DESENVOLVIMENTO WEB BACK END

AULA 3

Prof. Rafael Veiga de Moraes

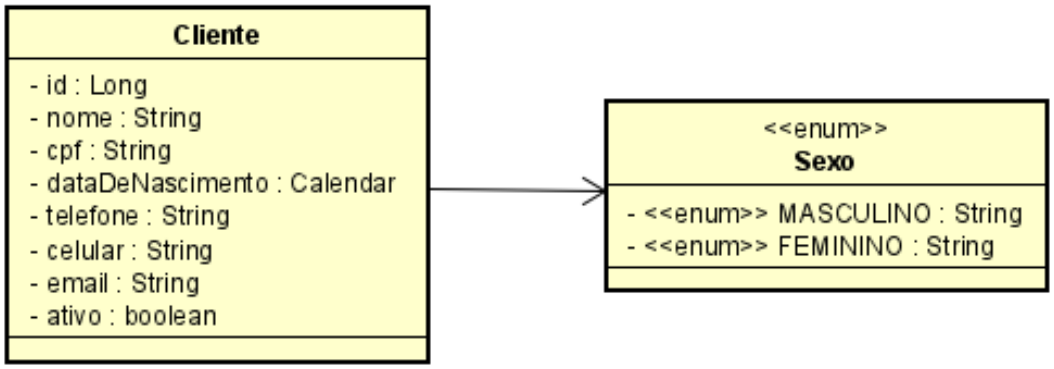
CONVERSA INICIAL

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

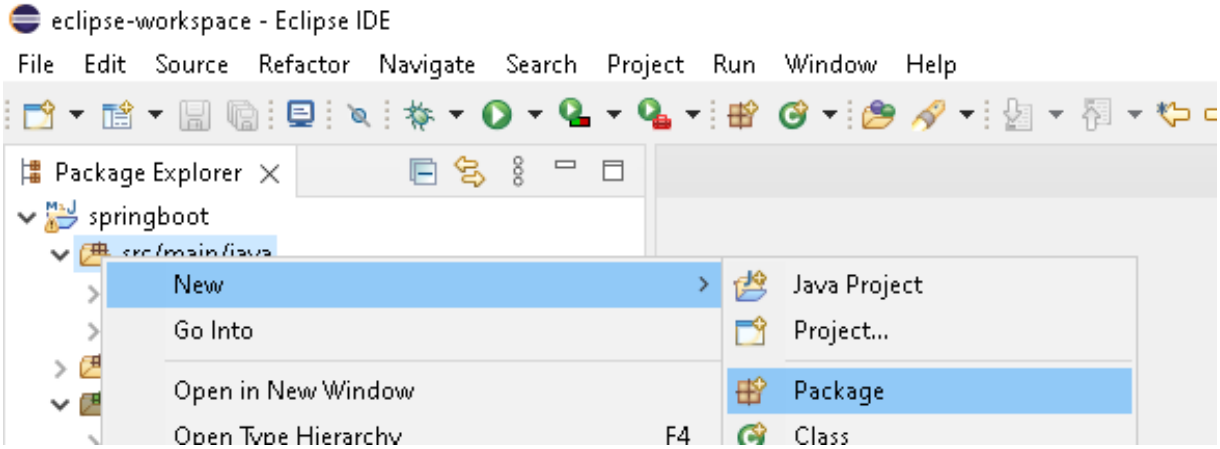
Vamos iniciar o desenvolvimento da nossa aplicação pela tela de cadastro de Cliente. Para isso, devemos inicialmente criar a classe que irá conter os seus dados. Conforme especificado no diagrama de classes da aplicação, devemos implementar a classe Cliente conforme nos mostra a figura abaixo.

