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

Laboratório: Exemplo 02 – 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
```

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.

Spring Data JPA

O **Spring Data JPA** permite executar diferentes tipos de consultas com base nos nomes de métodos da classe entidade, utilizando os chamados *Derived Query Methods*. Para isso, o nome do método é dividido em dois componentes separados pelo delimitador *By*.

- **Introdutor:** *find*, *read*, *query*, *count* ou *get*. Indica ao Spring Data JPA o que se deseja fazer com o método, podendo conter outras expressões como *Distinct*.

- **Critério:** aparece após o delimitador *By* e define o critério de seleção das tuplas. Pode ser concatenado com as palavras *And* e *Or*.

Exemplos

```
Optional<Campus> findBySigla(String sigla);
```

```
List<Campus> findByNome(String nome);
```

```
List<Curso> findByNome(String nomeDoCurso);
```

```
List<Curso> findDistinctByNome(String nomeDoCurso);
```

```
int countByCampus(Campus campus);
```

Spring Data REST

Neste exemplo, é utilizado o **Spring Data REST**, que permite criar facilmente recursos REST com base nos repositórios utilizados para interagir com as entidades.

Para isso, é necessário adicionar a seguinte dependência no arquivo *build.gradle*:

```
implementation 'org.springframework.boot:spring-boot-starter-data-rest'
```

O caminho do recurso é derivado do nome da classe e convertido para o plural (seguindo as regras do inglês). É possível alterar esse caminho com a anotação *@RepositoryRestResource*:

```
@RepositoryRestResource(collectionResourceRel = "campus", path = "campus")
public interface CampusRepository extends CrudRepository<Campus, Long> {
}
```

Paginação com Spring Data REST

Para adicionar paginação em uma coleção, a interface do repositório deve estender *PagingAndSortingRepository<T, ID>*. Exemplo:

```
@RepositoryRestResource(collectionResourceRel = "cursos", path = "cursos")
public interface CursoRepository
    extends PagingAndSortingRepository<Curso, Long>,
            CrudRepository<Curso, Long> {
}
```

Com isso, nas consultas personalizadas, será necessário adicionar um parâmetro do tipo *Pageable* e o tipo de retorno deverá ser *Page* ou *Slice*, e não mais *List*.

```
Page<Curso> findByNomeStartingWith(String prefixo, Pageable pageable);
```

Biblioteca Lombok

Neste exemplo foi feito uso da **biblioteca Lombok**, que tem como objetivo facilitar a escrita de código Java, evitando a repetição de métodos como *getters*, *setters*, *toString*, construtores, entre outros.

Para isso, utilizam-se anotações que geram automaticamente esses métodos em tempo de compilação.

Adicionando o Lombok ao projeto

No projeto Gradle, a dependência do Lombok pode ser adicionada de forma mais moderna utilizando o plugin *io.freefair.lombok*. O comando a seguir deve ser incluído no bloco de plugins do arquivo *build.gradle.kts*:

```
plugins {
    id 'io.freefair.lombok' version "8.14"
}
```

Com isso, o Gradle se encarrega de adicionar todas as dependências necessárias do Lombok.

Suporte na IDE

- O **IntelliJ** já possui o plugin do Lombok habilitado por padrão (em versões mais recentes).
- No **Visual Studio Code**, é necessário instalar manualmente a extensão *"Lombok Annotations Support for VS Code"*, disponível no painel de extensões.

Exemplo de uso com Lombok

```
import lombok.Data;

