CENTRO UNIVERSITÁRIO ALFREDO NASSER
ENGENHARIA DE SOFTWARE

Aluno: João Gustavo Caetano Sales
Professor: Brenno Pimenta
Matéria: Programação Orientada a Objetos I

Projeto de Desenvolvimento de Software (PDS)

1. Introdução

Este projeto tem como objetivo desenvolver uma aplicação em **Java** que tem a classe professor e a classe aluno e os 2 tem a classe controller, service e entity.

A aplicação será desenvolvida para o Trabalho do Professor Brenno.

2. Processo:

Em sala de aula fomos em um site chamado Spring Initializr e criamos uma pasta com as seguintes configurações:
Project: Maven
Language: Java
Spring Boot: 3.2.9
Packaging: Jar
Java: 17 (por erros com o computador utilizado teve que ser modificado para o 11)
Dependencies: Spring Boot DevTools, Lombok, Spring Configuration Processor, Spring Web.

Com a pasta nos começou a fazer os algoritmos implementados que irão aparecer em Especificações Funcionais.

Depois nós extraímos a pasta e carregamos ela no IntelliJ depois disso nos deu play no UniversidadeEsn3Application.java e depois disso fomos fazendo as classes etc.

Depois de fazer as classes abaixamos as imagens fornecida pelo professor pelo cmd e usando

o Docker Desktop usamos para entrar em um site para criar os container etc. Pos isso nos configuro para funcionar no Postman e la que cria os alunos etc.

3. Especificações Funcionais

Algoritmos Implementados:

- ### Aluno:
  Este código em Java define uma classe chamada `Aluno`, que parece ser uma entidade do sistema representando um estudante. Ele utiliza a biblioteca Lombok e as anotações do JPA (Java Persistence API) para mapear esta classe como uma entidade do banco de dados.

  ### Estrutura e Função do Código

1. **Pacote e importações:**
   - O pacote define a estrutura onde esta classe está localizada: `com.example.universidadeESN3.entity`.
   - As importações incluem o Lombok (`@Data`) para gerar automaticamente getters, setters, `equals`, `hashCode` e `toString`, e as anotações do JPA para o mapeamento da entidade.
2. **Anotações:**
   - `@Data`: Lombok gera automaticamente métodos como getters, setters e outros.
   - `@Entity`: Indica que esta classe é uma entidade gerenciada pelo JPA.
   - `@Id`: Marca o atributo `id` como a chave primária da tabela.
   - `@GeneratedValue`: Especifica que o valor do `id` será gerado automaticamente pelo banco de dados (estratégia `IDENTITY`).
   - `@Enumerated(EnumType.STRING)`: Mapeia o campo `genero`, que é um enum, para ser salvo como `String` no banco de dados.
3. **Atributos:**
   - `id`: Identificador único gerado automaticamente.
   - `matricula`: Número de matrícula do aluno.
   - `nome`: Nome do aluno.
   - `genero`: Gênero do aluno (provavelmente uma enumeração chamada `Genero`).
   - `active`: Um booleano indicando se o aluno está ativo ou não.
4. **Método `toString`:**
   - Reescreve o método `toString` para retornar uma representação em texto da classe. O Lombok gera este método automaticamente, mas aqui foi personalizado.

Função Geral:

Esta classe é usada para representar alunos em uma aplicação que usa um banco de dados relacional. Com o JPA, é possível mapear esta entidade para uma tabela no banco, onde os campos da classe correspondem às colunas da tabela. Além disso, o uso do Lombok reduz a necessidade de escrever código boilerplate, como getters e setters.

Código:

```java
package com.example.universidadeESN3.entity;

import lombok.Data;

import javax.persistence.*;

@Data
@Entity
public class Aluno {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long matricula;
    private String nome;
    @Enumerated(EnumType.STRING)
    private Genero genero;

    private Boolean active;

    @Override
    public String toString() {
        return "Aluno{" +
                "id=" + id +
                ", matricula=" + matricula +
                ", nome='" + nome + '\'' +
                ", genero=" + genero +
                '}';
    }
}
```

.

