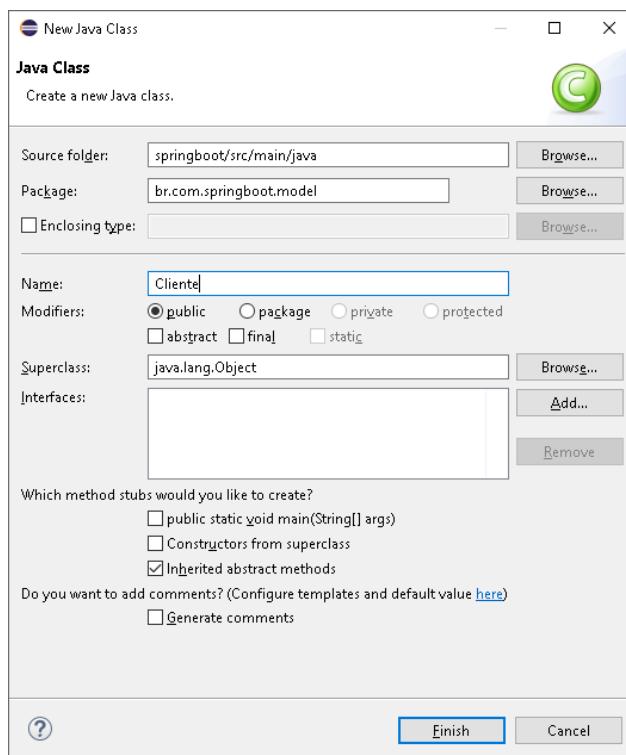
Figura 6 – Menu Class



Ao clicar sobre a opção *Class* (Classe), será aberta a tela para a criação da classe. Nela, devemos informar o nome da classe que desejamos criar no campo *Name*, conforme exemplificado na Figura 7.



Figura 7 – Tela de cadastro da classe



Informado o nome da classe no campo *Name*, clique no botão *Finish*. Adicione os atributos à classe Cliente, conforme especificado no diagrama de classes da aplicação. A declaração dos atributos da classe Cliente pode ser visualizada no quadro 2.

Quadro 2 – Classe Cliente

```
package br.com.springboot.model;

import java.time.LocalDate;

public class Cliente {

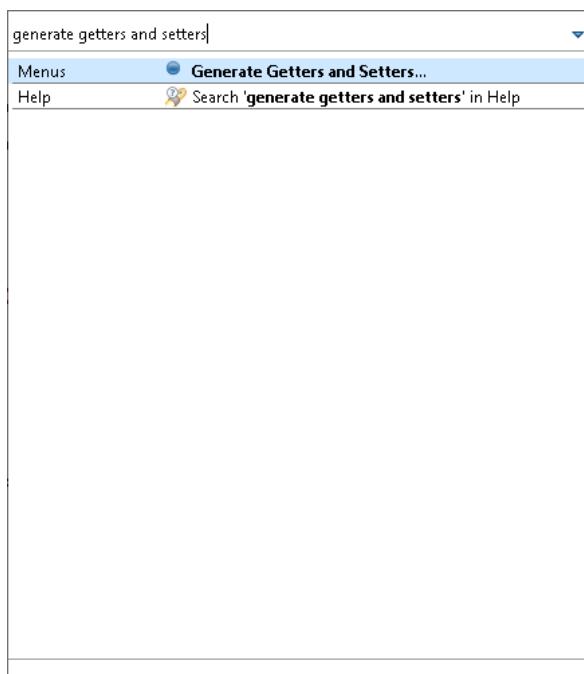
    // Atributos
    private Long id;
    private String nome;
    private String cpf;
    private LocalDate dataDeNascimento;
    private Sexo sexo;
    private String telefone;
    private String celular;
    private String email;
    private boolean ativo;
}
```

Adicionado os atributos à classe Cliente, utilize a ferramenta *Generate getters and setters* do Eclipse para gerar os *getters* e *setters* da classe Cliente



de forma automática. Para acessar esse recurso, clique sobre a lupa no canto superior direito da IDE ou a combinação das teclas Ctrl + 3 para abrir a tela de pesquisa do Eclipse. No campo de pesquisa, digite *Generate getters and setters* e clique sobre a opção que será listada conforme podemos visualizar na Figura 8.

Figura 8 – Menu Generate getters and setters



Clique sobre a opção *Generate getters and setters*. Na tela que será exibida, clique sobre o botão *Select All* para selecionar todos os atributos e depois no botão *Generate* para gerar os *getters* e *setters* da classe Cliente, finalizando assim a implementação da classe Cliente.

## TEMA 2 – ADICIONANDO AS DEPENDÊNCIAS DE ACESSO A DADOS

Após implementar a classe Cliente, precisamos agora implementar a classe de acesso a dados referente essa classe, que será responsável por prover os métodos de interação com o banco de dados. Antes disso, é necessário realizar a configuração do ambiente de desenvolvimento, adicionando as dependências pertinentes para que a aplicação possa se conectar ao banco de dados.

Iremos adicionar duas dependências para realizarmos a interação com o banco de dados. A primeira dependência a ser adicionada ao projeto será a *MySQL Connector/J*, responsável por prover o *driver* de conexão com o



MySQL, visto que iremos utilizar esse SGBD para o gerenciamento de dados da aplicação. A segunda dependência a ser adicionada ao projeto será a *Spring Boot Starter Data JPA*, para que possamos implementar os métodos de acesso a dados utilizando o Hibernate, *framework* de mapeamento objeto-relacional, em conjunto com a JPA (*Java Persistence API*), especificação Java EE para o mapeamento objeto-relacional.

As dependências podem ser encontradas no site dos fornecedores. Porém, para não ter que ficar acessando diferentes sites a todo momento, recomendo o site *MVN Repository* que pode ser acessado por meio da seguinte url <<https://mvnrepository.com>>. Nesse site, você irá encontrar de forma rápida e fácil diversas dependências que podem ser adicionadas ao seu projeto. As duas dependências elencadas anteriormente se encontram nos quadros 3 e 4 logo a seguir.

Quadro 3 – Dependência MySQL Connector/J

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

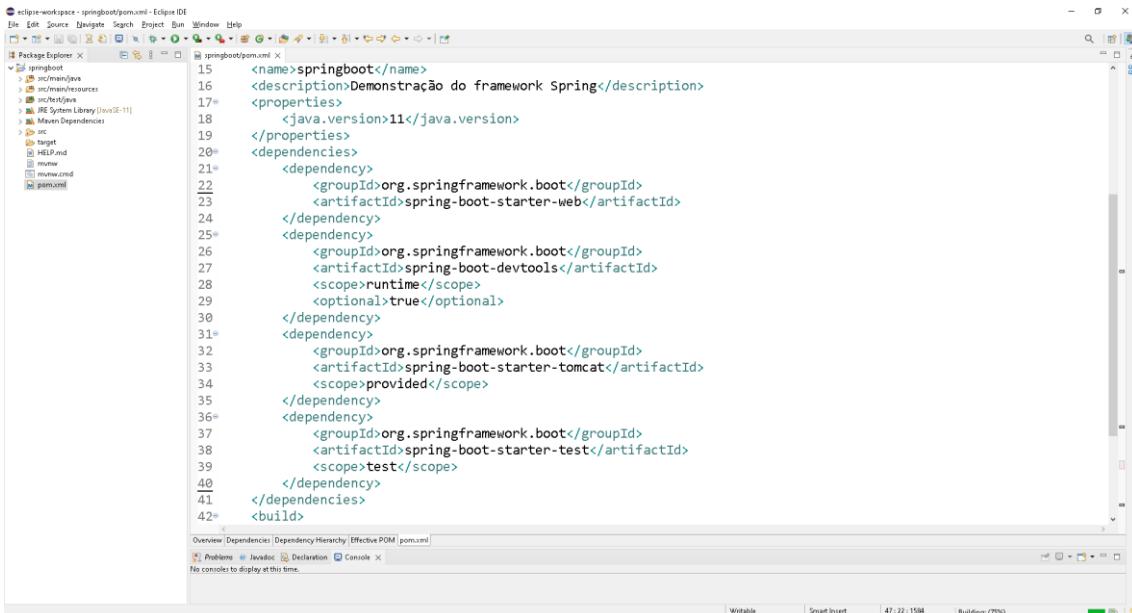
Quadro 4 – Dependência Spring Boot Starter Data JPA

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

Para adicionar as dependências anteriores a nossa aplicação, precisamos adicioná-las ao arquivo pom.xml que está localizado na raiz do projeto, responsável pelo gerenciamento de todas as dependências da aplicação. Ao abrir o arquivo pom.xml, todas as dependências do projeto estarão listadas entre as tags `<dependencies>` e `</dependencies>`. Note que as duas dependências que adicionamos ao projeto por meio do *Spring Boot*, *Spring Boot DevTools* e *Spring Web* se encontram entre essas tags, conforme podemos visualizar na Figura 9.



Figura 9 – Arquivo pom.xml o pom.xml



```
<name>springboot</name>
<description>Demonstração do framework Spring</description>
<properties>
    <java.version>11</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
```

Ao adicionar as dependências listadas anteriormente ao arquivo pom.xml, salve as alterações para que as dependências sejam baixadas e incorporadas ao projeto.

## 2.1 Hibernate

O Hibernate é o *framework* de mapeamento objeto relacional Java mais conhecido do mercado por abstrair a complexidade no processo de conversão de dados em objetos e vice-versa. Além de facilitar a persistência de dados da aplicação, esse *framework* apresenta um desempenho superior em relação ao código JDBC (*Java Database Connectivity*), tanto em termos de produtividade do desenvolvedor quanto em desempenho em tempo de execução.

Além da sua própria API “nativa”, o Hibernate também é uma implementação da especificação JPA (*Java Persistence API*). Portanto, ele pode ser facilmente utilizado em qualquer ambiente que suporte essa especificação, incluindo aplicações Java SE, servidores de aplicações Java EE, contêineres Enterprise OSGI etc.

Outra vantagem de se utilizar o Hibernate é que ele não requer tabelas ou campos de banco de dados especiais e gera grande parte do SQL no momento da inicialização do sistema, não comprometendo a performance da aplicação em tempo de execução. Aliado a isso, o Hibernate fornece suporte aos idiomas naturais da programação orientada a objetos como herança, polimorfismo, associação, composição e a estrutura de coleções Java.



Para demonstrar o porquê desse *framework ORM* (*Object Relational Mapping*) ser uma das ferramentas mais utilizadas pelos desenvolvedores Java, vamos comparar a diferença entre duas classes de acesso a dados diferentes, uma utilizando a JDBC e a outra o Hibernate, realizando a persistência de dados do mesmo objeto. Para isso, iremos adotar a classe Categoria, conforme podemos observar no Quadro 6.

Quadro 5 – Classe categoria

```
public class Categoria {  
    private Long id;  
    private String nome;  
    private boolean ativo;  
  
    // Declaração dos getters e setters  
}
```

Quadro 6 – Classe de acesso a dados utilizando JDBC

```
public class CategoriaDAO {  
    private Connection connection;  
  
    public CategoriaDAO() {  
        this.connection = new ConnectionFactory().getConnection();  
    }  
  
    public void insere(Categoria categoria) throws SQLException {  
        String sql = "insert into categoria(nome, ativo) values (?, ?)";  
        PreparedStatement stmt = this.connection.prepareStatement(sql);  
        stmt.setString(1, categoria.getNome());  
        stmt.setBoolean(2, categoria.isAtivo());  
        stmt.execute();  
        stmt.close();  
    }  
}
```

A classe de acesso a dados referente à classe Categoria utilizando JDBC pode ser visualizada no Quadro 7, enquanto a classe de acesso a dados, para essa mesma classe utilizando o Hibernate, pode ser visualizada no Quadro 8.



## Quadro 7 – Classe de acesso a dados utilizando Hibernate

```
@Transactional  
public class CategoriaDAO {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public void insere(Categoria categoria) {  
        entityManager.persist(categoria);  
    }  
}
```

Para não tornar ambas as classes de acesso a dados tão extensas, foram implementados apenas o método insere. Porém, a análise a seguir pode ser aplicado aos demais métodos da classe que contém código SQL na sua implementação.

Note que a classe de acesso a dados implementada utilizando o Hibernate é bem mais enxuta que a classe de acesso a dados via JDBC, visto que, com apenas um único comando, realizamos a inserção de um objeto no banco de dados, enquanto, na outra classe, foram utilizados cinco comandos mais a declaração do comando SQL a ser executado.

Outra enorme vantagem ao utilizar o Hibernate é que, a cada atualização na estrutura da tabela Categoria no SGBD, como, por exemplo, adicionar uma nova coluna à tabela, não é necessário refatorar os métodos de acesso a dados, pois, na sua implementação, não há código SQL, diferentemente do que ocorre na classe de acesso a dados com JDBC.

Também podemos deixar a encargo do Hibernate o gerenciamento das tabelas do SGBD. Isso significa que o próprio *framework* se encarrega de criar a tabela no banco de dados referente a uma determinada classe e, até mesmo, adicionar as colunas à tabela quando for adicionado um novo atributo a uma classe já existente. E como o Hibernate consegue realizar a persistência de dados e gerenciar as tabelas do SGBD se não injetamos código SQL nas classes? Por meio do mapeamento objeto-relacional, próximo tema desta aula.

## TEMA 3 – MAPEAMENTO OBJETO-RELACIONAL

O mapeamento objeto-relacional (ORM – *Object Relational Mapping*) é utilizado para conversão de dados entre banco de dados relacionais e linguagens orientadas a objetos, estabelecendo uma correlação entre classes e



entidades. Para estabelecer essa correlação, utilizamos as anotações da JPA (*Java Persistence API*).