@Data
public class Aluno {
    private Long matricula;
    private String nome;
    private String email;
}
```

No exemplo acima, a anotação *@Data* gera automaticamente os métodos *get*, *set*, *equals*, *hashCode* e *toString*.

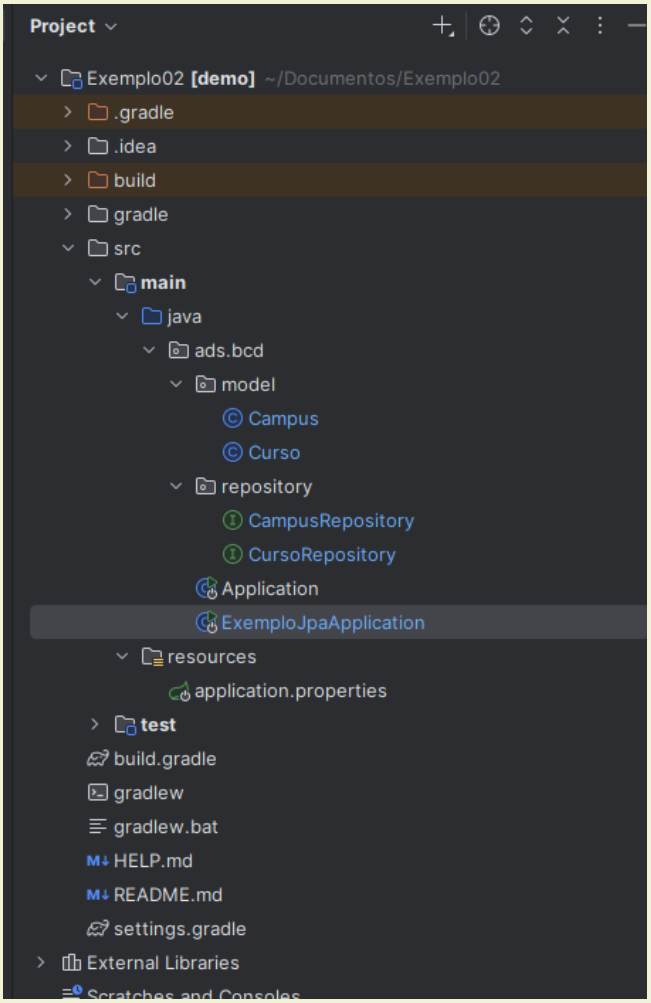
Próximos passos

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*.

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

Classe Campus

```
package ads.bcd.model;

import jakarta.persistence.*;
import lombok.*;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

/**
 * POJO para representar a entidade Campus.
 *
 * As anotações Getter, Setter, EqualsAndHashCode, toString e NoArgsConstructor
 * são da biblioteca lombok que facilita a criação do POJO
 */
@Getter
@Setter
@EqualsAndHashCode
@ToString(exclude = {"cursos"})
@RequiredArgsConstructor
@Entity
public class Campus implements Serializable {
```

```java
    // JPA exige construtor padrão
    protected Campus() {}

    /**
     * A anotação @Id indica que o atributo é a chave primária da entidade.
     * A anotação @GeneratedValue define a estratégia de geração automática.
     * Aqui usamos IDENTITY, o que significa que o valor será gerado no banco (MySQL AUTO_INCREMENT).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer idCampus;

    /**
     * A anotação @Column define restrições para a coluna no banco.
     * O atributo nome é obrigatório (nullable = false).
     * A anotação @NonNull é do Lombok e marca esse campo como obrigatório
     * no construtor gerado pela anotação @RequiredArgsConstructor.
     */
    @Column(nullable = false)
    @NonNull
    private String nome;

    /**
     * A sigla do campus deve ser única e não nula.
     */
    @Column(nullable = false, unique = true)
    @NonNull
    private String sigla;

    @NonNull
    private String endereco;

    @NonNull
    private String cidade;

