Análise e Desenvolvimento de Sistemas — Profª. Ana Luiza Scharf
Programação Java — ana.scharf@ifsc.edu.br

Laboratório: Exemplo 01 – Java com SQLite usando JDBC

Introdução

Grande parte desse texto abaixo é de autoria do Professor Emerson Ribeiro de Mello. Link do repositório Link aqui

Este repositório apresenta pequenos exemplos com o framework Spring para persistir dados em um banco de dados MySQL.

Nos exemplos é feito uso do Spring Data JPA, que utiliza os padrões de projeto Repository e Data Access Objects (DAO) e é baseado na especificação Java Persistence API (JPA2), usada por frameworks que fazem o mapeamento objeto-relacional (Object-Relational Mapping - ORM).

Como criar o projeto Java

O Spring Boot permite a criação simplificada de aplicações isoladas, ideais durante a etapa de desenvolvimento, bem como para aplicações de produção baseadas no framework Spring.

Com o Spring Boot não é necessário fazer qualquer configuração em arquivos XML, algo típico com JPA, pois algumas configurações ficariam no arquivo *persistence.xml*. No Spring Boot, toda configuração pode ser feita diretamente no código Java e por meio de arquivos de propriedades (properties file) para configurações de conexão com o banco de dados, entre outras.

Para cada um dos exemplos disponíveis neste repositório foi usado o Spring Initializr para criar o esqueleto do projeto. Se deseja criar um projeto como foi criado aqui, siga os passos abaixo:

- Gerar o projeto em https://start.spring.io/
- Configurações:
  - Project: *gradle - groovy*
  - Language: *Java*
  - Spring Boot: *3.5.3*
  - Project metadata:
    - group: *ads.bcd*
    - packaging: *jar*
    - java: *17* (Obs: depende da versão do java do instalada)
  - Dependências:
    - Spring Data JPA
    - MySQL Driver
    - Spring Boot DevTools
- Os demais campos e deixa em branco
- Baixe o arquivo *.ZIP* contendo o projeto Gradle, descompacte-o em uma pasta (de preferência com o nome *Exemplo0X*), e abra essa pasta com o Visual Studio Code ou IntelliJ.

O Spring Boot DevTools inclui um conjunto de ferramentas para tornar mais agradável a experiência de desenvolvimento. De forma resumida, ele irá reiniciar automaticamente a aplicação sempre que notar alguma alteração nos arquivos contidos no classpath. Se não desejar tal comportamento, você pode remover o Spring Boot DevTools da lista de dependências no arquivo *build.gradle*.

Vamos seguir o exemplo disponível no repositório oficial da disciplina, que demonstra a configuração básica do *Spring Boot* com *JPA* e mapeamento de relacionamento “um para um”:

https://github.com/bcd29008/exemplos-com-spring-jpa/blob/main/exemplo-01-um-para-um/Readme.md

Esse modelo nos ajudará a estruturar o projeto corretamente, garantindo a conexão com o banco de dados, o uso de entidades JPA e a exposição dos dados via *Spring Data REST*.

Servidor MySQL

Para executar esse exemplo, é necessário que tenha um servidor MySQL disponível. Você pode subir um rapidamente dentro de um contêiner com o Docker. **Dica: no Windows, o Docker precisa estar rodando — ou seja, é necessário abrir a ferramenta antes de usar.** .Basta executar o comando abaixo:

```
docker run -d --rm -p 3306:3306 -e MYSQL_ROOT_PASSWORD=senhaRoot \
-e MYSQL_DATABASE=bcd -e MYSQL_USER=aluno -e MYSQL_PASSWORD=aluno \
-e MYSQL_ROOT_HOST='%' --name meumysql mysql/mysql-server:latest
```

Cabe lembrar que sempre que o contêiner for parado, ele será excluído (opção *–rm*) e todos os dados serão perdidos. Se quiser que os dados continuem mesmo depois da parada e exclusão do contêiner, passe o parâmetro *-v $(pwd)/db_data:/var/lib/mysql*, que fará o mapeamento do diretório usado pelo MySQL no contêiner para um diretório no computador hospedeiro.

Configuração do Spring para conexão com o banco de dados MySQL

O projeto criado terá o arquivo *src/main/resources/application.properties*, onde são colocadas informações de configuração da aplicação, incluindo as informações de conexão com o banco de dados MySQL.