Figura 1 – Classe Cliente



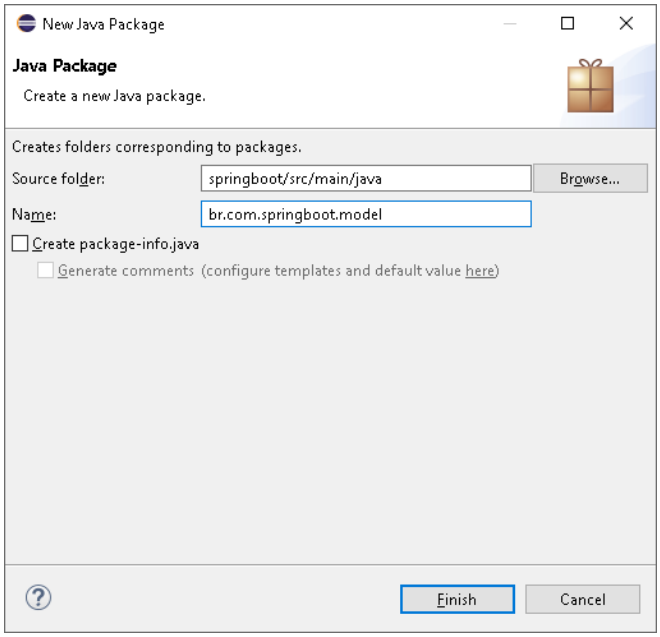
Como iremos criar diversas classes durante o processo de desenvolvimento da aplicação, devemos agrupá-las em pacotes de acordo com as suas responsabilidades. Quanto às classes que irão representar as entidades do banco de dados, como, por exemplo, a classe Cliente, iremos manter dentro do pacote *br.com.springboot.model*. 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



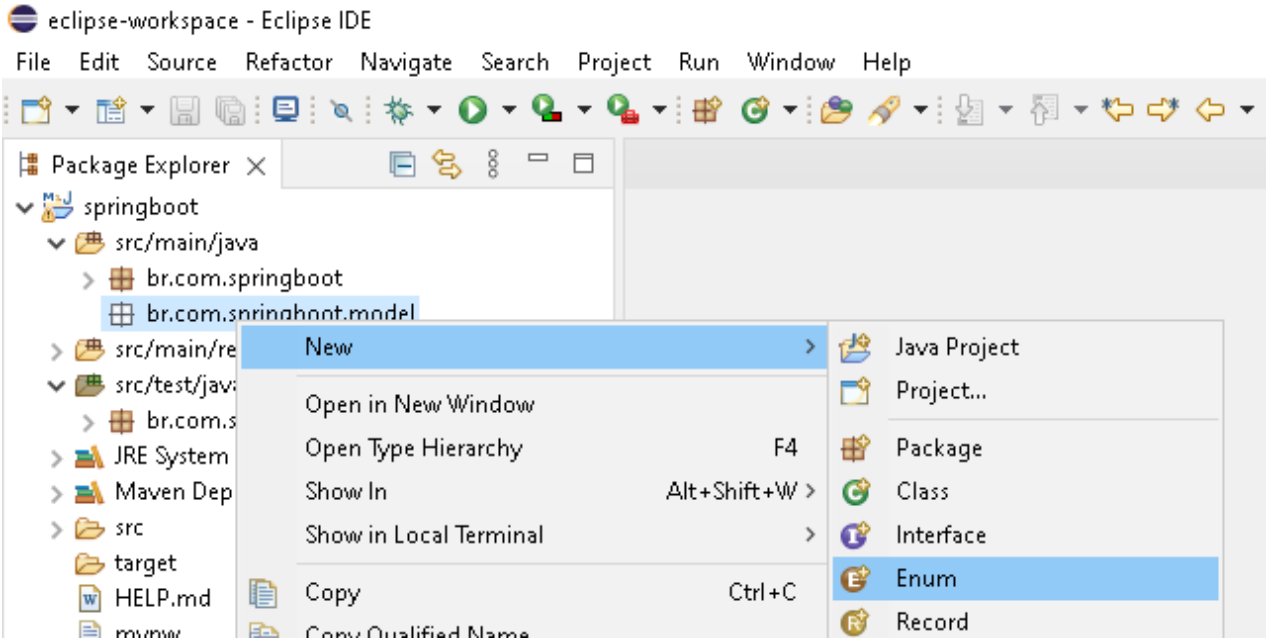
Ao clicar sobre a opção *Package* (Pacote), será aberta a tela para a criação do pacote. Nela, devemos informar o nome do pacote que desejamos criar no campo *Name*, conforme exemplificado na Figura 3.