    /**
     * Relacionamento 1:N com a entidade Curso.
     * Um campus pode ter vários cursos.
     * A anotação mappedBy indica que o lado dono da associação é o atributo "campus" da classe Curso.
     * CascadeType.ALL garante que persistência, atualização e remoção se apliquem também aos cursos.
     */
    @OneToMany(mappedBy = "campus", cascade = {CascadeType.ALL})
    private Set<Curso> cursos = new HashSet<>();
}
```

Resumo das Anotações Utilizadas

- *@Entity* — transforma a classe em uma entidade JPA.
- *@Id* — define o campo como chave primária.
- *@GeneratedValue* — gera automaticamente o valor do ID.
- *@Column* — configura restrições da coluna no banco.
- *@OneToMany* — define relacionamento com múltiplos cursos.
- *@Getter, @Setter, @EqualsAndHashCode, @ToString, @RequiredArgsConstructor* — geram código automaticamente com Lombok.

Observações

- A exclusão do curso ao excluir um campus é possível devido ao *CascadeType.REMOVE* (presente em *CascadeType.ALL*).
- O uso de *@ToString(exclude = {"cursos"})* evita problemas de loop infinito ao imprimir objetos com referências circulares.

- A interface *Serializable* permite que objetos do tipo *Campus* possam ser transmitidos via rede ou armazenados em arquivos.

Classe *Curso*

A classe *Curso* representa uma entidade do banco de dados com um relacionamento do tipo **muitos-para-um** com a entidade *Campus*.

Ela também utiliza JPA para o mapeamento objeto-relacional e Lombok para reduzir o código repetitivo.

Código da Classe

```java
package ads.bcd.model;

import java.io.Serializable;
import jakarta.persistence.*;
import lombok.*;

/**
 * POJO para representar a entidade Curso.
 *
 * É necessário que a classe tenha getter/setter, construtores, toString, hashCode e equals.
 * O Lombok gera tudo isso automaticamente.
 */
@Getter
@Setter
@EqualsAndHashCode
@ToString
@RequiredArgsConstructor
@Entity
@Table(name = "Curso")
public class Curso implements Serializable {

    // Construtor padrão exigido pelo JPA
    protected Curso() {}

    /**
     * A anotação @Id define a chave primária da entidade.
     * @GeneratedValue com estratégia IDENTITY permite que o valor seja gerado automaticamente
     * no banco de dados (AUTO_INCREMENT no MySQL).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer idCurso;

    /**
     * Nome do curso (obrigatório).
     */
    @NonNull
    private String nome;

    /**
     * Carga horária do curso (obrigatória).
     */
    @NonNull
    private int cargaHoraria;

    /**
     * Relacionamento muitos-para-um com a entidade Campus.
     *
     * Muitos cursos podem estar associados a um único campus.
     * A anotação @JoinColumn define o nome da chave estrangeira na tabela Curso.
     */
    @ManyToOne
    @JoinColumn(name = "idCampus", nullable = false)
    @NonNull
    private Campus campus;
}
```

Resumo das Anotações Utilizadas

- *@Entity* — define a classe como uma entidade JPA.
- *@Table(name = "Curso")* — opcional, define o nome da tabela explicitamente.
- *@Id* — define o campo como chave primária.
- *@GeneratedValue* — define a geração automática da chave primária.
- *@ManyToOne* — indica um relacionamento N:1 com Campus.
- *@JoinColumn* — configura a chave estrangeira.
- *@Getter, @Setter, @EqualsAndHashCode, @ToString, @RequiredArgsConstructor* — anotações Lombok que geram o código repetitivo automaticamente.

Observações

- O nome da chave estrangeira será *idCampus*, como especificado em *@JoinColumn*.
- O campo *campus* deve estar preenchido sempre, pois está marcado como *nullable = false*.
- Como o relacionamento é bidirecional, o lado dono da associação é a entidade *Curso*.

Interface *CampusRepository*

A interface *CampusRepository* é responsável pelo acesso aos dados da entidade *Campus*, utilizando o framework **Spring Data JPA**.

Ela estende *CrudRepository*, o que permite herdar métodos prontos para realizar operações básicas (CRUD) no banco de dados.

Além disso, essa interface faz uso da anotação *@RepositoryRestResource*, o que expõe automaticamente os métodos da interface como endpoints RESTful.

Código da Interface

```java
package ads.bcd.repository;

import java.util.List;
import java.util.Optional;

import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import ads.bcd.model.Campus;

@RepositoryRestResource(collectionResourceRel = "campus", path = "campus")
public interface CampusRepository extends CrudRepository<Campus, Long> {

    // Consulta personalizada: encontrar um campus pela sigla
    Optional<Campus> findBySigla(String sigla);