- **Professor:**
  Este código em Java define a classe `Professor`, que representa uma entidade no contexto de um sistema, provavelmente relacionado a uma universidade. Assim como o exemplo anterior, a classe utiliza o JPA (Java Persistence API) e Lombok para mapear e gerenciar a entidade no banco de dados. Vamos analisar os detalhes:

Estrutura e Função do Código

1. **Pacote e importações:**
    - O pacote `com.example.universidadeESN3.entity` organiza esta classe dentro do projeto.
    - As importações incluem anotações do JPA para configurar a entidade e a biblioteca Lombok para reduzir código repetitivo.
2. **Anotações:**
    - `@Data`: Gerado pelo Lombok, cria automaticamente métodos como getters, setters, toString, equals e hashCode para os atributos da classe.
    - `@Entity`: Indica que esta classe é uma entidade JPA, ou seja, será mapeada para uma tabela no banco de dados.
    - `@Id`: Define o atributo `id` como a chave primária da tabela.
    - `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Configura o JPA para gerar automaticamente os valores do campo `id` usando a estratégia de identidade (geração automática pelo banco de dados, geralmente auto-incremento).
3. **Atributos:**
    - `id`: Identificador único gerado automaticamente.
    - `matricula`: Um número de matrícula exclusivo para identificar o professor.
    - `nome`: O nome do professor.

Função Geral:

Esta classe serve como uma representação de um professor no sistema e é mapeada para uma tabela no banco de dados, onde os atributos da classe correspondem às colunas da tabela. Com essa estrutura, é possível realizar operações de persistência (salvar, atualizar, deletar, consultar) sobre objetos `Professor` diretamente no banco de dados.

Assim como na classe `Aluno`, o uso de Lombok simplifica o código, eliminando a necessidade de escrever métodos padrão manualmente, enquanto as anotações do JPA garantem o mapeamento e gerenciamento da entidade no banco.

Código:

```java
package com.example.universidadeESN3.entity;

import lombok.Data;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class Professor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long matricula;
    private String nome;
}
```

- **Gênero:**
  Este código define uma enumeração chamada `Genero`, usada para representar os diferentes gêneros possíveis no sistema. As enumerações (ou *enums*) são tipos especiais no Java que limitam um campo a um conjunto pré-definido de valores constantes.

Detalhes do Código:

1. **Pacote:**
   - O código está no pacote `com.example.universidadeESN3.entity`, organizando esta enum como parte do modelo de entidades do sistema.
2. **Declaração da Enum:**
   - `public enum Genero`: Declara a enumeração `Genero`, que é pública e pode ser usada em todo o projeto.
3. **Constantes:**
   - A enum contém três valores possíveis:
     - `FEMININO`: Representa o gênero feminino.
     - `MASCULINO`: Representa o gênero masculino.
     - `OUTROS`: Representa outros gêneros ou casos não especificados.

Função Geral:

A função dessa enum é restringir o campo `genero` (presente na classe `Aluno`) para aceitar apenas os valores definidos (`FEMININO`, `MASCULINO` ou `OUTROS`). Isso promove validação automática e facilita o controle do sistema, reduzindo o risco de erros ou valores inválidos.

No banco de dados, graças à anotação `@Enumerated(EnumType.STRING)` na classe `Aluno`, os valores dessa enum serão armazenados como Strings (`FEMININO`, `MASCULINO` ou `OUTROS`) em vez de números inteiros.

- **Código:**

```
1    package com.example.universidadeESN3.entity;
2
3    public enum Genero {
4        FEMININO,
5        MASCULINO,
6        OUTROS
7    }
```

- **AlunoService:**
  Este código define a classe `AlunoService`, que implementa a lógica de negócios para gerenciar entidades `Aluno` no sistema. Ela utiliza o Spring Framework e um repositório JPA (`AlunoRepository`) para interagir com o banco de dados. Vamos detalhar as funções principais dessa classe:

Função Geral

A classe `AlunoService` é um serviço no contexto de um projeto Spring. Sua principal função é fornecer operações específicas para gerenciar objetos da entidade `Aluno`, atuando como uma camada intermediária entre o controller (responsável por lidar com requisições HTTP) e o repositório (que acessa diretamente o banco de dados).