Edite o arquivo e faça alterações nas seguintes propriedades:

```
spring.datasource.url=jdbc:mysql://localhost:3306/bcd
spring.datasource.username=aluno
spring.datasource.password=aluno
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
```

Próximos passos

Com o projeto criado e com as informações de conexão com MySQL definidas, é hora de criar as classes Java contendo a lógica da aplicação:

- Criar um POJO para cada entidade do banco;
- Criar uma interface para atuar como repositório de cada POJO, esta interface deverá herdar de alguma interface do Spring, por exemplo, *CrudRepository*;
- Criar uma classe com o método *public static void main*, que deverá ser anotada com *@SpringBootApplication*;
- Por fim, executar a aplicação com a tarefa Gradle: *gradle bootRun*.

Configuração do Gradle para saída colorida no console

Para melhorar a visualização dos logs durante a execução da aplicação com Spring Boot, é possível configurar o Gradle para permitir a exibição de cores no console. Para isso, deve-se adicionar o seguinte bloco ao arquivo *build.gradle*:

```
bootRun {
    environment 'spring.output.ansi.console-available', true
}
```

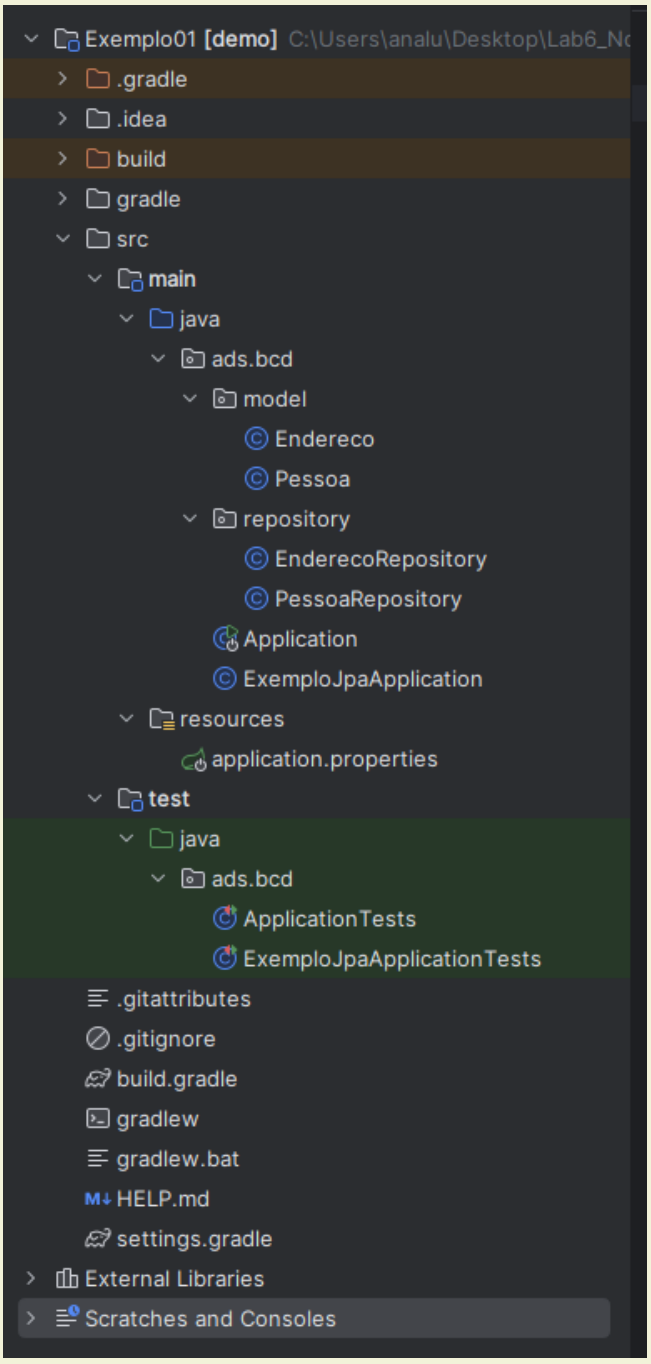
Essa configuração define a variável de ambiente *spring.output.ansi.console-available* como *true* durante a execução da aplicação com o comando *bootRun*.