    // Consulta personalizada: listar todos os campus com determinado nome
    List<Campus> findByNome(String nome);
}
```

Anotações e Conceitos Importantes

- *@RepositoryRestResource*: torna o repositório acessível via requisições HTTP. O nome do recurso será *campus* e o caminho também será */campus*.
- *CrudRepository<Campus, Long>*: herda os seguintes métodos prontos:
  - *save(S entity)*: salva ou atualiza uma entidade.

  - *findById(ID id)*: retorna uma entidade pelo ID.
  - *findAll()*: retorna todas as entidades.
  - *count()*: retorna o número total de entidades.
  - *delete(T entity)*: exclui uma entidade.
  - *existsById(ID id)*: verifica se uma entidade existe.
- As consultas *findBySigla* e *findByNome* são exemplos de **consultas derivadas** (*Derived Query Methods*) do Spring Data JPA, que funcionam com base no nome do método.

Referências Úteis

- Spring Data JPA Documentation
- API - CrudRepository
- Query Creation - Spring Data JPA

Interface *CursoRepository*

A interface *CursoRepository* é responsável pelas operações de acesso a dados relacionadas à entidade *Curso*. Ela estende tanto *CrudRepository* quanto *PagingAndSortingRepository*, o que permite realizar operações CRUD, paginação e ordenação de forma automática, sem a necessidade de implementar métodos.

Código da Interface

```
package ads.bcd.repository;

import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import ads.bcd.model.Campus;
import ads.bcd.model.Curso;

@RepositoryRestResource(collectionResourceRel = "cursos", path = "cursos")
public interface CursoRepository extends PagingAndSortingRepository<Curso, Long>, CrudRepository<Curso, Long>
      {

    Page<Curso> findByNome(String nomeDoCurso, Pageable pageable);

    Page<Curso> findDistinctByNomeIgnoreCase(String nomeDoCurso, Pageable pageable);

    Page<Curso> findByNomeContainingOrderByNome(String nomeDoCurso, Pageable pageable);

    int countByCampus(Campus campus);

    List<Curso> findByCampusAndCargaHoraria(Campus campus, int cargaHoraria);

    List<Curso> findByCargaHorariaIsNull();

    Page<Curso> findByNomeStartingWith(String prefixo, Pageable pageable);

    Page<Curso> findByNomeEndingWith(String sufixo, Pageable pageable);

    Page<Curso> findByNomeContaining(String padrao, Pageable pageable);

    Page<Curso> findByCargaHorariaGreaterThan(int valor, Pageable pageable);

    Page<Curso> findByCargaHorariaBetween(int inicio, int fim, Pageable pageable);

    void deleteByNome(String nome);
}
```

Anotações e Extensões

- *@RepositoryRestResource*: disponibiliza a interface como recurso REST automático. O caminho base será */cursos*.
- *PagingAndSortingRepository*: permite uso de paginação e ordenação.
- *CrudRepository*: fornece operações CRUD básicas.

Consultas Derivadas (Derived Query Methods)

- *findByNome(String, Pageable)* — lista cursos com nome exato, paginado.
- *findDistinctByNomeIgnoreCase(...)* — sem repetições e sem diferenciar maiúsculas/minúsculas.
- *findByNomeContainingOrderByNome(...)* — busca por parte do nome e ordena em ordem crescente.
- *countByCampus(...)* — conta quantos cursos existem para um campus.
- *findByCampusAndCargaHoraria(...)* — combinação de dois critérios.
- *findByCargaHorariaIsNull()* — lista cursos sem carga horária.
- *findByNomeStartingWith(...)* — nomes que começam com uma determinada string.
- *findByNomeEndingWith(...)* — nomes que terminam com uma string.
- *findByNomeContaining(...)* — nomes que contêm uma string.
- *findByCargaHorariaGreaterThan(...)* — carga horária maior que o valor informado.
- *findByCargaHorariaBetween(...)* — carga horária entre dois valores.
- *deleteByNome(...)* — exclui curso com nome informado.

Sobre Paginação e Ordenação

- O tipo de retorno *Page<Curso>* exige que o método receba um parâmetro do tipo *Pageable*, que define:
  - **page** — número da página (começa em 0)
  - **size** — quantidade de registros por página
  - **sort** — campo(s) para ordenação
- Exemplo de chamada: */cursos/search/findByNome?nomeDoCurso=Engenharia&page=0&size=5*

Links Úteis

- Documentação de consultas derivadas
- Documentação sobre paginação

Classe *ExemploJpaApplication*

Esta é a classe principal da aplicação Spring Boot, responsável por inicializar o contexto da aplicação, povoar o banco de dados com alguns registros e realizar consultas de teste usando o Spring Data JPA.

Código da Classe