Detalhes do Código

1. **Anotações:**
   - **`@Service`:** Marca a classe como um componente de serviço do Spring, permitindo que seja gerenciada pelo *container* do Spring.
   - **`@Slf4j`:** Permite o uso de um logger para gerar logs em métodos específicos.

    - **`@Autowired`:** Injeta automaticamente a dependência `AlunoRepository` para que ela possa ser usada sem necessidade de instância manual.
2. **Interface `IAlunoService`:**
    - A classe implementa `IAlunoService`, o que define um contrato para os métodos que precisam ser implementados na camada de serviço (interface não exibida aqui, mas inferida).
3. **Métodos da Classe:**

- **`buscarPorId(Long id)`:** Busca um aluno no banco de dados pelo `id`. Retorna um objeto `Aluno` caso o registro exista, ou `null` caso contrário.
- **`buscarTodos()`:** Retorna todos os registros de alunos no banco.
- **`salvar(Aluno aluno)`:** Salva um novo aluno no banco ou atualiza um aluno existente. Também registra o processo no log.
- **`atualizar(Aluno aluno)`:** Atualiza as informações de um aluno existente no banco. É similar ao método `salvar`, mas a intenção semântica é diferente.
- **`excluir(Long id)`:** Exclui um aluno pelo `id` no banco de dados.
- **`desativar(Aluno aluno)`:** Define o campo `active` de um aluno como `false` e salva a atualização no banco.
- **`buscarPorNome(String nome)`:** Retorna uma lista de alunos cujos nomes começam com o valor fornecido (`nome`), ignorando diferenças entre maiúsculas e minúsculas.

Conclusão

Este serviço é responsável pelas principais operações relacionadas à entidade `Aluno`. Ele encapsula a lógica de negócios, mantém os métodos organizados e permite reutilizar facilmente as operações em diferentes partes do sistema.

Código:

```java
package com.example.universidadeESN3.service;

import com.example.universidadeESN3.entity.Aluno;
import com.example.universidadeESN3.repository.AlunoRepository;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;

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

@Service
@Slf4j
public class AlunoService implements IAlunoService {

    @Autowired
    private AlunoRepository alunoRepository;

    @Override
    public Aluno buscarPorId(Long id) {
        Optional<Aluno> response = alunoRepository.findById(id);
        if (response.isPresent()) {
            return response.get();
        }
        return null;
    }

    @Override
    public List<Aluno> buscarTodos() {
        return alunoRepository.findAll();
    }

    @Override
    public Aluno salvar(Aluno aluno) {
        log.info("salvar() - aluno:{}", aluno );
        return alunoRepository.save(aluno);
    }

    @Override
    public void atualizar(Aluno aluno) {
        log.info("atualizar() - aluno:{}", aluno );
        alunoRepository.save(aluno);
    }

    @Override
    public void excluir(Long id) {
        alunoRepository.deleteById(id);
    }

    public void desativar(Aluno aluno) {
        aluno.setActive(Boolean.FALSE);
        alunoRepository.save(aluno);
    }

    public List<Aluno> buscarPorNome(String nome) {
//        return alunoRepository.findByNome(nome);
        return alunoRepository.findByNomeStartingWithIgnoreCase(nome);
    }
}
```

- **IAlunoService:**
  Este código define a interface `IAlunoService`, que serve como um contrato para a camada de serviço relacionada à entidade `Aluno`. Em Java, uma interface declara

os métodos que devem ser implementados por qualquer classe que a implemente, mas não contém a implementação dos métodos em si.

Função Geral

A função principal desta interface é garantir que qualquer classe que implemente `IAlunoService` (como a classe `AlunoService`, mencionada anteriormente) siga um conjunto específico de regras, ou seja, implemente os métodos listados. Isso promove consistência e facilita a substituição ou alteração da implementação do serviço no futuro, caso necessário.