A anotação é um recurso da linguagem Java que permite automatizarmos algumas tarefas por meio de metadados, sendo ela identificada dentro do código Java pelo prefixo da arroba (@). Na tabela a seguir, serão descritas as principais anotações da JPA para posteriormente, realizarmos o mapeamento objeto-relacional das classes da nossa aplicação.

Tabela 1 – Anotações da JPA

Anotação	Descrição
<b>@Entity</b>	Especifica que a classe é uma entidade
<b>@Table</b>	Especifica parâmetros referentes a tabela de uma entidade
<b>@Column</b>	Especifica que o atributo da classe é uma coluna
<b>@Transient</b>	Especifica que o atributo não será persistido
<b>@Temporal</b>	Especifica que o atributo é do tipo Date ou Calendar
<b>@Enumerated</b>	Especifica que o atributo a ser persistido é uma enumeração
<b>@Id</b>	Especifica que o atributo é a chave primária da entidade
<b>@IdClass</b>	Especifica que a chave primária da entidade é composta por múltiplos campos
<b>@GeneratedValue</b>	Especifica a estratégia de geração dos valores da chave primária
<b>@JoinColumn</b>	Especifica qual coluna será utilizada para realizar a junção de uma associação
<b>@JoinColumns</b>	Especifica quais colunas serão utilizadas para realizar a junção de uma associação
<b>@OneToOne</b>	Especifica um relacionamento do tipo 1:1
<b>@OneToMany</b>	Especifica um relacionamento do tipo 1:N
<b>@ManyToMany</b>	Especifica um relacionamento do tipo N:N
<b>@ManyToOne</b>	Especifica um relacionamento do tipo N:1

Para compreender como e em que empregar as anotações da JPA, vamos realizar o mapeamento objeto-relacional da classe Categória



mencionada anteriormente. Como desejamos persistir os objetos dessa classe, devemos empregar a anotação `@Entity`. Por padrão, o Hibernate entende que o nome da classe é equivalente ao nome da tabela no banco de dados. Porém, caso eles sejam diferentes, deve-se utilizar a anotação `@Table` e especificar na sua propriedade `name`, o nome da tabela.

Com relação as colunas o Hibernate adota o mesmo critério, ele comprehende que os nomes dos atributos são equivalentes aos nomes das colunas, porém, caso eles sejam diferentes, deve-se utilizar a anotação `@Column` e especificar na sua propriedade `name`, o nome da coluna.

Como toda tabela deve ter uma chave primária, devemos mapear o atributo chave da classe com a anotação `@Id`, quando esse for uma chave primária simples, e com o `@IdClass`, quando se tratar de uma chave primária composta. Também podemos definir por meio da anotação `@GeneratedValue` qual será a estratégia de definição do seu conteúdo, como, por exemplo, gerar o id de forma automática.

Portanto, para realizar o mapeamento objeto-relacional da classe `Categoria`, vamos convencionar que:

- Os dados serão persistidos na tabela `categorias`;
- O atributo `id` é chave primária e será gerado de forma automática pelo SGBD;
- O conteúdo do atributo `id` está associado à coluna `categoria_id`;
- Os demais atributos, `nome` e `ativo` estão associados às colunas de mesmo nome.

O mapeamento objeto-relacional da classe `Categoria`, conforme as proposições anteriormente pode ser visualizada no quadro a seguir.



Quadro 8 – Mapeamento objeto-relacional da classe Categoria

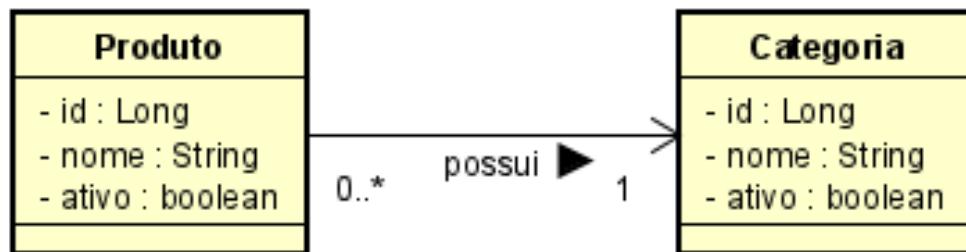
```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
@Table(name="categorias")
public class Categoria {
    @Id
    @Column(name="categoria_id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String nome;
    private boolean ativo;

    // Declaração dos getters e setters
}
```

Por meio do mapeamento objeto-relacional o Hibernate será capaz de realizar a persistência de dados, visto que, por meio das anotações da JPA, especificamos em qual tabela os dados serão persistidos e em quais colunas o conteúdo dos atributos serão armazenados.

Figura 10 – Relacionamento Produto x Categoria



Além do mapeamento das classes, precisamos realizar também mapear o relacionamento entre as entidades do sistema. Na Figura 10, temos o relacionamento entre as classes Produto e Categoria, no qual a sua cardinalidade é do tipo 1:N (um para muitos).

Por estar do lado N do relacionamento, a classe Produto irá receber a chave estrangeira; por isso, adiciona-se a essa classe um atributo do tipo Categoria. A implementação da classe Produto pode ser visualizada no Quadro 10.



## Quadro 9 – Classe Produto

```
public class Produto {  
    private Long id;  
    private String nome;  
    private Categoria categoria;  
    private boolean ativo;  
  
    // Declaração dos getters e setters  
}
```

Por ser um atributo oriundo de um relacionamento, a categoria no mapeamento objeto-relacional deverá ser mapeada de acordo com a cardinalidade do relacionamento. Como estamos mapeando a classe Categoria dentro da classe Produto, fazemos a leitura do relacionamento com base na entidade Produto. Dessa forma, temos que muitos produtos estão vinculados a uma categoria, ou seja, a cardinalidade do mapeamento é do tipo muitos para um (N:1).

Definida a cardinalidade do mapeamento, basta utilizar a anotação da JPA correspondente, para que o Hibernate saiba como efetuar a persistência dos dados e realizar a junção entre as tabelas. Referente à cardinalidade dos mapeamentos, temos as seguintes anotações:

- **@OneToOne**: cardinalidade um para um (1:1);
- **@OneToMany**: cardinalidade um para muitos (1:N);
- **@ManyToOne**: cardinalidade muitos para um (N:1);
- **@ManyToMany**: cardinalidade muitos para muitos (N:N).

Além da cardinalidade, podemos também especificar qual será o nome da coluna que faz referência à chave estrangeira. Para especificá-la, deve-se utilizar a anotação **@JoinColumn**, definindo o seu nome por meio da propriedade *name*. O mapeamento objeto-relacional para a classe Produto pode ser visualizada no Quadro 11.



#### Quadro 10 – Mapeamento objeto-relacional da classe Produto

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="produtos")
public class Produto {
    @Id
    @Column(name="produto_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    @ManyToOne
    @JoinTable(name="categoria_id")
    private Categoria categoria;
    private boolean ativo;

    // Declarar getters e setters
}
```

#### TEMA 4 – CRIANDO A CLASSE DE ACESSO A DADOS

Compreendido o conceito de mapeamento objeto-relacional e de como realizá-lo por meio das anotações da JPA, iremos retomar o desenvolvimento da nossa aplicação. Anteriormente, havíamos criado a classe Cliente, restando nesse momento apenas efetuar o seu mapeamento objeto-relacional, que pode ser visualizado no quadro a seguir.



## Quadro 11 – Mapeamento objeto-relacional da classe Cliente

```
package br.com.springboot.model;

import java.time.LocalDate;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name="clientes")
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(nullable=false, length = 50)
    private String nome;
    @Column(length = 11)
    private String cpf;
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    @Column(name="data_nascimento", columnDefinition = "DATE")
    private LocalDate dataDeNascimento;
    @Enumerated(EnumType.STRING)
    private Sexo sexo;
    @Column(length = 10)
    private String telefone;
    @Column(length = 11)
    private String celular;
    @Column(length = 50)
    private String email;
    private boolean ativo;

    // Declaraçõe getters e setters
}
```

Note que, no mapeamento em questão, foram utilizadas algumas anotações e propriedades que não haviam sido empregadas anteriormente. Dentre elas, destacam-se:

- **@Temporal**: anotação utilizada para mapear atributos do tipo Date ou Calendar. Deve-se especificar qual é o tipo do dado a ser persistido utilizando a enumeração *TemporalType*, que possui três valores:
- **DATE**: utilizado quando o conteúdo do atributo for do tipo data.
- **TIME**: utilizado quando o conteúdo do atributo for do tipo hora.



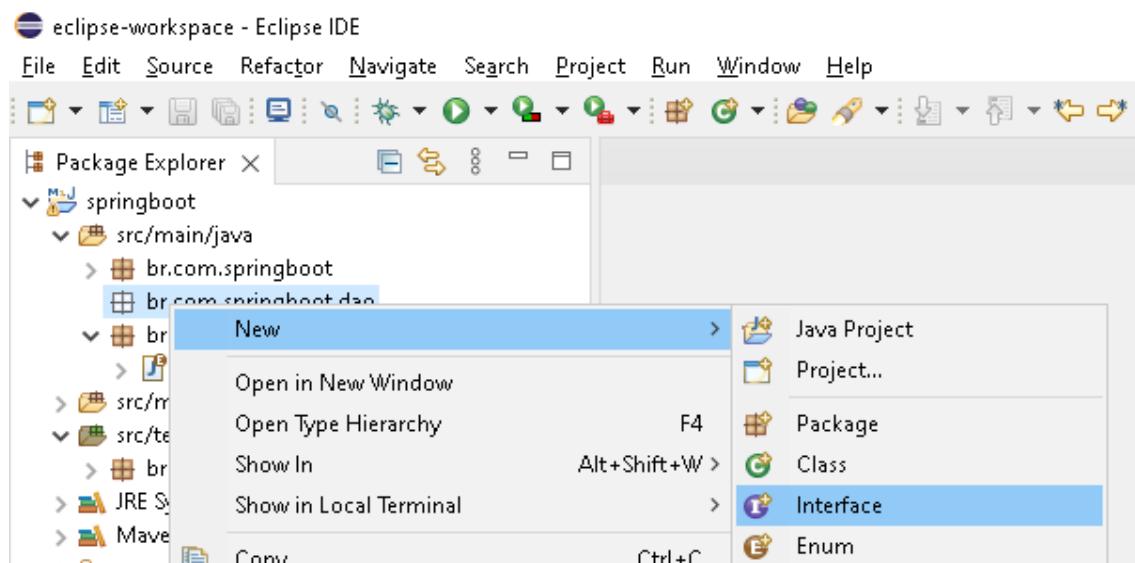
- **TIMESTAMP**: utilizado quando o conteúdo do atributo for do tipo data e hora.
- **@Enumerated**: anotação utilizada para mapear as enumerações. Deve-se especificar como o conteúdo do atributo será persistido por meio da enumeração *EnumType*, que possui dois valores:
- **STRING**: ao adotar esse tipo, o atributo será armazenado como texto no banco de dados e seu conteúdo equivalente ao nome do tipo da enumeração. Se o conteúdo do atributo sexo for igual a Sexo.MASCULINO, será armazenado o valor MASCULINO.
- **ORDINAL**: ao adotar esse tipo, o atributo será armazenado como número no banco de dados, de acordo com o índice do valor da enumeração. Se o conteúdo do atributo sexo for igual a Sexo.MASCULINO, será armazenado o valor 0; caso seja Sexo.FEMININO, o valor a ser armazenado será 1, e assim sucessivamente.
- **@Column**: embora já tenhamos utilizado essa anotação, ainda não havíamos abordado as propriedades *length* e *nullable*. A primeira limita a quantidade de caracteres do seu conteúdo e a segunda indica se a coluna aceita valor nulo.

Finalizado o mapeamento objeto-relacional deve-se criar a classe de acesso a dados referente à classe mapeada, a qual será responsável por realizar a interação com o banco de dados. As classes de acesso a dados devem estar agrupadas dentro de um pacote específico, a fim de separarmos a lógica de negócios da lógica de persistência de dados conforme o padrão de projeto DAO (*Data Access Object*).

Crie o pacote `br.com.springboot.dao` dentro da pasta `src/main/java`, clicando com botão direito sobre ela e acessando o menu *New > Package*. Na tela de cadastro de pacote, informe o nome do pacote no campo *Name* e clique no botão *Finish*. Criado o pacote, vamos incluir uma interface com os principais métodos de acesso a dados, popularmente conhecido CRUD (*Create, Read, Update, Delete*), que compreende os métodos de inserção, remoção, leitura e atualização. A essa interface, daremos o nome de CRUDe.

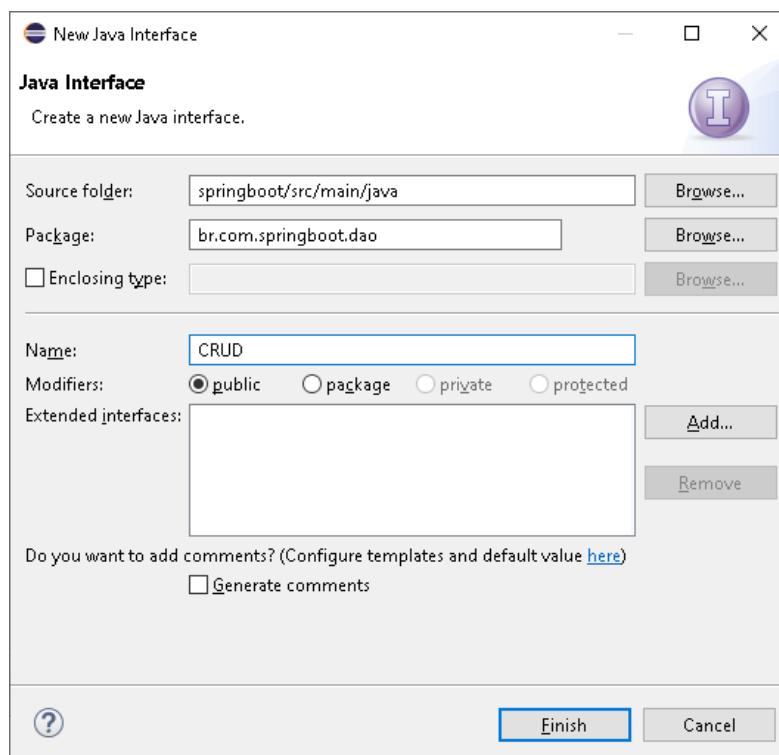
Para criar uma interface, clique com o botão direito sobre o pacote que acabamos de criar e selecione o menu *New > Interface*, conforme mostra a imagem a seguir.

Figura 11 – Menu Interface



Na tela de cadastro de interface, informe o seu nome no campo *Name* e clique no botão *Finish*, conforme a imagem a seguir.

Figura 12 – Tela de cadastro de Interface



Criada a interface CRUD, implemente-a conforme o bloco de comandos do Quadro 13, apresentado na sequência.



## Quadro 12 – Interface CRUD