O objetivo dessa variável é indicar ao Spring Boot que o terminal suporta códigos ANSI para exibição de cores. Com isso, as mensagens de log são coloridas, facilitando a identificação de diferentes níveis de log (como *INFO*, *WARN*, *ERROR*) e melhorando a leitura geral da saída.

Essa personalização é especialmente útil ao executar o projeto via linha de comando, pois o comportamento padrão pode não ativar as cores dependendo do terminal utilizado.

Estrutura de Arquivos do Projeto

A seguir, será incluída uma imagem ilustrando a estrutura de diretórios e arquivos do projeto que vamos desenvolver. Essa estrutura serve como guia para a organização do código-fonte, recursos e configurações, seguindo boas práticas no uso do Spring com JPA.



A imagem apresenta os principais pacotes, como *model*, *repository*, *controller*, além dos arquivos de configuração como o *application.properties* e o *build.gradle*. Com base nessa estrutura, vamos criar e organizar as classes necessárias para o funcionamento completo da aplicação.

Observação: A versão inicial do projeto está disponível no seguinte repositório GitHub:
https://github.com/analuscharf/Lab06_01.git

Código comentado: *Endereco.java*

```
// Pacote onde a classe está localizada
package ads.bcd.model;

// Importações necessárias para trabalhar com JPA e mapeamentos
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;

/**
 * Representa a entidade Endereco do banco de dados.
 * É uma classe POJO (Plain Old Java Object).
 */
@Entity // Informa ao JPA que essa classe será mapeada para uma tabela
public class Endereco {

    // Chave primária gerada automaticamente (AUTO_INCREMENT no MySQL)
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer idEndereco;

    // Campos simples da entidade
    private String rua;
    private String cidade;
    private String estado;
    private String cep;

    /**
     * Relacionamento um-para-um com Pessoa.
     * FetchType.LAZY evita que Pessoa seja carregada imediatamente.
     * optional = false exige que um Endereco sempre tenha uma Pessoa.
     * A chave estrangeira será a coluna id_pessoa.
     */
    @OneToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "id_pessoa")
    private Pessoa pessoa;

    // Construtor padrão obrigatório para JPA
    protected Endereco() {
    }

    // Construtor com todos os atributos para facilitar instâncias
    public Endereco(String rua, String cidade, String estado, String cep, Pessoa p) {
        this.rua = rua;
        this.cidade = cidade;
        this.estado = estado;
        this.cep = cep;
        this.pessoa = p;
    }

    // Getters e Setters gerados automaticamente ou com Lombok

    public Integer getIdEndereco() {
        return idEndereco;
    }

    public void setIdEndereco(Integer idEndereco) {
        this.idEndereco = idEndereco;
    }

    public String getRua() {
        return rua;
```

```
    }

    public void setRua(String rua) {
        this.rua = rua;
    }

    public String getCidade() {
        return cidade;
    }

    public void setCidade(String cidade) {
        this.cidade = cidade;
    }

    public String getEstado() {
        return estado;
    }

    public void setEstado(String estado) {
        this.estado = estado;
    }

    public String getCep() {
        return cep;
    }

    public void setCep(String cep) {
        this.cep = cep;
    }

    public Pessoa getPessoa() {
        return pessoa;
    }

    public void setPessoa(Pessoa pessoa) {
        this.pessoa = pessoa;
    }

    // Representação textual da entidade
    @Override
    public String toString() {
        return "Endereco [cep=" + cep + ", cidade=" + cidade + ", estado=" + estado +
               ", idEndereco=" + idEndereco + ", rua=" + rua + "]";
    }

    // hashCode e equals baseados no idEndereco (boas práticas)
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((idEndereco == null) ? 0 : idEndereco.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Endereco other = (Endereco) obj;
        return idEndereco != null && idEndereco.equals(other.idEndereco);
    }
}
```

Observações importantes

- O campo *idEndereco* será a chave primária da tabela *endereco*.
- O relacionamento *pessoa* é feito via chave estrangeira com a tabela *pessoa*.

- O JPA exige um construtor padrão e métodos getters/setters.
- O *hashCode* e o *equals* são importantes para o bom funcionamento em coleções (como *List*, *Set*, etc.).