Figura 3 – Tela de cadastro do pacote



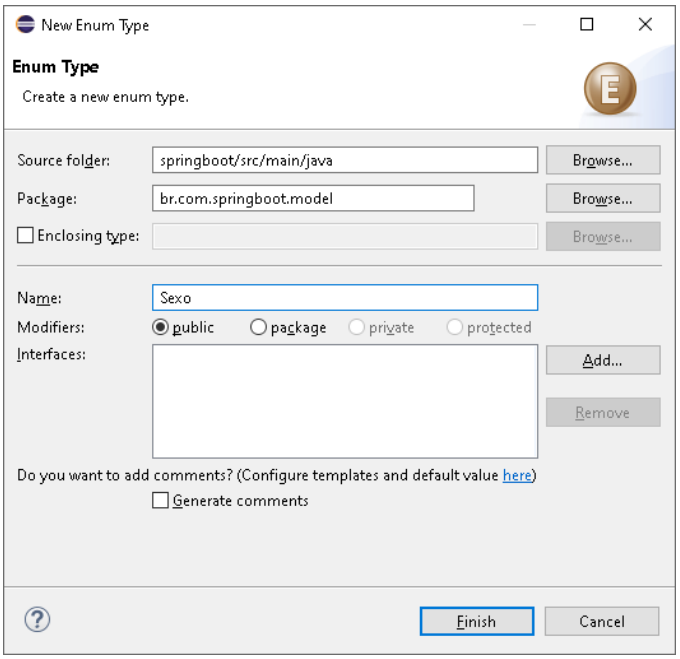
Informado o nome do pacote no campo *Name*, clique no botão *Finish*. Criado o pacote, vamos implementar a enumeração Sexo e a classe Cliente, iniciando a codificação pela primeira. Como a enumeração compõe a classe Cliente, ela também será adicionada ao pacote *br.com.springboot.model*. Para adicioná-la 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. Nelam devemos informar o nome da enumeração que desejamos criar no campo *Name*, conforme exemplificado na Figura 5.

Figura 5 – Tela de cadastro da enumeração



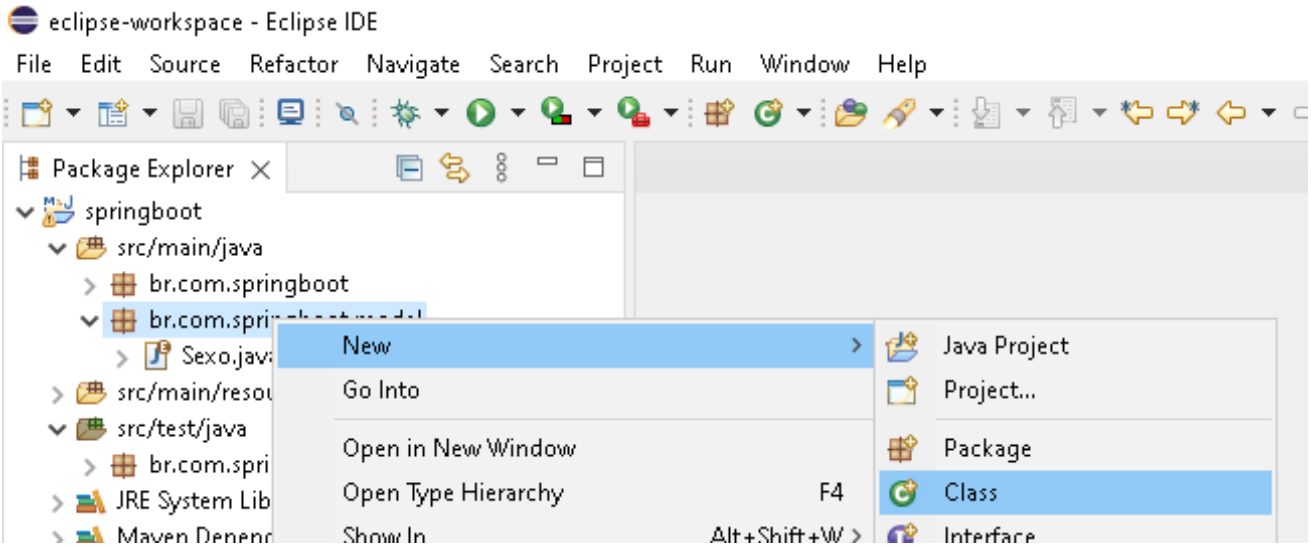
Informado o nome da enumeração no campo *Name* clique no botão *Finish*. A enumeração Sexo irá conter apenas dois valores (Masculino e Feminino) e sua implementação pode ser vista no quadro a seguir.