```
package br.com.springboot.dao;

import java.util.List;

public interface CRUD<T, ID> {
    T pesquisaPeloId(ID id);
    List<T> lista();
    void insere(T t);
    void atualiza(T t);
    void remove(T t);
}
```

A interface CRUD é composta pelos seguintes métodos que contém as seguintes funcionalidades:

- **Método pesquisaPeloid**: responsável retornar o objeto cadastrado no banco de dados cujo id é passado por parâmetro;
- **Método lista**: responsável por retornar todos os registros cadastrados retornando uma lista de objetos;
- **Método insere**: responsável por realizar a inserção do objeto passado por parâmetro no banco de dados;
- **Método atualiza**: responsável por atualizar os dados do objeto passado por parâmetro no banco de dados;
- **Método remove**: responsável por remover o objeto passado por parâmetro no banco de dados.

Implementada a interface CRUD, adicione a classe ClienteDAO ao pacote *br.com.springboot.dao*, clicando com o botão direito sobre ele e acessando o menu *New > Class*. Informe no campo *Name* o nome da classe e clique no botão *Finish*. Criada a classe, essa deverá implementar a interface CRUD, tornando obrigatório a implementação dos métodos dessa interface.

Para implementá-los, iremos utilizar os métodos da classe EntityManager fornecida pela JPA; dentre eles, destacam-se:

- **Método persist**: responsável por realizar a inserção do objeto no SGBD;
- **Método merge**: responsável por realizar a atualização do objeto no SGBD;
- **Método find**: responsável por localizar um objeto pelo id no SGBD;
- **Método createQuery**: responsável por executar uma consulta no SGBD;



- **Método remove:** responsável por remover um objeto no SGBD.

Como o Hibernate irá gerenciar o ciclo de vida das entidades da aplicação, devemos anotar a classe EntityManager com a anotação @PersistenceContext. Finalmente, por se tratar de uma classe de persistência de dados, devemos mapeá-la com a anotação @Repository do Spring, indicando para o *framework* que se trata de uma classe de acesso a dados. A implementação da classe ClienteDAO pode ser visualizada no quadro a seguir.

Quadro 13 – Classe ClienteDAO

```
package br.com.springboot.dao;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.stereotype.Repository;

import br.com.springboot.model.Cliente;

@Repository
public class ClienteDAO implements CRUD<Cliente, Long> {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public Cliente pesquisaPeloId(Long id) {
        return entityManager.find(Cliente.class, id);
    }

    @Override
    public List<Cliente> lista() {
        Query query = entityManager.createQuery("SELECT c FROM Cliente c");
        return (List<Cliente>) query.getResultList();
    }

    @Override
    public void insere(Cliente cliente) {
        entityManager.persist(cliente);
    }

    @Override
    public void atualiza(Cliente cliente) {
        entityManager.merge(cliente);
    }

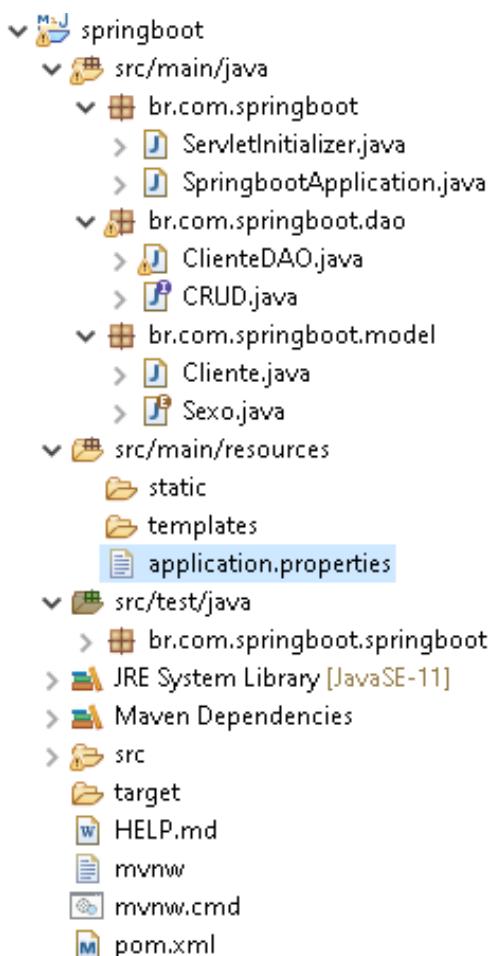
    @Override
    public void remove(Cliente cliente) {
        entityManager.remove(cliente);
    }
}
```



## TEMA 5 – CONFIGURANDO O AMBIENTE DE ACESSO A DADOS

Implementada a classe de acesso a dados, precisamos configurar o projeto para que a aplicação possa efetuar a persistência dos dados; portanto, deve-se especificar como será estabelecida a conexão com o servidor de banco de dados. Para isso, devemos acessar o arquivo de configuração do Spring, no qual são configurados os diversos serviços e recursos utilizados no projeto, bem como parâmetros da própria aplicação. O arquivo de configuração do projeto, denominado *application.properties*, encontra-se dentro da pasta *src/main/resources*, localizada na raiz do projeto, conforme ilustra a Figura 13.

Figura 13 – Arquivo *application.properties*



Nesse arquivo, iremos configurar as propriedades de fonte de dados (*spring.datasource*) e JPA (*spring.jpa*). Na primeira, serão especificados os parâmetros de conexão com o SGDB; na segunda, será definido o *framework* de persistência da aplicação, no caso, o Hibernate. Adicione ao arquivo de configuração do Spring as propriedades listadas no quadro a seguir.



#### Quadro 14 – Propriedades de acesso a dados

```
# Configuração do SGBD
spring.datasource.url=jdbc:mysql://[servidor]:[porta]/[base_dados]
spring.datasource.username=[usuario]
spring.datasource.password=[senha]

# Configuração do Hibernate
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto=[hibernate_schema]
```

Na propriedade `spring.datasource.url`, deve-se especificar como será efetuada a conexão com o banco de dados por meio de uma *string* de conexão, a qual contém as informações referentes ao servidor de banco de dados. Na linguagem Java, comumente se utiliza a JDBC, *api* de interação com o banco de dados, que fornece a seguinte *string* de conexão: `jdbc:[subprotocolo]://[servidor]:[porta]/[base_dados]`. Dentre os parâmetros que a compõem, temos:

- **Subprotocolo:** especifica o driver de conexão, como será utilizado o MySQL como SGBD da aplicação, o termo [subprotocolo] da *string* de conexão deve ser substituído por mysql;
- **Servidor:** especifica o endereço do servidor com o qual a aplicação irá se conectar, devendo o termo [servidor] da *string* de conexão ser substituído pelo IP do servidor no arquivo de configuração;
- **Porta:** especifica o número da porta na qual está rodando o serviço de banco de dados no servidor, devendo o termo [porta] da *string* de conexão ser substituído pelo número da porta no arquivo de configuração;
- **Base\_dados:** especifica o nome da base de dados na qual serão persistidos os dados da aplicação, devendo o termo [base\_dados] da *string* de conexão ser substituído pelo nome da base de dados no arquivo de configuração.

Além da propriedade abordada anteriormente, para estabelecer a conexão com o banco de dados, também é necessário informar o usuário e senha de autenticação. Portanto, deve-se utilizar a propriedade `spring.datasource.username` para definir o usuário e a propriedade `spring.datasource.password` para definir a senha. No arquivo de configuração



do projeto, substitua os termos [usuário] e [senha] respectivamente pelo usuário e a senha que será utilizado para realizar a autenticação.

Configuradas as propriedades de acesso a dados, deve-se configurar na sequência o Hibernate, utilizando para isso as propriedades referentes a JPA. Assim sendo, devemos adicionar ao projeto a propriedade `spring.jpa.database-platform` com o valor `org.hibernate.dialect.MySQL5InnoDBialect`, especificando para o Spring que será utilizado o Hibernate como *framework* de persistência de dados.

Como o Hibernate gera os comandos SQL de forma automática baseado no mapeamento objeto-relacional, vamos adicionar duas propriedades que irão nos permitir os comandos que estão sendo executados quando um método de acesso a dados for invocado. Para tal, são utilizadas as propriedades `spring.jpa.show-sql` e `spring.jpa.properties.hibernate.format_sql` com valor `true`, sendo a primeira responsável por exibir os comandos sql e a segundo por exibi-los formatados.

Outra propriedade que será utilizada no projeto é a `spring.jpa.hibernate.ddl-auto`, a qual especifica como o Hibernate gerencia a base de dados. A essa propriedade, podem ser atribuídos os seguintes valores:

- **Validate**: valida a base de dados; caso haja alguma inconformidade entre a base de dados e o mapeamento objeto-relacional, a aplicação não será inicializada;
- **Update**: atualiza a base de dados, cria e atualiza as tabelas, caso haja alguma inconformidade entre a base de dados e o mapeamento objeto-relacional;
- **Create**: recria a base de dados, apagando os dados armazenados;
- **Create-drop**: recria a base de dados, apagando os dados armazenados e a destrói assim que finaliza a SessionFactory é finalizado.

No arquivo de configuração do projeto, defina o valor `update` para a propriedade `spring.jpa.hibernate.ddl-auto`. Finalizada as configurações, salve o arquivo `application.properties` e execute o projeto. Note que no *log* de inicialização do Spring foi exibido o comando que cria a tabela clientes na base de dados, conforme nos mostra a Figura 14.



Figura 14 – Tabela criada automaticamente pelo Hibernate