Métodos Declarados

1. **`Aluno buscarPorId(Long id);`**
    - **Descrição:** Busca um aluno pelo seu `id` único.
    - **Retorno**: Um objeto `Aluno` correspondente ao `id`, ou `null`/`Optional` caso o aluno não seja encontrado.
2. **`List<Aluno> buscarTodos();`**
    - **Descrição:** Retorna uma lista de todos os alunos cadastrados no sistema.
    - **Retorno:** Uma lista de objetos `Aluno`.
3. **`Aluno salvar(Aluno aluno);`**
    - **Descrição:** Salva um novo aluno no sistema ou atualiza um existente.
    - **Parâmetro:** Um objeto `Aluno` com os dados a serem salvos.
    - **Retorno:** O objeto `Aluno` que foi salvo, possivelmente com informações adicionais (como o `id` gerado).
4. **`void atualizar(Aluno aluno);`**
    - **Descrição:** Atualiza os dados de um aluno existente.
    - **Parâmetro:** Um objeto `Aluno` com as informações atualizadas.
    - **Retorno:** Não retorna nada (`void`).
5. **`void excluir(Long id);`**
    - **Descrição:** Exclui um aluno do sistema com base no seu `id`.
    - **Parâmetro:** O `id` do aluno a ser excluído.
    - **Retorno:** Não retorna nada (`void`).

Importância e Benefícios da Interface

1. **Consistência:**

    - Garante que todas as classes que implementem `IAlunoService` tenham os mesmos métodos disponíveis, mesmo que a lógica interna seja diferente.
2. **Flexibilidade:**
    - Torna o código mais modular. Por exemplo, é possível criar outra implementação de `IAlunoService` sem impactar outras partes do sistema.
3. **Facilidade de Testes:**
    - Interfaces facilitam a criação de *mocks* e *stubs* em testes unitários, promovendo um design orientado a testes.
4. **Dependência por Interface:**
    - No contexto do Spring, é comum injetar dependências com base na interface, o que desacopla o código da implementação específica.

Conclusão

A interface `IAlunoService` define as operações essenciais para gerenciar a entidade `Aluno` no sistema. Ela permite a criação de implementações específicas que obedecem a este contrato, promovendo consistência e flexibilidade na aplicação.

Código:

```java
package com.example.universidadeESN3.service;

import com.example.universidadeESN3.entity.Aluno;
import java.util.List;

public interface IAlunoService {

    Aluno buscarPorId(Long id);

    List<Aluno> buscarTodos();

    Aluno salvar(Aluno aluno);

    void atualizar(Aluno aluno);

    void excluir(Long id);
}
```

- **ProfessorService:**
  O código apresentado define a classe `ProfessorService`, que implementa a interface `IProfessorService`. No entanto, os métodos estão apenas declarados,

mas não possuem uma implementação funcional (apenas retornam `null` ou estão vazios). Isso significa que a classe está incompleta e não pode ser usada de maneira útil no sistema sem que os métodos sejam implementados corretamente.

Função Geral

A classe `ProfessorService` deveria ser a implementação da interface `IProfessorService`, fornecendo a lógica de negócio para gerenciar a entidade `Professor`. Como um componente do Spring (graças à anotação `@Service`), ela seria responsável por manipular os dados de `Professor` em um banco de dados por meio do `ProfessorRepository`.

No entanto, como os métodos não estão implementados, atualmente a classe não executa nenhuma operação real.

Detalhamento das Anotações

1. **`@Service`:**
   - Indica que esta classe é um componente de serviço no Spring e que será gerenciada pelo container do Spring.
   - Classes marcadas com `@Service` geralmente contêm a lógica de negócios.
2. **`@Slf4j`:**
   - Habilita o uso do logger da biblioteca Lombok para gerar logs durante a execução do serviço. Embora esteja configurado, os métodos não utilizam logs no estado atual.
3. **`@Autowired`:**
   - Injeta automaticamente uma instância do repositório `ProfessorRepository`, que deveria ser usado para interagir com o banco de dados.