Código comentado: *Pessoa.java*

```
// Pacote da classe
package ads.bcd.model;

// Importações para JPA e relacionamentos
import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;

/**
 * POJO para representar a entidade Pessoa.
 *
 * Essa classe será mapeada para a tabela Pessoa no banco de dados.
 */
@Entity // Indica que essa classe é uma entidade JPA
@Table(name = "Pessoa") // Define explicitamente o nome da tabela
public class Pessoa {

    /**
     * Chave primária da entidade, com geração automática (AUTO_INCREMENT).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer idAluno;

    /**
     * Coluna que não aceita valores nulos.
     */
    @Column(nullable = false)
    private String nome;

    // Campo email, pode aceitar nulo por padrão
    private String email;

    /**
     * CPF único e obrigatório. A anotação unique cria restrição UNIQUE no banco.
     */
    @Column(nullable = false, unique = true)
    private String cpf;

    /**
     * Relacionamento bidirecional OneToOne com a entidade Endereco.
     *
     * mappedBy = "pessoa" indica que Pessoa é o lado inverso da relação,
     * e que Endereco é o dono da associação.
     *
     * fetch LAZY: Endereco será carregado somente quando acessado.
     *
     * cascade ALL: operações cascata, ex: excluir Pessoa exclui Endereco.
     */
    @OneToOne(mappedBy = "pessoa", fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    private Endereco endereco;

    // Construtor padrão (obrigatório para JPA)
    protected Pessoa() {
    }

    // Construtor com os principais atributos para facilitar a criação
    public Pessoa(String nome, String email, String cpf) {
```

```
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
    }

    // Getters e setters

    public Integer getIdAluno() {
        return idAluno;
    }

    public void setIdAluno(Integer idAluno) {
        this.idAluno = idAluno;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getCpf() {
        return cpf;
    }

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

    public Endereco getEndereco() {
        return endereco;
    }

    public void setEndereco(Endereco endereco) {
        this.endereco = endereco;
    }

    // Método toString para impressão legível do objeto
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Pessoa [cpf=").append(cpf)
          .append(", email=").append(email)
          .append(", idAluno=").append(idAluno)
          .append(", nome=").append(nome);
        if (this.endereco != null) {
            sb.append(", endereco=").append(endereco);
        }
        sb.append("]");
        return sb.toString();
    }

    // hashCode baseado na chave primária
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((idAluno == null) ? 0 : idAluno.hashCode());
        return result;
    }

    // equals baseado na chave primária para comparar objetos corretamente
    @Override
```

```
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Pessoa other = (Pessoa) obj;
        if (idAluno == null) {
            if (other.idAluno != null)
                return false;
        } else if (!idAluno.equals(other.idAluno))
            return false;
        return true;
    }
}
```

Observações

- O atributo *idAluno* é a chave primária e é gerado automaticamente pelo banco.
- A anotação *@Column* permite controlar restrições como *nullable* e *unique*.
- O relacionamento *OneToOne* com *Endereco* é bidirecional, com *Pessoa* sendo o lado inverso (não dono da relação).
- O uso de *cascade = CascadeType.ALL* faz com que ações em *Pessoa* repercutam no *Endereco* associado (por exemplo, excluir Pessoa exclui Endereco).
- O *fetch = FetchType.LAZY* indica que o endereço só será carregado quando for explicitamente acessado, melhorando performance.
- É importante ter os métodos *hashCode* e *equals* para que a entidade funcione bem em coleções e caches.

Código Comentado: *EnderecoRepository.java*

```
// Pacote onde está essa interface
package ads.bcd.repository;

// Importa a interface CrudRepository do Spring Data
import org.springframework.data.repository.CrudRepository;

// Importa a classe Endereco que será gerenciada
import ads.bcd.model.Endereco;

/**
 * Objetivo da interface repository do Spring Data:
 * Reduzir a quantidade de código repetitivo necessário para a camada
 * de acesso a dados.
 *
 * Documentação oficial:
 * https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories
 *
 * Métodos herdados da interface CrudRepository:
 *
 * <S extends T> S save(S entity);       // salva ou atualiza uma entidade no banco
 * Optional<T> findById(ID primaryKey);  // busca uma entidade pelo ID
 * Iterable<T> findAll();                 // retorna todas as entidades
 * long count();                         // retorna a quantidade total de entidades
 * void delete(T entity);                // remove uma entidade
 * boolean existsById(ID primaryKey);    // verifica se existe uma entidade com dado ID
 */
public interface EnderecoRepository extends CrudRepository<Endereco, Integer> {

}
```