```
2022-01-31 03:21:09.498  INFO 4952 --- [ restar
Hibernate:
    create table clientes (
        id bigint not null auto_increment,
        ativo bit not null,
        celular varchar(11),
        cpf varchar(11),
        data_nascimento date,
        email varchar(50),
        nome varchar(50),
        sexo varchar(255),
        telefone varchar(10),
        primary key (id)
    ) engine=InnoDB
2022-01-31 03:21:10.896  INFO 4952 --- [ restar
2022-01-31 03:21:10.906  INFO 4952 --- [ restar
```

Com isso, conseguimos validar que a configuração de acesso a dados está correta, pois a aplicação está acessando o banco de dados, visto que o Hibernate executou o comando de criação da tabela clientes com sucesso.

## FINALIZANDO

Nesta aula, foi iniciado o processo de implementação dos requisitos funcionais do projeto, começando pela codificação da classe Cliente. Para realizar a persistência de dados dessa classe, foi necessário adicionar duas dependências ao projeto: a *MySQL Connector/J* responsável por estabelecer a conexão com o banco de dados, e a *Spring Boot Starter Data JPA* para realizar o mapeamento objeto-relacional.

Feito isso, foi realizado o mapeamento objeto-relacional da classe Cliente e criada a classe de acesso a dados ClienteDAO, responsável por efetuar a persistência de dados dos objetos da classe Cliente. Na sequência, especificamos os parâmetros de acesso ao banco de dados e definimos o Hibernate como *framework* de persistência de dados, tornando assim a aplicação apta a estabelecer conexão com o banco de dados e realizar a persistência dos objetos.



# **DESENVOLVIMENTO WEB BACK END**

AULA 4

Prof. Rafael Moraes



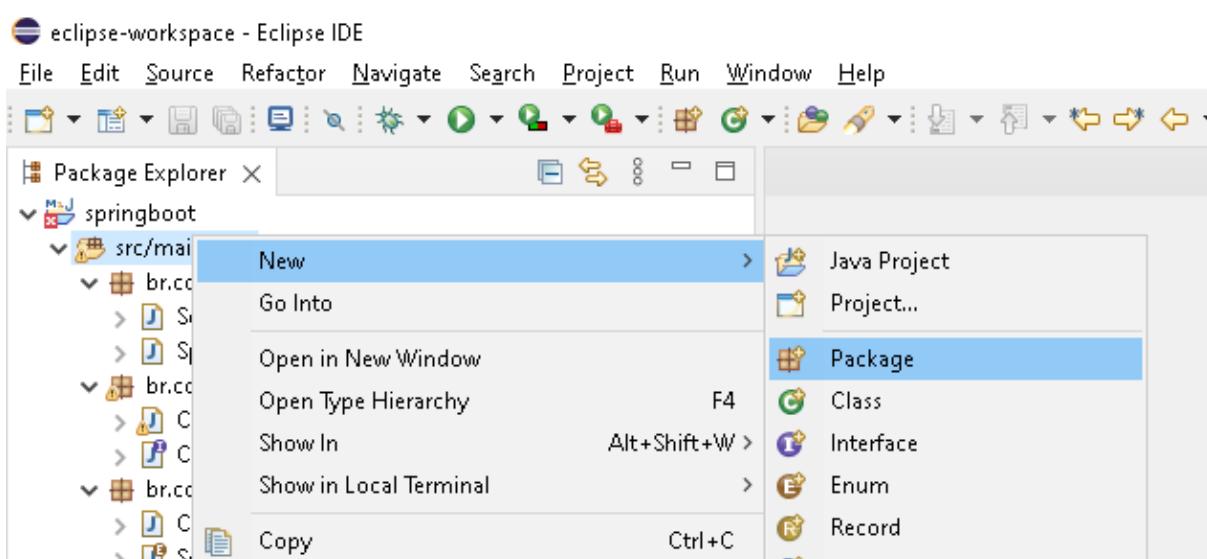
## CONVERSA INICIAL

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

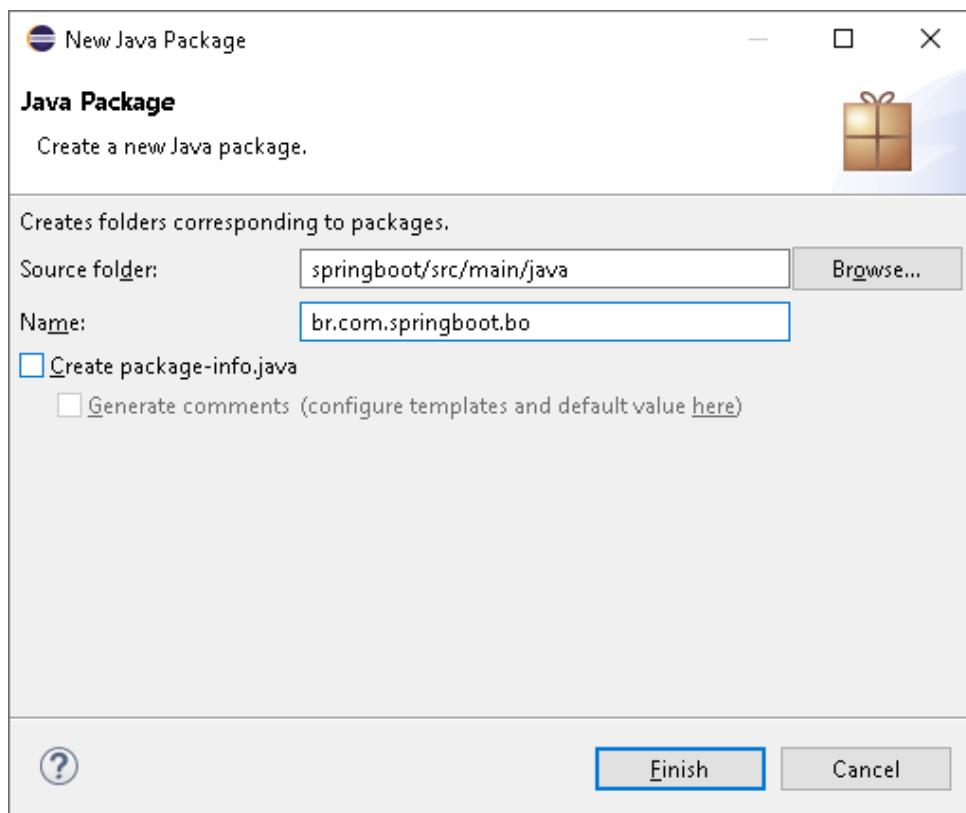
Dando sequência ao processo de desenvolvimento da aplicação, devemos implementar a classe de negócio referente à classe Cliente, responsável pelo encapsulamento das regras de negócio do objeto, conforme o padrão de projeto BO (*Business Object*). Portanto, todos os serviços devem ser implementados dentro 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



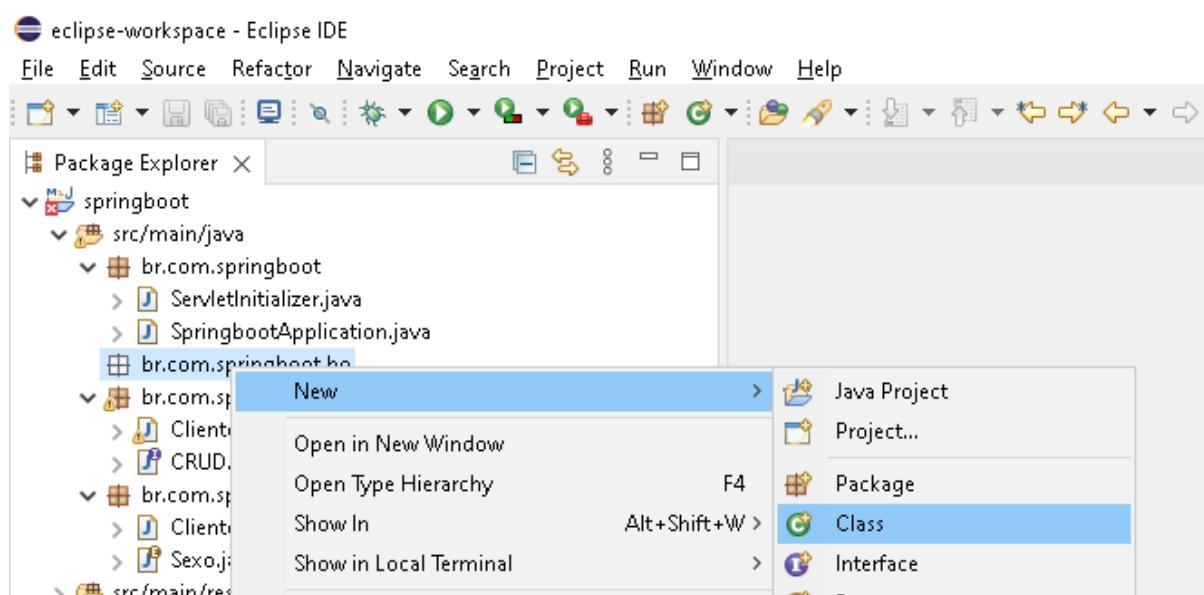
Na tela de cadastro de pacote, informe o nome do pacote no campo *Name* e clique no botão *Finish*, conforme a figura a seguir:

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:

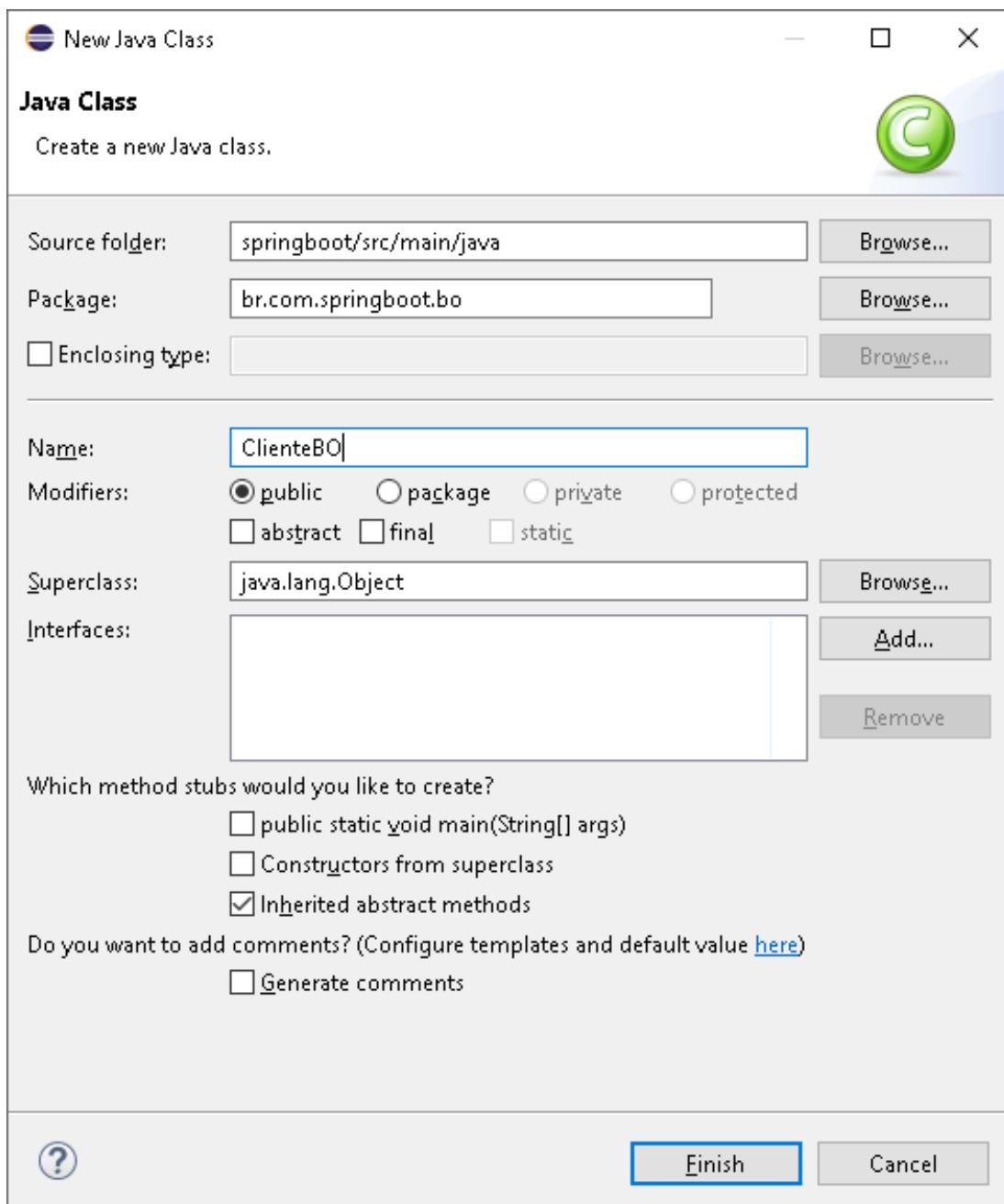
Figura 3 – Menu para adicionar uma classe



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



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 instancia 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.

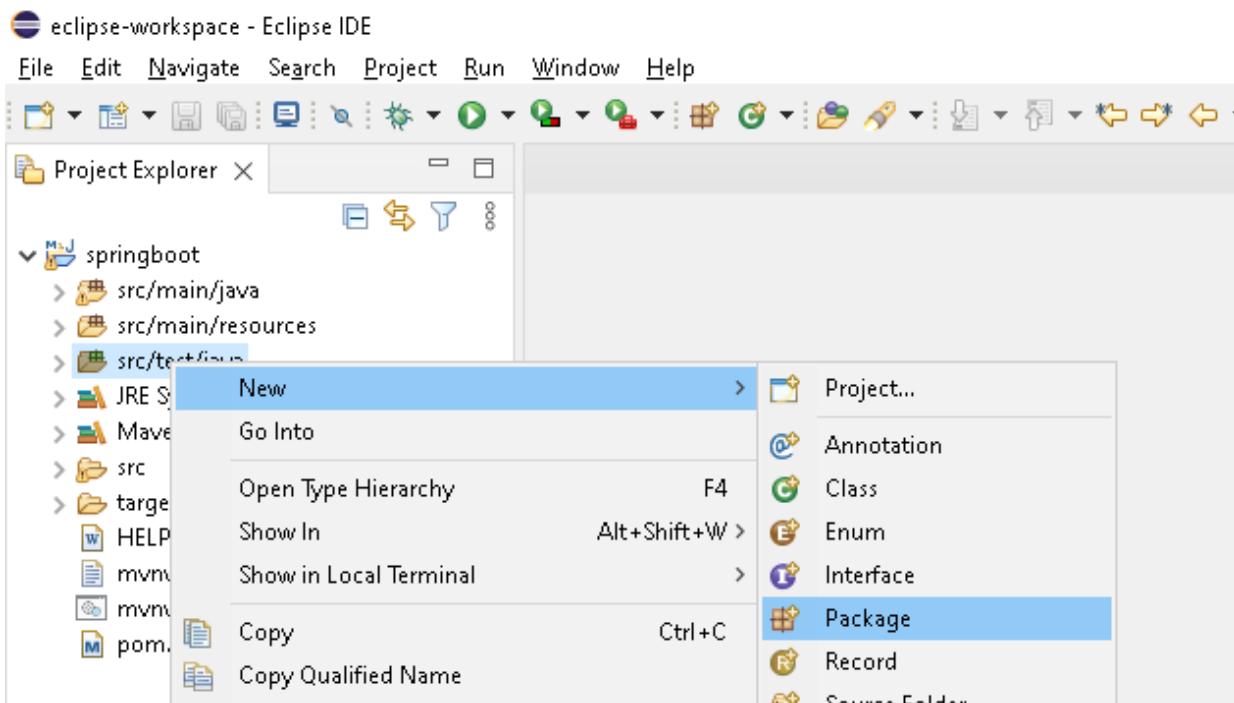


## TEMA 2 – CRIANDO A CLASSE TESTE

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:

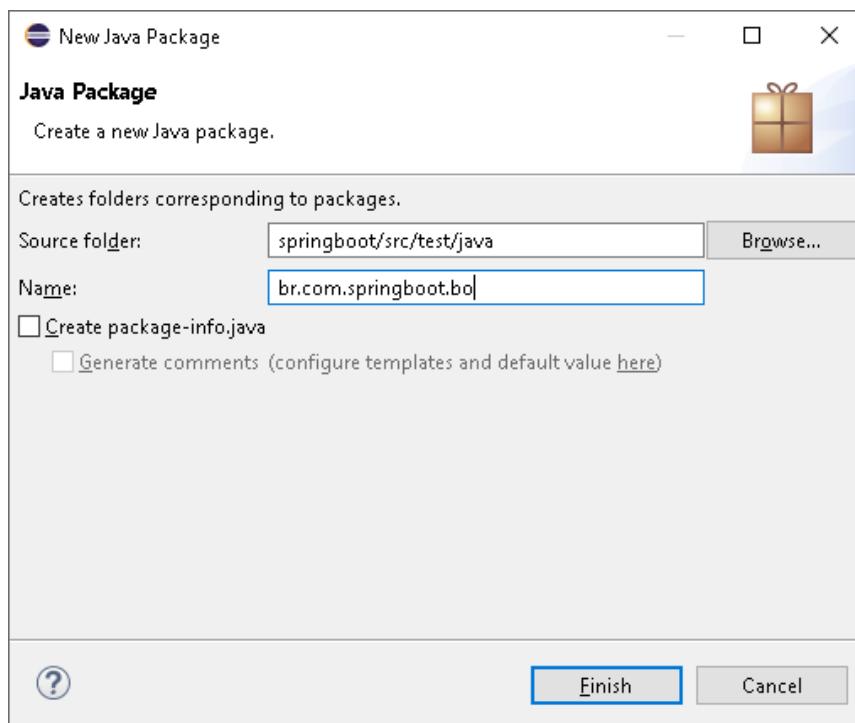
Figura 5 – Menu para criação do pacote



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

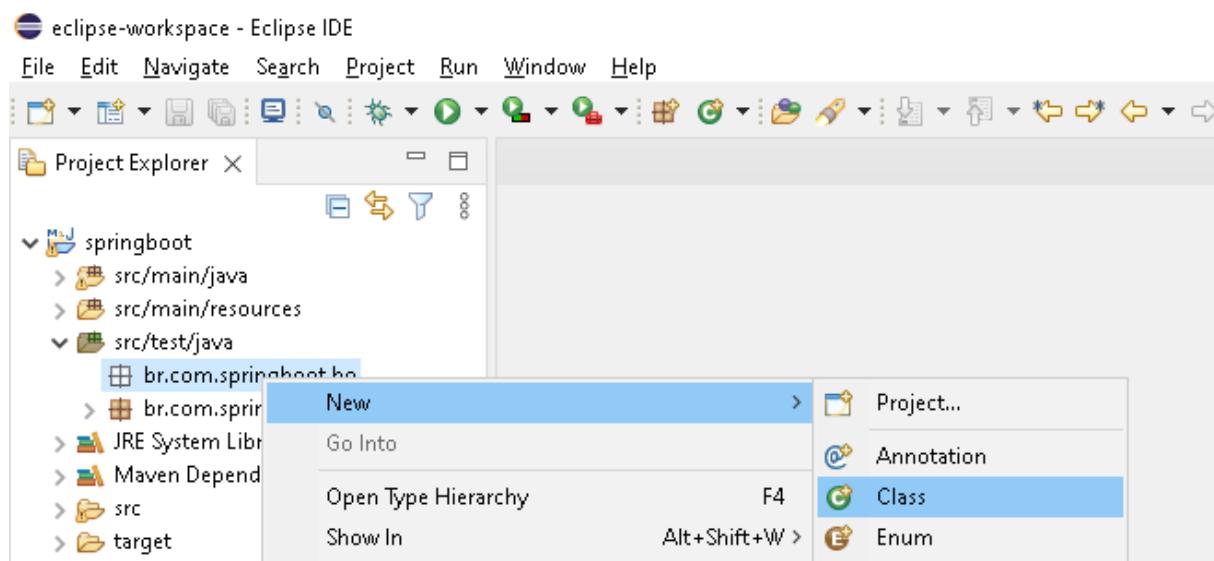


Figura 6 – Criação do pacote br.com.springboot.bo



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:

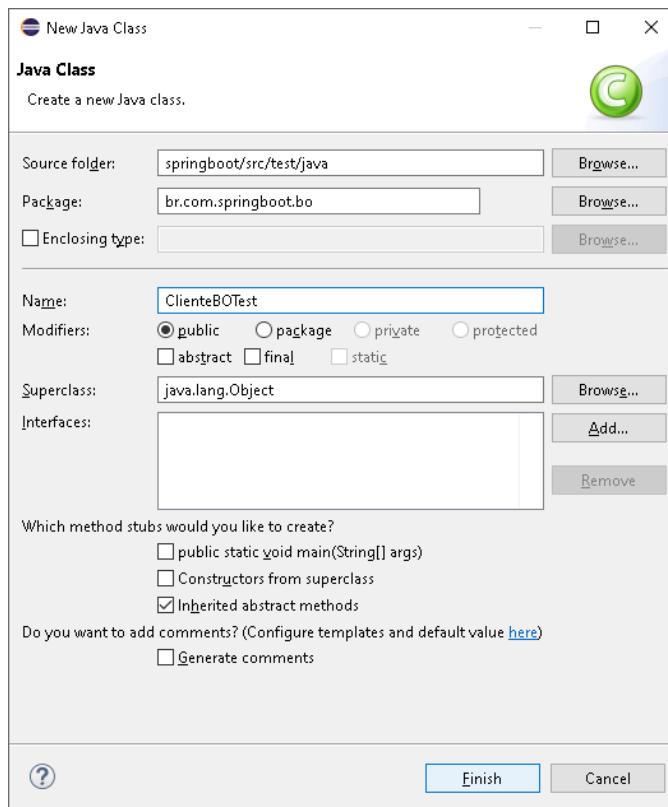
Figura 7 – Menu para criação da classe



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("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

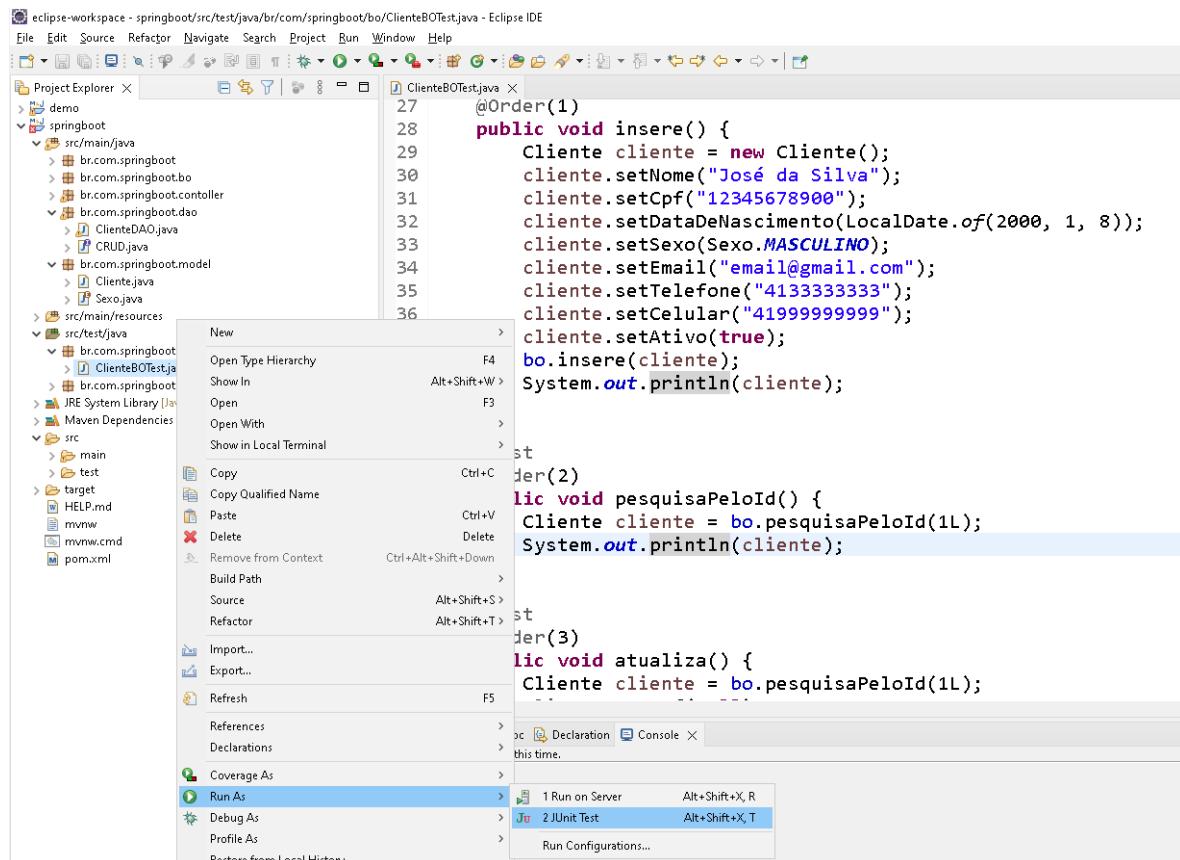
Antes de iniciarmos os testes, vamos adicionar o método `toString` à classe `Cliente`. Esse método irá formatar o objeto sempre que tentarmos convertê-lo para `String`. Dessa forma, podemos utilizar o comando `System.out.print` para imprimir o conteúdo desse objeto. O método `toString` pode ser verificado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

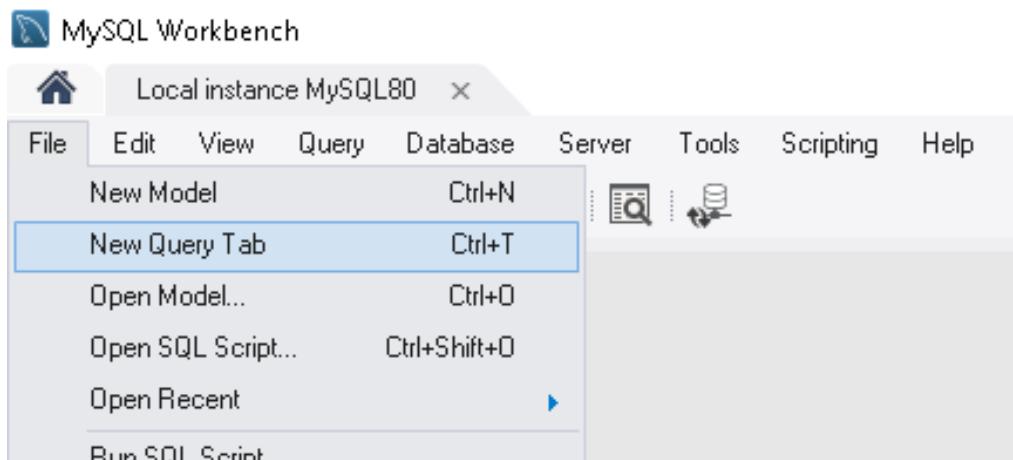


pela anotação @Order. O primeiro método a ser executado, conforme visto no tema anterior, é o método insere. Esse método irá inserir um registro na tabela clientes com as seguintes informações:

- 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

	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 pesquisaPelold, deve-se passar pelo parâmetro desse id, a fim de que seja retornado os dados desse cadastro. Ao executar o método pesquisaPelold, será apresentada a seguinte mensagem no console do Eclipse:

Figura 12 – Dados do objeto retornado pelo método pesquisaPelold