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;
	}
}
```

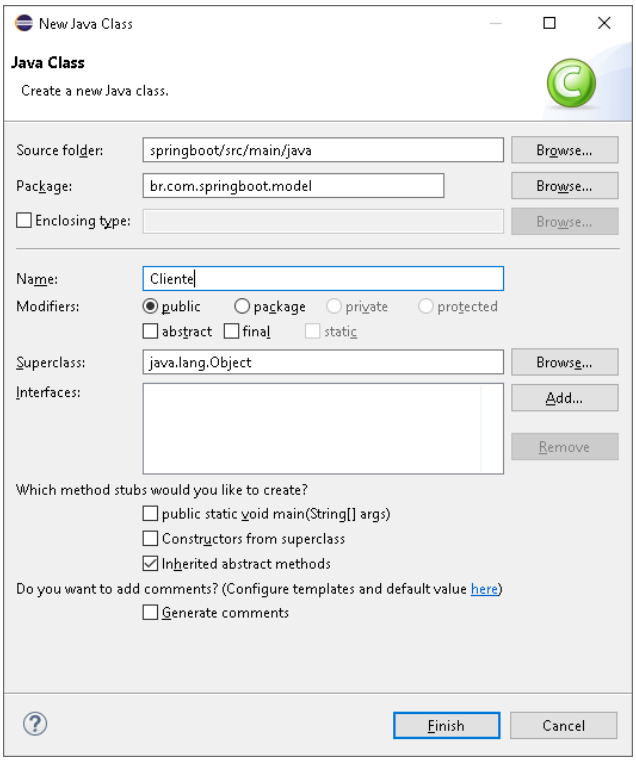
Finalizada a enumeração Sexo, podemos agora iniciar a implementação da classe Cliente. Para 5ria-la, clique com o botão direito sobre o pacote *br.com.springboot.model* e selecione o menu *New* > *Class*, conforme a Figura 6.

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.

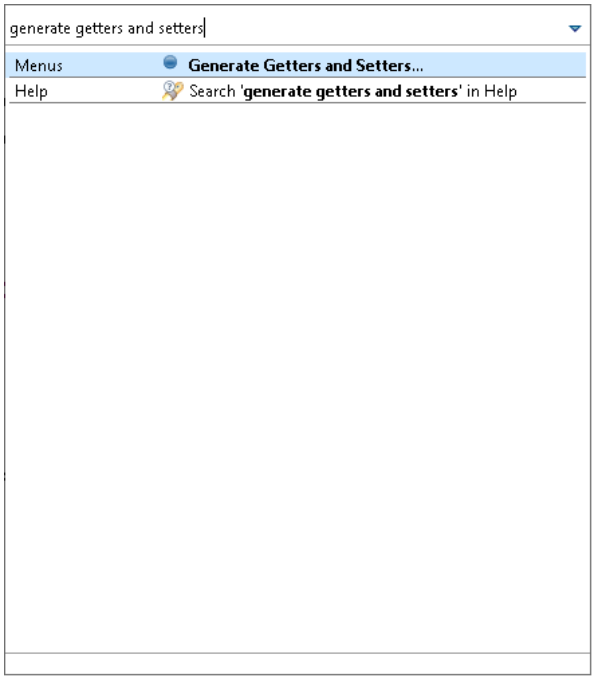
Quadro 2 – Classe Cliente

```
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

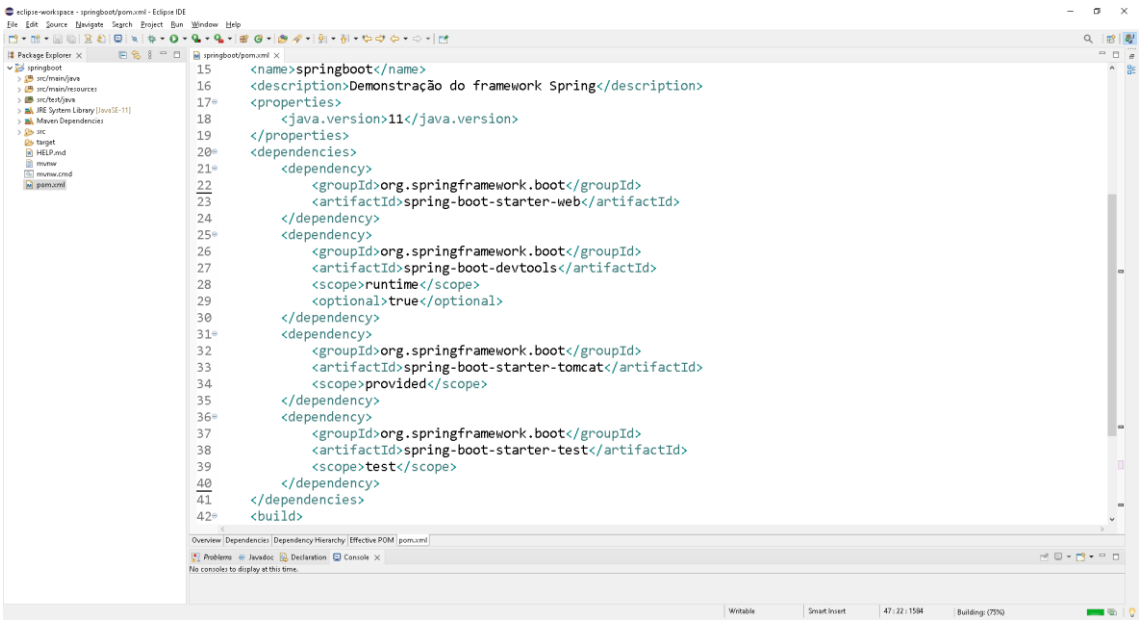
```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
</dependency>
```