Explicação detalhada

- *import org.springframework.data.repository.CrudRepository;*
  Importa a interface base do Spring Data que oferece operações CRUD básicas.
- *public interface EnderecoRepository extends CrudRepository<Endereco, Integer>*
  Declara uma interface que herda de *CrudRepository*, parametrizada com *Endereco* (entidade) e *Integer* (tipo da chave primária).

  Com isso, o Spring Data automaticamente implementa métodos para:

  - salvar (*save*)
  - buscar por id (*findById*)
  - buscar todos (*findAll*)
  - contar (*count*)
  - deletar (*delete*)
  - verificar existência (*existsById*)
- O uso dessa interface elimina a necessidade de criar manualmente as implementações das operações básicas no banco, facilitando e agilizando o desenvolvimento.

Código Comentado: *PessoaRepository.java*

```
// Define o pacote do projeto
package ads.bcd.repository;

// Importa List para retorno de consultas personalizadas
import java.util.List;

// Importa CrudRepository do Spring Data para operações CRUD básicas
import org.springframework.data.repository.CrudRepository;

// Importa a entidade Pessoa que será gerenciada pelo repositório
import ads.bcd.model.Pessoa;

/**
 * O objetivo da interface repository do Spring Data é reduzir a quantidade
 * de código repetitivo para acesso a dados.
 *
 * Documentação oficial:
 * https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories
 *
 * Métodos herdados da interface CrudRepository:
 *
 * <S extends T> S save(S entity);        // salva ou atualiza uma entidade
 * Optional<T> findById(ID primaryKey);   // busca uma entidade pelo ID
 * Iterable<T> findAll();                  // busca todas as entidades
 * long count();                          // conta o total de entidades
 * void delete(T entity);                 // remove uma entidade
 * boolean existsById(ID primaryKey);     // verifica existência por ID
 */
public interface PessoaRepository extends CrudRepository<Pessoa, Integer> {

    /*
     * Consultas personalizadas podem ser criadas apenas declarando o método,
     * sem implementar. O Spring Data gera a implementação automaticamente,
     * basta seguir regras para nomeação dos métodos.
     *
     * Documentação:
     * https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-
     creation
     * https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.subject
     */
    List<Pessoa> findByCpf(String cpf);
}
```

Explicação detalhada

- *package ads.bcd.repository;*
  Define o pacote da interface, mantendo a organização do projeto.
- *import java.util.List;*
  Importa a classe *List* para retorno de múltiplos objetos em consultas personalizadas.
- *import org.springframework.data.repository.CrudRepository;*
  Importa a interface base que oferece métodos CRUD prontos.
- *import ads.bcd.model.Pessoa;*
  Importa a entidade Pessoa que será gerenciada pelo repositório.
- *public interface PessoaRepository extends CrudRepository<Pessoa, Integer>*
  Define a interface repositório para a entidade *Pessoa*, com chave primária do tipo *Integer*.
- Método personalizado *List<Pessoa> findByCpf(String cpf);*
  Este método cria automaticamente uma consulta que busca pessoas pelo campo CPF, retornando uma lista de objetos que correspondem ao CPF fornecido.

  A implementação é gerada pelo Spring Data, basta que o nome do método siga o padrão *findBy[Campo]* para criar consultas simples.