```

Problems @ Javadoc Declaration Console
terminated> ClienteBOTest [JUnit] C:\Users\vmor\p2\pool\plugins\org.eclipse.jdt\openjdk.hotspot.jre.full.win32.x86_64.17.0.0.v2021012-1059\jre\bin\javaw.exe
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

```

Problems JavaDoc Declaration Console X
terminated@ClientB0Test [JUnit] C:\Users\jvmonr.p2\pool\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64.17.0.0.v20211012-1059\jre\bin\java.exe
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 cliente0_
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:

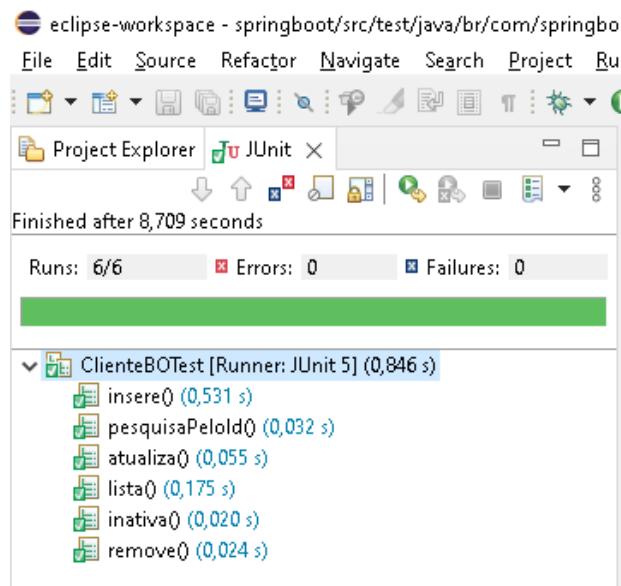
Figura 16 – Registro removido

The screenshot shows a MySQL Workbench interface with a 'Result Grid' tab selected. The grid has columns for id, ativo, celular, cpf, data\_nascimento, email, nome, sexo, and telefone. A single row is present, marked with an asterisk (\*), and all its values are set to NULL.

	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.

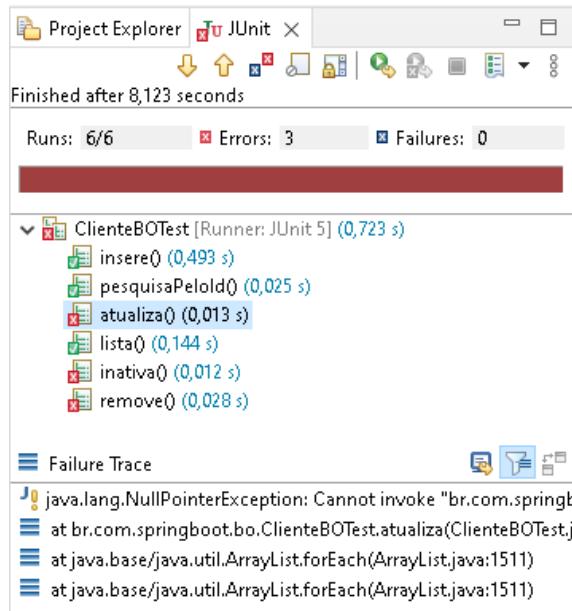
Figura 17 – Aba JUnit



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:

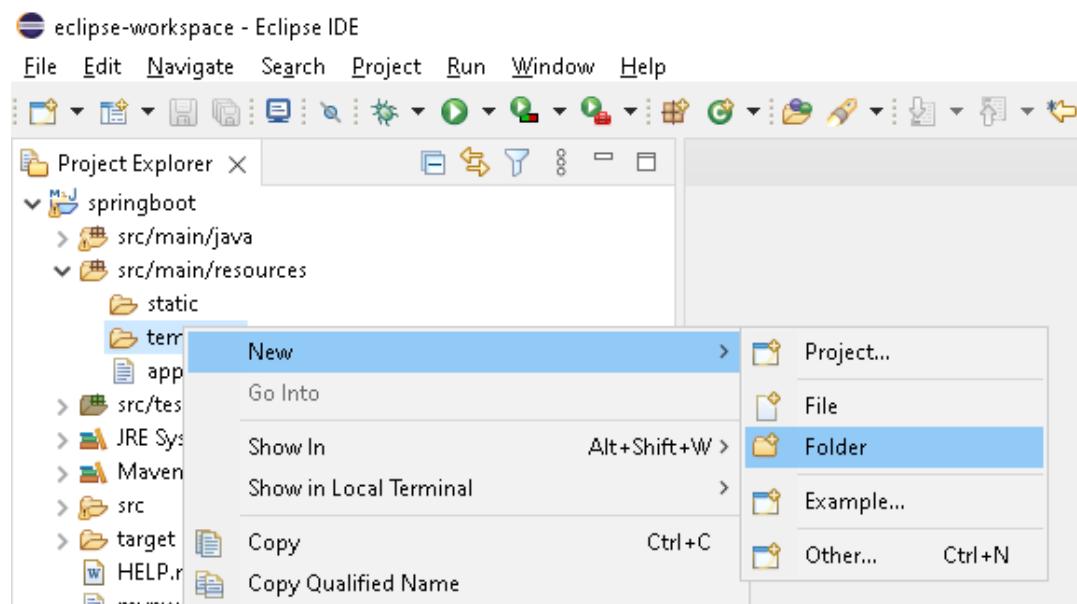
Quadro 11 – Dependência do Thymeleaf