Métodos Declarados (Incompletos)

1. **`buscarPorId(Long id)`:**
   - **Descrição:** O método deveria buscar um professor com base no seu `id` no banco de dados, mas atualmente retorna `null`.
   -
2. **`buscarTodos()`:**

- **Descrição:** Deveria retornar uma lista com todos os professores cadastrados no banco, mas atualmente retorna `null`.

3. **`salvar(Professor professor)`:**
   - **Descrição:** Este método deveria salvar um novo professor no banco de dados ou atualizar um existente, mas atualmente retorna `null`.
4. **`atualizar(Professor professor)`:**
   - **Descrição:** Deveria atualizar os dados de um professor já existente no banco. Está vazio no momento.
5. **`excluir(Long id)`:**
   - **Descrição:** Deveria excluir um professor com base no seu `id`. Está vazio no momento.

Conclusão

Atualmente, a classe `ProfessorService` está incompleta e não funcional. Sua principal função deveria ser implementar a lógica de negócios para gerenciar a entidade `Professor` por meio do `ProfessorRepository`. Assim que os métodos forem implementados corretamente, ela poderá ser utilizada para manipular os dados de professores no sistema.

Código:

```java
package com.example.universidadeESN3.service;

import com.example.universidadeESN3.entity.Professor;
import com.example.universidadeESN3.repository.ProfessorRepository;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@Slf4j
public class ProfessorService implements IProfessorService {

    @Autowired
    private ProfessorRepository professorRepository;

    @Override
    public Professor buscarPorId(Long id) {
        return null;
    }

    @Override
    public List<Professor> buscarTodos() {
        return null;
    }

    @Override
    public Professor salvar(Professor professor) {
        return null;
    }

    @Override
    public void atualizar(Professor professor) {

    }

    @Override
    public void excluir(Long id) {

    }
}
```

- **IProfessorService:**
  Este código define a interface `IProfessorService`, que funciona como um contrato para a camada de serviço da entidade `Professor`. Assim como a interface `IAlunoService` (vista anteriormente), ela estabelece um conjunto de métodos que devem ser implementados por qualquer classe que deseje oferecer operações para gerenciar `Professor`.

Função Geral

A interface `IProfessorService` especifica as operações básicas para manipular registros da entidade `Professor`. Isso inclui buscar, salvar, atualizar e excluir dados de professores no sistema.

Métodos Declarados

1. **`Professor buscarPorId(Long id);`**
   - **Descrição:** Busca um professor específico no sistema com base no `id`.
   - **Retorno:** Um objeto `Professor` correspondente ao `id` informado, ou `null`/`Optional` caso o registro não exista.
2. **`List<Professor> buscarTodos();`**
   - **Descrição:** Retorna uma lista com todos os professores cadastrados no sistema.
   - **Retorno:** Uma lista de objetos `Professor`.
3. **`Professor salvar(Professor professor);`**
   - **Descrição:** Salva um novo professor no sistema ou atualiza um professor existente.
   - **Parâmetro:** Um objeto `Professor` com as informações que devem ser salvas.
   - **Retorno:** O objeto `Professor` que foi salvo, possivelmente com campos adicionais preenchidos (como o `id` gerado pelo banco de dados).
4. **`void atualizar(Professor professor);`**
   - **Descrição:** Atualiza os dados de um professor existente no sistema.
   - **Parâmetro:** Um objeto `Professor` contendo os dados atualizados.
   - **Retorno:** Não retorna nada (`void`).
5. **`void excluir(Long id);`**
   - **Descrição:** Exclui o registro de um professor do sistema com base no seu `id`.
   - **Parâmetro:** O `id` do professor que deve ser removido.
   - **Retorno:** Não retorna nada (`void`).

Função da Interface no Sistema

- Garante que todas as classes que implementem `IProfessorService` sigam as mesmas operações padrão para a entidade `Professor`.
- Permite flexibilidade ao sistema, permitindo diferentes implementações da lógica de negócios sem afetar outras partes do código.

- Facilita o uso de boas práticas, como injeção de dependências no Spring, desacoplando a implementação da lógica de negócios.