Código comentado: ExemploJpaApplication

```
// Define o pacote da aplicação
package ads.bcd;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import ads.bcd.model.Endereco;
import ads.bcd.model.Pessoa;
import ads.bcd.repository.EnderecoRepository;
import ads.bcd.repository.PessoaRepository;

/**
 * A anotação @SpringBootApplication indica que esta é a classe principal da aplicação Spring Boot.
 */
@SpringBootApplication
public class ExemploJpaApplication {

    // Logger para registrar mensagens úteis durante a execução da aplicação
    private static final Logger log = LoggerFactory.getLogger(ExemploJpaApplication.class);

    // Injeção automática dos repositórios para acessar o banco de dados
    @Autowired
    PessoaRepository pessoaRepository;

    @Autowired
    EnderecoRepository enderecoRepository;

    // Método main que inicia a aplicação Spring Boot
    public static void main(String[] args) {
        SpringApplication.run(ExemploJpaApplication.class, args);
        log.info("Aplicação finalizada");
    }

    /**
```

```
     * Bean que será executado logo após a aplicação iniciar, graças à interface CommandLineRunner.
     * Serve para rodar código de exemplo, como popular banco e listar dados.
     */
    @Bean
    public CommandLineRunner demoOneToOne() {
        return (args) -> {
            try {
                log.info("Iniciando aplicação");

                // Método para inserir dados de exemplo
                this.povoarBase();

                // Método para listar dados inseridos
                this.listandoRegistros();

            } catch (Exception e) {
                // Registra erros no log caso ocorram
                log.error(e.toString());
            }
        };
    }

    /**
     * Método para popular o banco de dados com registros de Pessoa e Endereco.
     * Exemplo simples para demonstrar relacionamento um-para-um.
     */
    private void povoarBase() throws Exception {
        // Criando objetos Pessoa
        Pessoa juca = new Pessoa("Juca de Oliveira", "juca@email.com", "123.456.789-00");
        Pessoa pedro = new Pessoa("Pedro", "pedro@email.com", "456-789-012-33");

        // Criando objetos Endereco associados às pessoas
        Endereco enderecoJuca = new Endereco("Rua das Oliveiras, 10", "São José", "SC", "88.103-30", juca);
        Endereco enderecoPedro = new Endereco("Rua José Lino Kretzer, 608", "São José", "SC", "88.103-30",
     pedro);

        // Atribuindo os endereços às pessoas
        juca.setEndereco(enderecoJuca);
        pedro.setEndereco(enderecoPedro);

        // Salvando as pessoas no banco, que por cascata salvam os endereços
        pessoaRepository.save(juca);
        pessoaRepository.save(pedro);
    }

    /**
     * Método para listar todos os registros de Pessoa e buscar pessoas por CPF específico.
     */
    private void listandoRegistros() throws Exception {
        System.out.println("----------- Todas Pessoas ---------------------");
        for (var pessoa : pessoaRepository.findAll()) {
            System.out.println(pessoa);
        }
        System.out.println("-----------------------------------------------");

        System.out.println("------- Pessoas com CPF específico ------------");
        List<Pessoa> resultado = pessoaRepository.findByCpf("123.456.789-00");
        resultado.forEach(System.out::println);
        System.out.println("-----------------------------------------------");
    }
}
```

Resumo

Este código mostra como:

- Inicializar uma aplicação Spring Boot com *@SpringBootApplication*.
- Usar injeção de dependência para acessar repositórios do Spring Data.

- Popular o banco de dados com dados de exemplo usando JPA e relacionamentos.
- Executar comandos no startup da aplicação com *CommandLineRunner*.
- Fazer consultas e listar os resultados no console.

Executando o Projeto

Abra sua IDE e execute a classe *ExemploJpaApplication.java*, ou então utilize a linha de comando com Gradle:

```
./gradlew bootRun
```

Referências

- https://www.datafaker.net/documentation/getting-started/
- https://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html
- https://spring.io/guides
- https://spring.io/guides/gs/accessing-data-mysql/
- https://spring.io/guides/gs/accessing-data-jpa/
- https://docs.spring.io/spring-boot/docs/2.6.3/reference/htmlsingle/#data.sql.jpa-and-spring-data
- https://docs.spring.io/spring-boot/docs/2.6.3/reference/htmlsingle/#boot-features-spring-mvc-template-engines
- https://www.oracle.com/technical-resources/articles/javase/persistenceapi.html
- https://www.baeldung.com/jpa-many-to-many
- https://www.baeldung.com/jpa-persisting-enums-in-jpa
- https://attacomsian.com/blog
- http://querydsl.com/
- https://www.oracle.com/corporate/features/project-lombok.html
- https://projectlombok.org/
- Official Gradle documentation
- Spring Boot Gradle Plugin Reference Guide
- Create an OCI image
- Spring Data JPA
- Thymeleaf