```
<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:

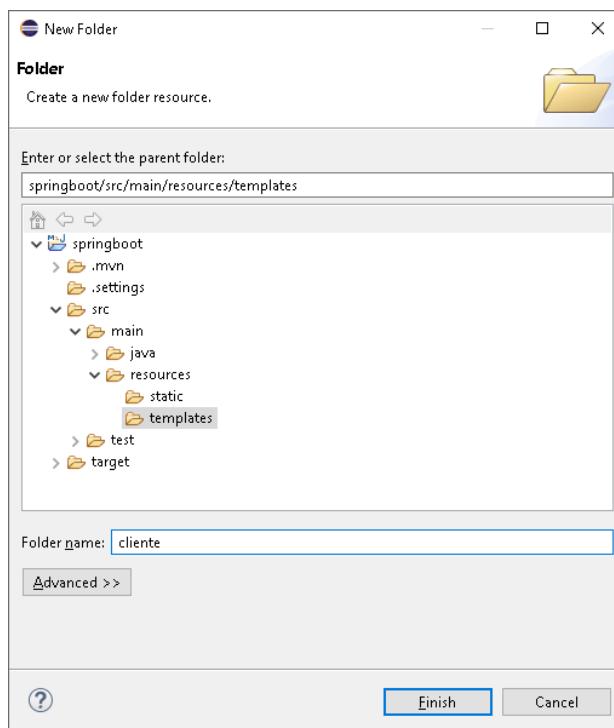


Figura 19 – Menu para criar uma pasta



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.

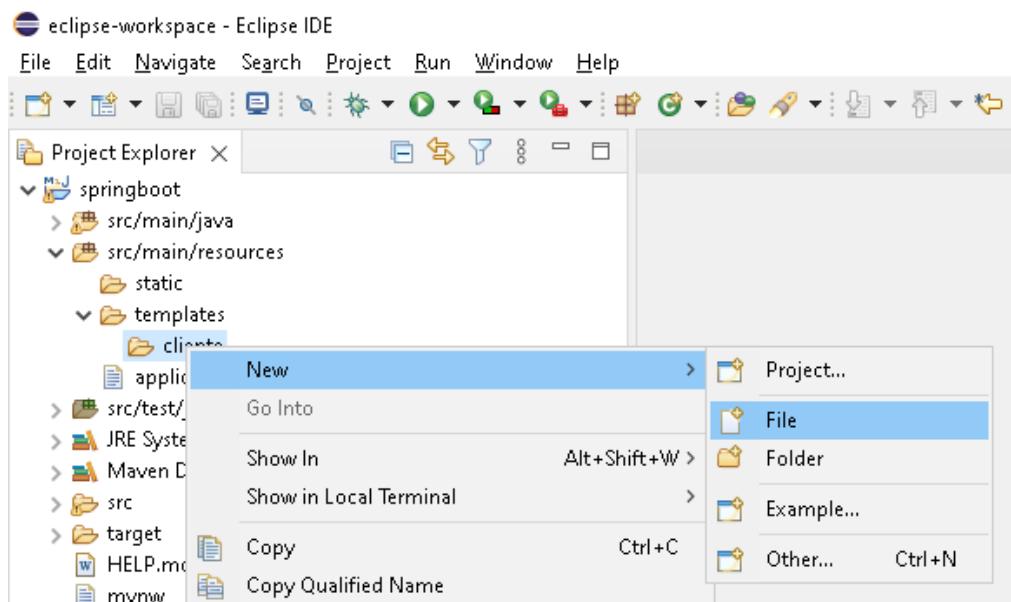
Figura 20 – Tela de criação da pasta



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.

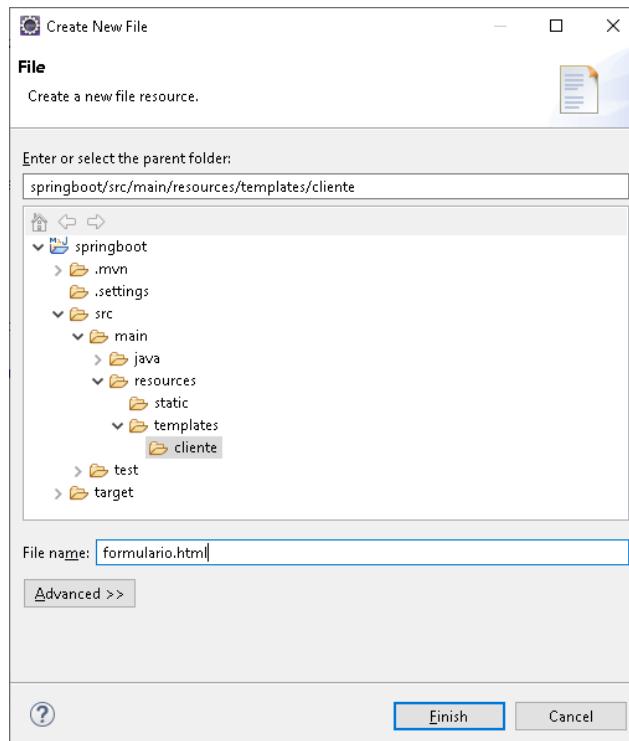


Figura 21 – Adicionando um arquivo à pasta



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
<!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

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

The screenshot shows a browser window with the title "HTML - Campos de entrada". The main content is a heading "Formulário" followed by a horizontal line. Below the line are four input fields: a checkbox labeled "Checkbox" which is unchecked; a text input labeled "Texto" which is empty; a date input labeled "Data" with the placeholder "dd/mm/aaaa" and a calendar icon; and a dropdown menu labeled "Combobox" with the value "Tipo 1". At the bottom is a "Salvar" button.

## 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

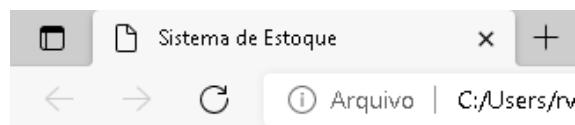
```
<input id="ativo" type="hidden" th:field="*{id}" />
<div>
    <label for="ativo">Registro ativo</label>
    <input id="ativo" type="checkbox" th:field="*{ativo}" />
</div>
<div>
    <label for="nome">Nome</label>
    <input id="nome" type="text" th:field="*{nome}" />
</div>
<div>
    <label for="cpf">CPF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



## Dados do Cliente

---

Registro ativo

Nome

CPF

Data de Nascimento  dd/mm/aaaa

Sexo

Telefone

Celular

E-mail

## 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 registros 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.



# **DESENVOLVIMENTO WEB**

## **BACK END**

AULA 5

Prof. Rafael Moraes



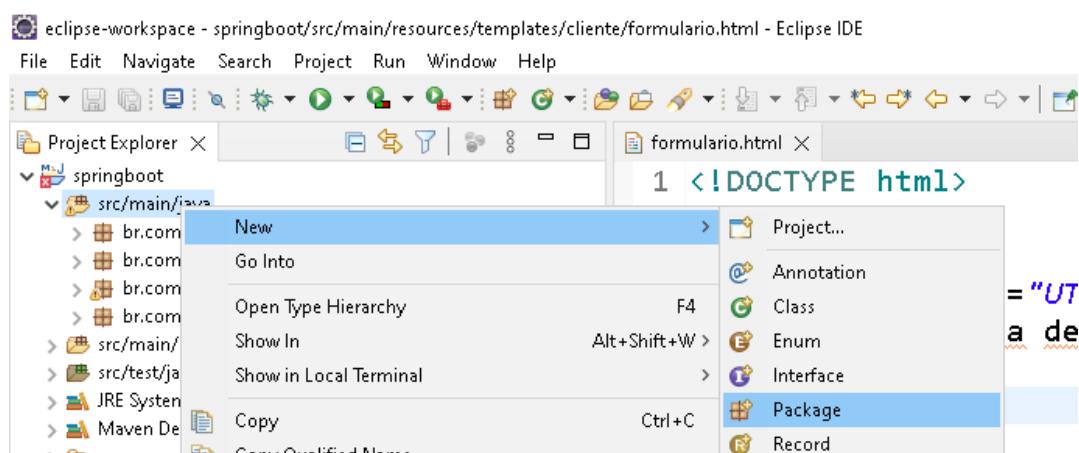
## CONVERSA INICIAL

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:

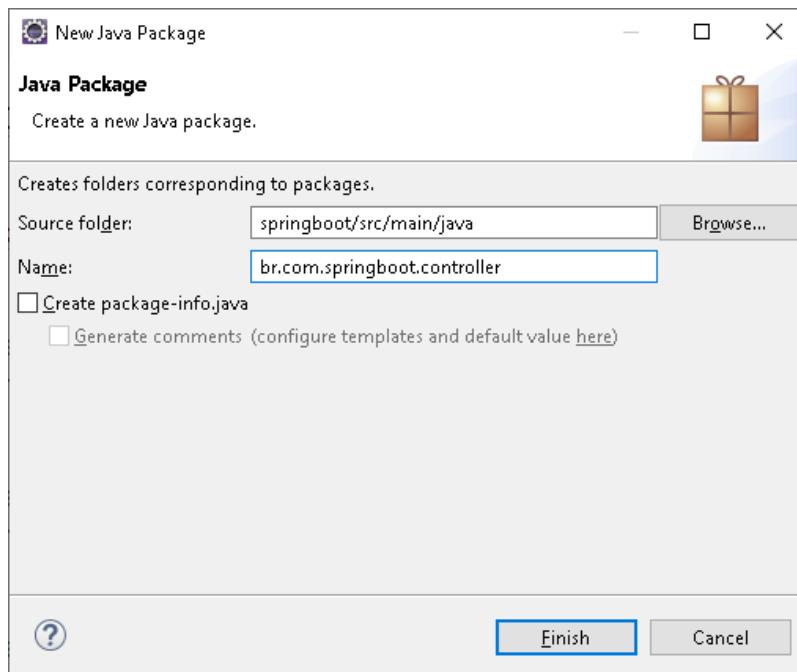
Figura 1 – Menu para criação do pacote



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:

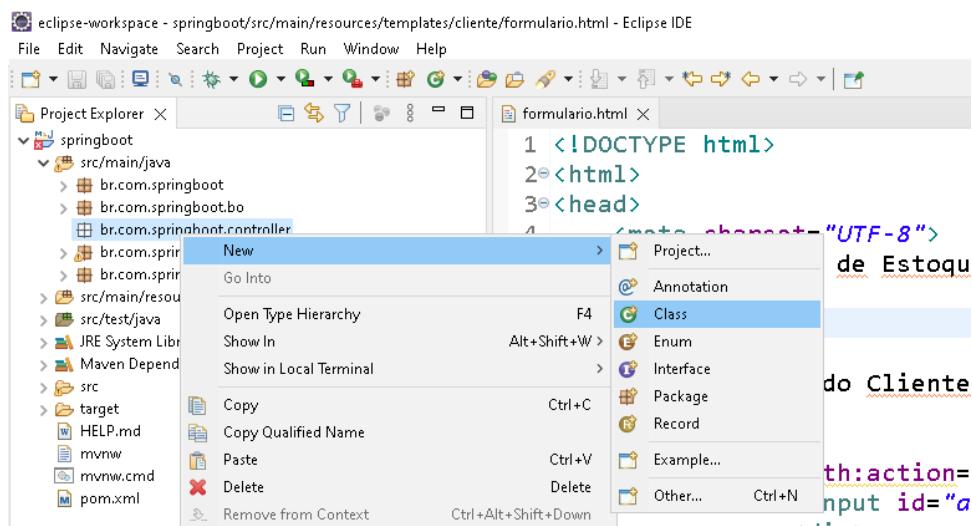


Figura 2 – Tela de criação do pacote



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:

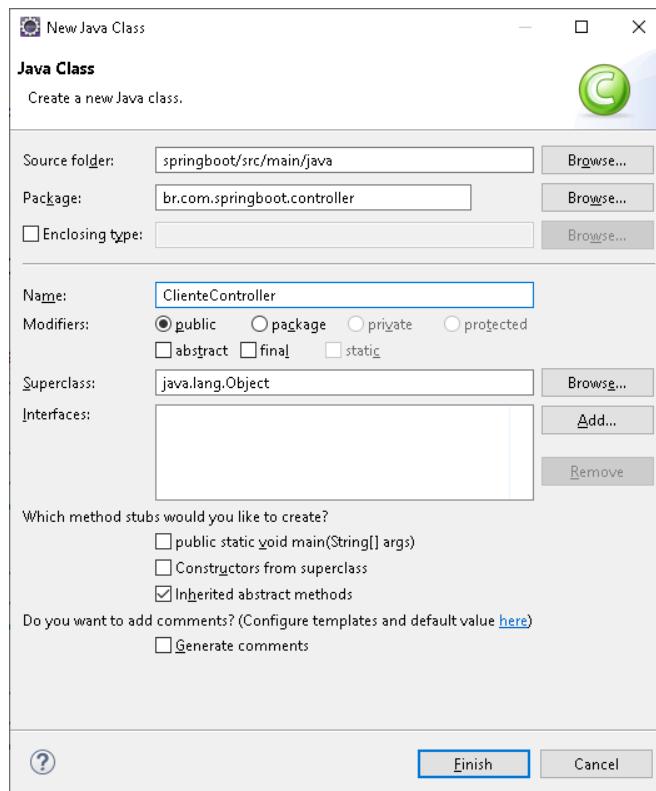
Figura 3 – Menu para criação da classe



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



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