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.xmlo pom.xml



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

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 compreende 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.

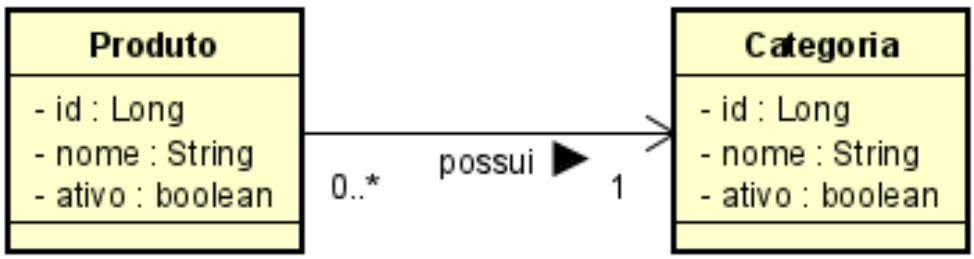
O mapeamento objeto-relacional da classe Categoria, conforme as proposições anteriormente pode ser visualizada no quadro a seguir.

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 @JoinTable, 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;

	// Declarar 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.

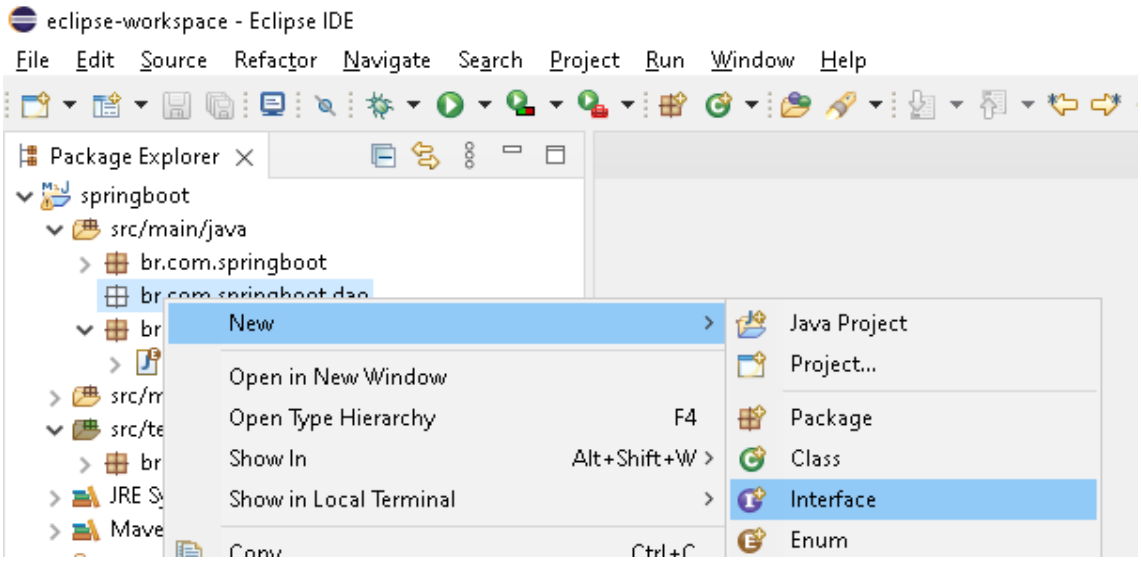
- **TIMESTAMP**: utilizado quando o conteúdo do atributo for do tipo data e hora.
- **@Enumerated**: anotação utilizada para mapear as enumerações. Deve-se especificar como o conteúdo do atributo será persistido por meio da enumeração *EnumType*, que possui dois valores:
- **STRING**: ao adotar esse 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 CRUD.