```java
package ads.bcd;

import java.util.Optional;
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 org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

import ads.bcd.model.Campus;
import ads.bcd.model.Curso;
import ads.bcd.repository.CampusRepository;
import ads.bcd.repository.CursoRepository;
import lombok.extern.slf4j.Slf4j;

@Slf4j
@SpringBootApplication
public class ExemploJpaApplication {

    @Autowired
    CampusRepository campusRepository;

    @Autowired
    CursoRepository cursoRepository;

    public static void main(String[] args) {
        SpringApplication.run(ExemploJpaApplication.class, args);
        log.info("Aplicação finalizada");
    }

    private void povoando() throws Exception {
        Campus campusSje = new Campus("São José", "SJE", "Rua José Lino, 608", "São José");
        Campus campusFln = new Campus("Florianópolis", "FLN", "Avenida Mauro Ramos, 100", "Florianópolis");

        campusRepository.save(campusSje);
        campusRepository.save(campusFln);

        cursoRepository.save(new Curso("Engenharia de Telecomunicações", 4300, campusSje));
        cursoRepository.save(new Curso("Engenharia de Computação", 4200, campusSje));
        cursoRepository.save(new Curso("Engenharia Elétrica", 4500, campusFln));
    }

    private void listandoRegistros() throws Exception {
        System.out.println("--------------- Campus -------------------------");
        for (var element : campusRepository.findAll()) {
            System.out.println(element);
        }

        System.out.println("--------------- Cursos -------------------------");
        cursoRepository.findAll().forEach(System.out::println);

        System.out.println("---------- Cursos paginados e ordenados ---------");
        Pageable pagina = PageRequest.of(0, 20, Sort.by("nome").ascending());
        cursoRepository.findByCargaHorariaGreaterThan(4000, pagina).forEach(System.out::println);

        System.out.println("--------------- Total de cursos ---------------");
        Optional<Campus> buscaCampus = campusRepository.findBySigla("SJE");
        if (buscaCampus.isPresent()) {
            int totalCursosSje = cursoRepository.countByCampus(buscaCampus.get());
            System.out.println("Total de cursos no campus São José: " + totalCursosSje);
        }
    }

    @Bean
    public CommandLineRunner demoUmParaMuitos() {
        return (args) -> {
            try {
```

```
                log.info("Iniciando aplicação");
                this.povoando();
                this.listandoRegistros();
            } catch (Exception e) {
                log.error(e.toString());
            }
        };
    }
}
```

Anotações Utilizadas

- *@SpringBootApplication*: indica que esta é a classe principal da aplicação.
- *@Slf4j*: fornece acesso ao serviço de log (*log.info*, *log.error*, etc.).
- *@Autowired*: realiza injeção de dependência dos repositórios.
- *@Bean*: define que o método será executado automaticamente ao iniciar a aplicação, retornando um *CommandLineRunner*.

Resumo das Funções

- *main()*: inicializa a aplicação Spring Boot.
- *povoando()*: cria e salva registros de exemplo no banco de dados.
- *listandoRegistros()*: realiza listagens e consultas utilizando métodos personalizados dos repositórios.
- *demoUmParaMuitos()*: executa automaticamente o povoamento e a listagem ao iniciar a aplicação.

Destaques Importantes

- O uso de *PageRequest* com *Sort* permite realizar consultas com paginação e ordenação.
- A anotação *@Bean* junto com *CommandLineRunner* é útil para testar a aplicação diretamente no console, sem precisar de uma interface gráfica.
- As mensagens do *log* ajudam a acompanhar o fluxo da aplicação.

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