```
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 <Erro! A referência de hiperlink não é válida.<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

```
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);
    }
}
```

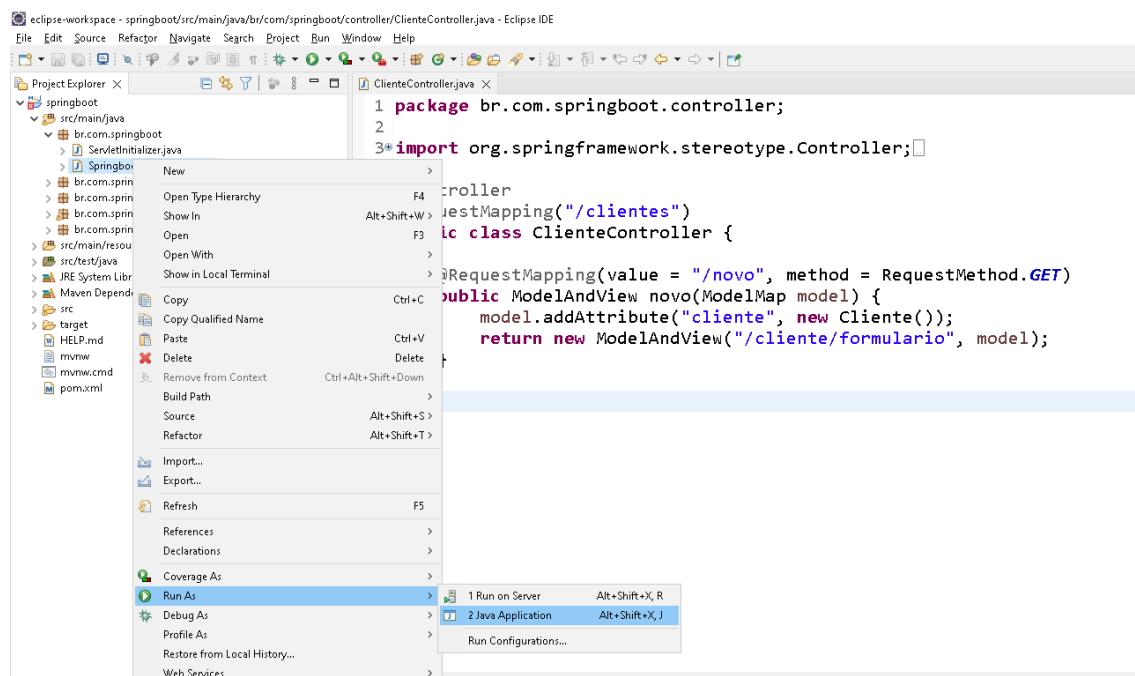


O método novo irá retornar um objeto do tipo ModelAndView, ou seja, um objeto com a página Web (view) e os dados que serão populados na tela (model). A página Web que será exibida no navegador é o formulário de cadastro do cliente, portanto, no primeiro parâmetro do objeto ModelAndView, deve-se informar 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 <Erro! A referência de hiperlink não é válida.<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

Dados do Cliente

Registro ativo

Nome

CPF

Data de Nascimento  dd/mm/aaaa

Sexo

Telefone

Celular

E-mail

## 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

```
<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 <Erro! A referência de hiperlink não é válida.<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

```
@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

```
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 <Erro! A referência de hiperlink não é válida.<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

Registro ativo

Nome

CPF

Data de Nascimento

Sexo

Telefone

Celular

E-mail

Figura 8 – Registro persistido

```
1 • select * from clientes;
2
```

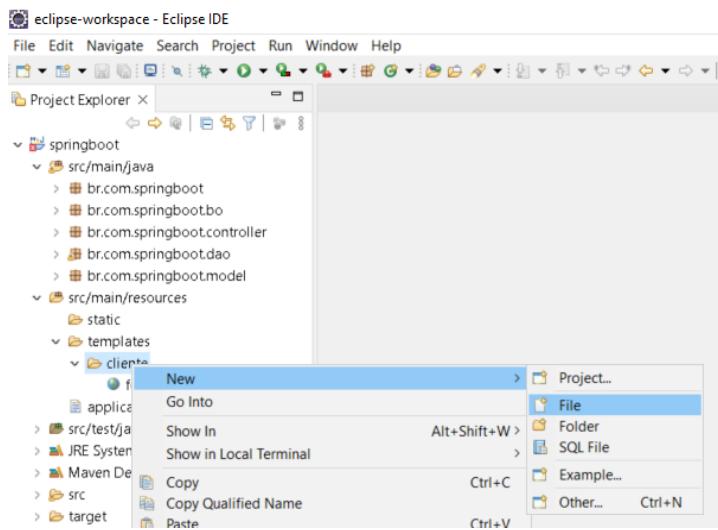
	id	ativo	celular	cpf	data_nascimento	email	nome	sexo	telefone
▶	1	1	41988887777	98765432100	2001-01-16	mariadasilva@outlook.com	Maria da Silva	FEMININO	4133334444
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

## 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:

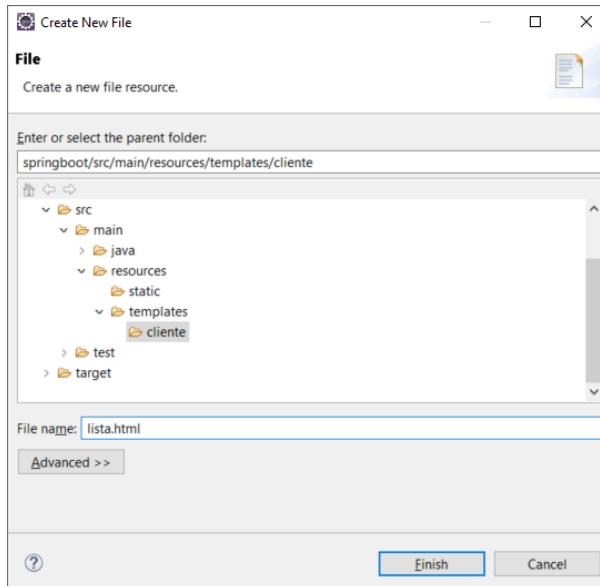


Figura 9 – Adicionar um novo arquivo



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 seguir:

Figura 10 – Tela de criação do arquivo



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

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Sistema de Estoque</title>
</head>
<body>
    <div>
        <h1>Clientes</h1>
        <hr>
        <div>
            <a th:href="@{/clientes/novo}">Novo</a>
        </div>
        <hr>
        <table>
        </table>
    </div>
</body>
</html>
```

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*

```
<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>
```



```
<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 **<Erro! A referência de hiperlink não é válida.>** 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

```
@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 <Erro! A referência de hyperlink não é válida.<http://localhost:8080/clientes/edita/{id}>

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

```
@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 <Erro! A referência de hyperlink não é válida.<http://localhost:8080/clientes/inativa/{id}>>. No quadro a seguir temos a implementação do método inativa:



#### Quadro 10 – Método inativa da classe ClienteController

```
@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";
}
```

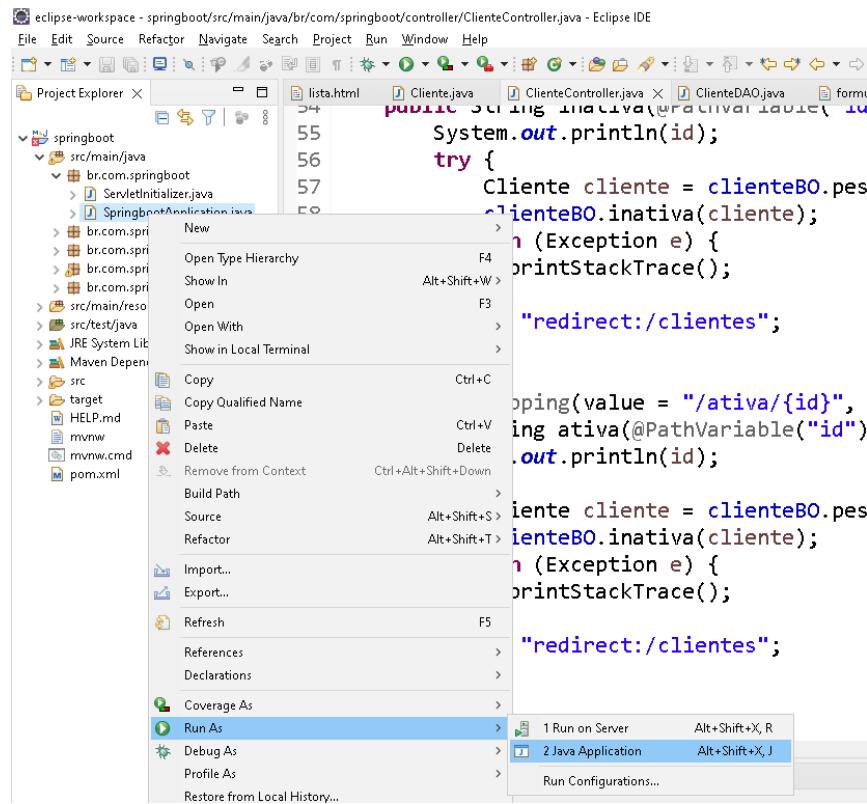
Repare que o retorno desse método não é um objeto do tipo ModelAndView, e sim uma String, indicando que, após inativar o registro, a aplicação será redirecionada para a tela de gerenciamento de cadastro do cliente. Vamos adotar essa mesma abordagem para o método salva, dessa forma, sempre que o usuário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

#### Quadro 11 – Método salva da classe ClienteController

```
@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:

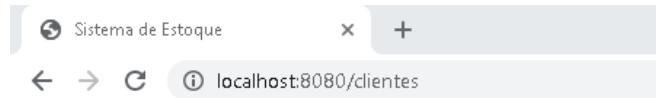
Figura 11 – Iniciando a aplicação



Assim que a aplicação for iniciada, abra o navegador e accese a url <Erro! A referêncie de hiperlink não é válida.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



## Clientes

[Novo](#)

NOME	DATA NASCIMENTO	CPF
------	-----------------	-----

Maria da Silva	2001-01-16	98765432100
----------------	------------	-------------

<a href="#">Editar</a>	<a href="#">Inativar</a>
------------------------	--------------------------

Pedro da Silva	1990-08-16	12345678900
----------------	------------	-------------

<a href="#">Editar</a>	<a href="#">Inativar</a>
------------------------	--------------------------

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.

- **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

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFLdvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyL2QvZ6jIW3" crossorigin="anonymous">
```

#### Quadro 13 – Arquivo de script CDN

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ka7Sk0GLn4gmtz2MLQnikT1wXgYsOg+OMhuP+ILRH9sENB00LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>
```

Adicionados os arquivos às páginas, basta aplicar aos componentes HTML às classes do Bootstrap para estilizá-los. Por meio do site <<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

The figure shows a clean, modern form layout. At the top is a field labeled "Email address" with a placeholder message: "We'll never share your email with anyone else.". Below it is a "Password" field. Underneath the password field is a checkbox labeled "Check me out". At the bottom is a prominent blue "Submit" button.