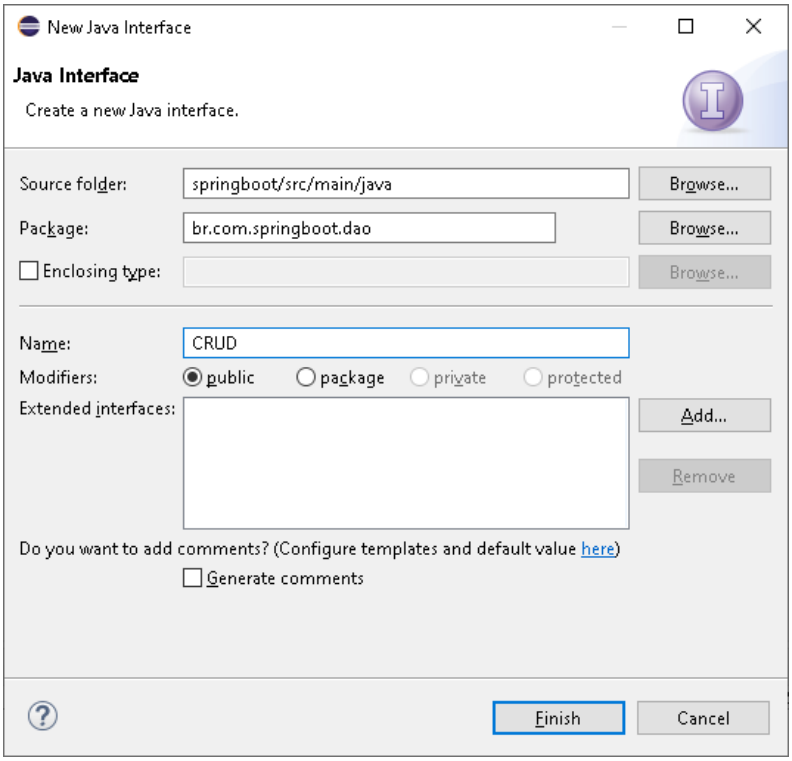
Para criar uma interface, clique com o botão direito sobre o pacote que acabamos de criar e selecione o menu New > Interface, conforme mostra a imagem a seguir.

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.

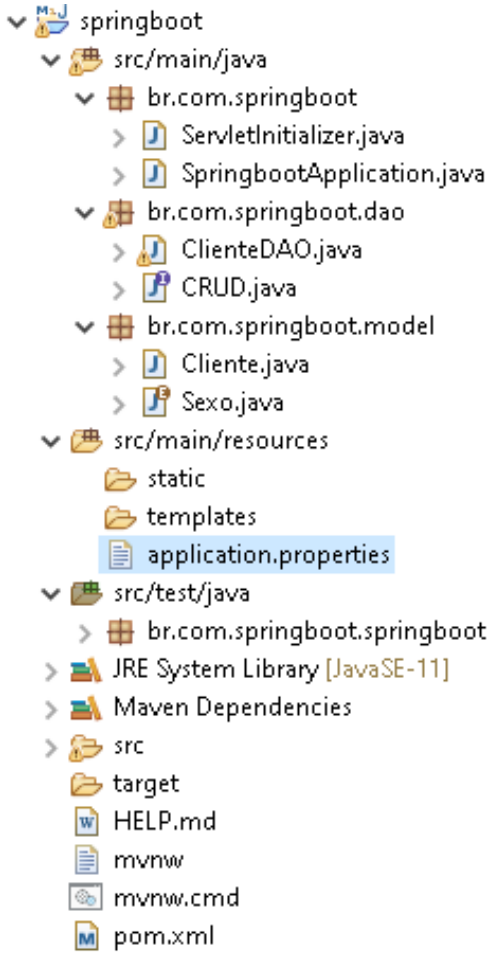
Como o Hibernate irá gerenciar o ciclo de vida das entidades da aplicação, devemos anotar a classe EntityManager com a anotação @PersistenceContext. Finalmente, por se tratar de uma classe de persistência de dados, devemos mapeá-la com a anotação @Repository do Spring, indicando para o *framework* que 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);
   }
}
```

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-plataform com o valor org.hibernate.dialect.MySQL5InnoDBDialect, 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 --- [  resta
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 --- [  resta
2022-01-31 03:21:10.906  INFO 4952 --- [  resta
```

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.