Quadro 14 – Código fonte do formulário estilizado com Bootstrap

```
<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
<b>container</b>	Classe que centraliza o formulário com espaçamentos laterais
<b>mb</b>	Classe que define o espaçamento entre elementos
<b>form-check</b>	Classe para estilizar o elemento que contém a label e o campo de entrada do tipo checkbox
<b>form-switch</b>	Classe para estilizar o checkbox no formato switch
<b>form-check-label</b>	Classe para estilizar a label do checkbox
<b>form-check-input</b>	Classe para estilizar um campo de entrada do tipo checkbox



<b>form-control</b>	Classe para estilizar um campo de entrada do tipo texto
<b>row</b>	Define o elemento com uma linha, tornando possível definir o tamanho e demais elementos que compõem essa linha
<b>col</b>	Define o tamanho de um elemento dentro de uma linha
<b>btn</b>	Deixa o elemento com formato de um botão, podendo essa classe ser aplicada a links
<b>btn-primary</b>	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

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Sistema de Estoque</title>

    <link
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
        rel="stylesheet"
        integrity="sha384-1BmE4kWBq78iYhFLdvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyL2QvZ6jIW3"
        crossorigin="anonymous">
</head>
<body>
    <div class="container">
        <h1>Dados do Cliente</h1>
        <hr>
        <div>
            <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}" />
</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}" />
    </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}" />
    </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}" />
    </div>
</div>
<div class="mb-3">
    <input class="btn btn-primary" type="submit" value="Salvar" />
</div>
</form>
</div>
</div>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ka7Sk0GLn4gmtz2MLQnikT1wXgYs0g+OMhuP+ILRH9sENBO0LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>
</body>
</html>
```



Figura 14 – Interface do formulário de cadastro do cliente

### Dados do Cliente

Registro ativo

Nome

CPF  
 Data de Nascimento  
 dd/mm/aaaa

Sexo  
 Masculino

Telefone

Celular

E-mail

## 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
<b>table</b>	Classe para estilizar uma tabela
<b>table-hover</b>	Destaca a linha da tabela na qual o cursor está posicionado
<b>btn-secondary</b>	Deixa o botão na cor cinza (botão secundário da tela)
<b>btn-sm</b>	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

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Sistema de Estoque</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFLdvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyL2QvZ6jIW3" crossorigin="anonymous">
</head>
<body>
    <div class="container">
        <h1>Clientes</h1>
        <hr>
        <div>
            <a class="btn btn-primary" th:href="@{/clientes/novo}">Novo</a>
        </div>
        <hr>
```

```
<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

## Clients

Clients				
Novo				
NOME	DATA NASCIMENTO	CPF		
Maria da Silva	2001-01-16	98765432100	<button>Editar</button>	<button>Inativar</button>
Pedro da Silva	1990-08-16	12345678900	<button>Editar</button>	<button>Inativar</button>

## 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.



## REFERÊNCIAS

THYMELEAF. **Tutorial: Using Thymeleaf.** Disponível em: <<https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html>>. Acesso em: 24 mar. 2022.

BOOTSTRAP. **Introduction.** Disponível em: <<https://getbootstrap.com/docs/5.1/getting-started/introduction/>>. Acesso em: 24 mar. 2022.



# **DESENVOLVIMENTO WEB**

## **BACK END**

AULA 6

Prof. Rafael Veiga de Moraes



## CONVERSA INICIAL

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

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

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

## TEMA 1 – VALIDANDO OS DADOS DO FORMULÁRIO

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

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



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

Quadro 1 – Dependência de validação

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

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

Quadro 2 – Anotações para validação dos atributos

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

Na classe Cliente, serão validados os seguintes atributos:

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

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



### Quadro 3 – Atributos da classe Cliente

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

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

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



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

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

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

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

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

Cada span irá contar com os seguintes atributos do Thymeleaf:

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

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



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

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

```

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

```

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

Figura 1 – Validação do formulário de cadastro

## Dados do Cliente

The screenshot shows a client registration form with the following fields and their current states:

- Nome:** An empty input field.
- CPF:** An empty input field with the error message "CPF inválido".
- Data de Nascimento:** An empty input field with the placeholder "dd/mm/aaaa" and the error message "informe a data de nascimento".
- Sexo:** A dropdown menu set to "Masculino".
- Telefone:** An empty input field.
- Celular:** An empty input field.
- E-mail:** An empty input field with the placeholder "email" and the error message "e-mail inválido".

At the bottom left is a blue "Salvar" button.



## TEMA 2 – FORMATANDO OS DADOS DO FORMULÁRIO

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

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

Quadro 6 – Script JQuery

```
<script      crossorigin="anonymous"      referrerpolicy="no-referrer"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"
integrity="sha512-
894YE6QWD5I59HgZOGReFYm4dnWc1Qt5NtvYSaNcOP+u1T9qYdvdihz0PPSiiqn/+/3e7Jo4EaG
7TubfWGUrMQ=="></script>
<script      crossorigin="anonymous"      referrerpolicy="no-referrer"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery.maskedinput/1.4.1/jquery
.maskedinput.min.js"                      integrity="sha512-
d4KkQohk+HswGs6A1d6Gak6Bb9rMltxj0a0IiY49Q3TeFd5xAzjWXDCBW9RS7m86FQ4RzM2BdHm
dJnnKRYknxw=="></script>
```

Dentre as bibliotecas de script adicionadas temos:

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



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

#### Quadro 7 – Declaração do campo CPF

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

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

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

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

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

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

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



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

Figura 2 – Formatação dos campos do formulário

## Dados do Cliente

Registro ativo

Nome

CPF 999.999.999-99	Data de Nascimento dd/mm/aaaa	Sexo Selecione uma opção
-----------------------	----------------------------------	-----------------------------

Telefone (99) 9999-9999	Celular (99) 99999-9999	E-mail <input type="text"/>
----------------------------	----------------------------	--------------------------------

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

Quadro 10 – Ajuste no mapeamento objeto-relacional

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

@Column(length = 14)
private String telefone;

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

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

## TEMA 3 – FEEDBACK PARA O USUÁRIO

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

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

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

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

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

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

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

Quadro 11 – Métodos com feedback para o usuário

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

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

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

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

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

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

Quadro 12 – Métodos com feedback para o usuário

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



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

Figura 3 – Feedback para o usuário

## Clientes

Clientes			
NOME	DATA NASCIMENTO	CPF	ACTION
José da Silva	01/01/2000	288.901.820-24	<button>Editar</button> <button>Inativar</button>

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

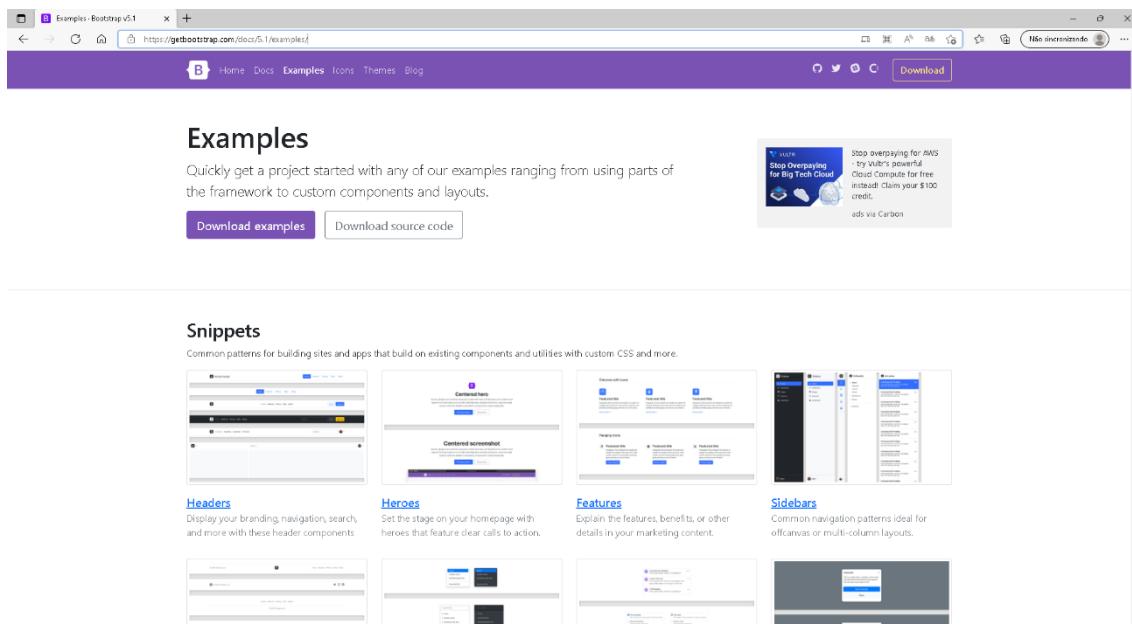
## TEMA 4 – CRIANDO A TELA INICIAL DO SISTEMA

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

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

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

Figura 4 – Templates do Bootstrap



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

## Quadro 13 – Página inicial da aplicação

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title>Sistema de Estoque</title>
        <link href="https://getbootstrap.com/docs/5.1/examples/navbar-fixed/" rel="canonical" />
        <link rel="stylesheet" crossorigin="anonymous" href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" />
        <style> body { padding-top: 4.5rem; } </style>
    </head>
    <body>
        <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
            <div class="container-fluid">
                <a class="navbar-brand" href="#">Sistema de Estoque</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse" aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="collapse navbar-collapse" id="navbarCollapse">
                    <ul class="navbar-nav me-auto mb-2 mb-md-0">
                        <li class="nav-item">
                            <a class="nav-link" href="/produtos">Produto</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="/clientes">Cliente</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="/fornecedores">Fornecedor</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="/nota-entrada">Nota de entrada</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="/nota-saida">Nota de saída</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="/estoque">Estoque</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
        <div class="container">
            <div class="bg-light p-5 rounded">
                <h1>Página inicial</h1>
                <p class="lead">Seja bem-vindo ao sistema de estoque</p>
            </div>
        </div>
    </body>
</html>
```

```

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/
    bootstrap.bundle.min.js"
    integrity="sha384-ka7Sk0Gln4gmtz2MLQnikT1wXgYsOg+OMhuP+ILRH
    9sENB00LRn5q+8nbTov4+1p"
    crossorigin="anonymous"></script>
</body>
</html>

```

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

Figura 5 – Pasta templates



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

Quadro 14 – Página inicial da aplicação

```

package br.com.springboot.controller;

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

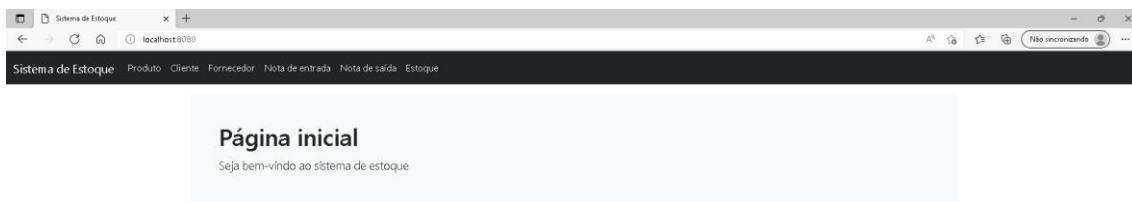
@Controller
public class AplicacaoController {
    @RequestMapping (value = {"", "/"}, method = RequestMethod.GET)
    public String index() {
        return "index";
    }
}

```



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

Figura 6 – Tela inicial da aplicação



## TEMA 5 – REALIZANDO O LOGIN NA APLICAÇÃO

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

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

Quadro 15 – Dependência do Spring Security

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

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



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

#### Quadro 16 – Classe SecurityConfiguration

```
package br.com.springboot;

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

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

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

#### Quadro 17 – Método passwordEncoder

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

Como ainda não temos uma classe de usuários e tampouco uma tabela de usuários no banco de dados, vamos criar dois usuários com permissões de acesso diferentes em memória. O primeiro poderá acessar o sistema utilizando o usuário `admin`, senha `12345` e permissão de administrador, e o segundo com o usuário `user`, senha `12345` e permissão de usuário. Para criar ambos os usuários em memória, iremos sobreescrivendo 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 sobreescriver o método configure das requisições http. Vamos adotar que somente o usuário admin poderá acessar os menus Nota de entrada, Nota de saída e Estoque. Para garantir que somente esse usuário possa acessar essas telas, devemos configurar que as URLs referentes a esses menus sejam acessadas somente por quem tem a permissão de Administrador. Para isso, vamos utilizar três métodos do objeto http fornecido pelo método configure:

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

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

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

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



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

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

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

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

Quadro 19 – Método configure referente as requisições http

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

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



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

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

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

## Quadro 21 – Formulário de login

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title>Sistema de estoque</title>
        <link rel="stylesheet" crossorigin="anonymous"
              integrity="sha384-1BmE4kBq78iYhFLdvKuhfTAU6auU8tT94WrHftjD
              brCEXSU1oBoqyl2QvZ6jIW3"
              href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/
              bootstrap.min.css" >
    </head>
    <body>
        <div class="container">
            <form th:action="@{/Login}" method="POST">
```

```

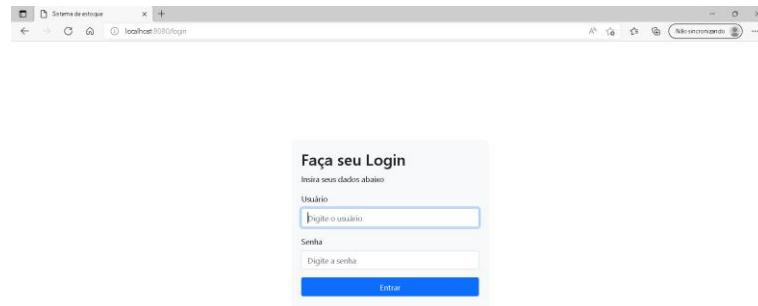
<h2>Faça seu Login</h2>
<p>Insira seus dados abaixo</p>
<div class="mb-3">
    <label class="form-label" for="username">Usuário</label>
    <input type="text" id="username" name="username"
           class="form-control" placeholder="Digite o usuário"
           autofocus />
</div>
<div class="mb-3">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

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

```

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

Figura 7 – Tela de login da aplicação





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

## FINALIZANDO

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

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

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