Conclusão

A interface `IProfessorService` funciona como um contrato para a gestão da entidade `Professor` no sistema, especificando os métodos essenciais para as operações CRUD (Create, Read, Update, Delete). Isso garante consistência entre implementações e facilita o desenvolvimento e manutenção do sistema.

Código:

```java
package com.example.universidadeESN3.service;

import com.example.universidadeESN3.entity.Aluno;
import com.example.universidadeESN3.entity.Professor;

import java.util.List;

public interface IProfessorService {

    Professor buscarPorId(Long id);

    List<Professor> buscarTodos();

    Professor salvar(Professor professor);

    void atualizar(Professor professor);

    void excluir(Long id);
}
```

- **AlunoController:**
  O código apresentado define o `AlunoController`, uma classe de controle em uma aplicação Spring Boot que gerencia as requisições relacionadas à entidade `Aluno`. Essa classe funciona como a interface entre o cliente (front-end ou API client) e o serviço do back-end que realiza operações na base de dados.

Função Geral:

O `AlunoController` atua como um ponto de entrada para todas as requisições HTTP relacionadas a alunos. Ele expõe endpoints para buscar, salvar, atualizar, excluir e desativar alunos, delegando as operações para a camada de serviço (`AlunoService`).

Anotações no Código

1. **`@RestController`:**
    - Indica que esta classe é um controller RESTful e que seus métodos retornam dados diretamente no corpo da resposta HTTP, geralmente no formato JSON.
2. **`@RequestMapping("/aluno")`:**
    - Define que todas as rotas dentro desta classe começam com o prefixo `/aluno`.
3. **`@Slf4j`:**
    - Habilita o uso do logger da biblioteca Lombok para gerar logs no sistema.
4. **`@Autowired`:**
    - Realiza a injeção de dependência do serviço `AlunoService`, permitindo que ele seja usado nos métodos do controlador.

Métodos Disponíveis e Suas Funções

1. **`@GetMapping` - Buscar todos os alunos**
    - **Endpoint:** `GET /aluno`
    - **Descrição:** Retorna uma lista de todos os alunos cadastrados.
- **Lógica:**
    - Chama o método `buscarTodos()` da camada de serviço e retorna o resultado em um objeto `ResponseEntity`.

---

2. **`@GetMapping("/{id}")` - Buscar aluno por ID**
    - **Endpoint:** `GET /aluno/{id}`
    - **Descrição:** Busca um aluno específico pelo seu `id`.
    - **Lógica:**
        - Chama o método `buscarPorId(id)` no serviço.
        - Se o aluno for encontrado, retorna com o código HTTP 200 (`ResponseEntity.ok()`).
        - Caso contrário, retorna o código HTTP 404 (`ResponseEntity.notFound().build()`).

---

3. `@GetMapping("/nome/{nome}")` - **Buscar alunos pelo nome**
    - **Endpoint:** `GET /aluno/nome/{nome}`
    - **Descrição:** Busca alunos que possuem nomes que começam com o valor fornecido, ignorando maiúsculas e minúsculas.
    - **Lógica:**
        - Chama o método `buscarPorNome(nome)` no serviço.
        - Se forem encontrados alunos, retorna a lista; caso contrário, retorna uma lista vazia.

---

4. `@PostMapping` - **Salvar um aluno**
    - **Endpoint:** `POST /aluno`
    - **Descrição:** Salva um novo aluno no banco de dados.
    - **Lógica:**
        - Recebe o objeto `Aluno` no corpo da requisição (`@RequestBody`).
        - Chama o método `salvar(aluno)` no serviço e retorna o objeto salvo.

---

5. `@PutMapping` - **Atualizar um aluno**
    - **Endpoint:** `PUT /aluno`
    - **Descrição:** Atualiza os dados de um aluno existente.
    - **Lógica:**
        - Verifica se o aluno existe chamando `buscarPorId(aluno.getId())`.
        - Se não encontrado, retorna `404 Not Found`.
        - Caso encontrado, chama o método `atualizar(aluno)` no serviço e retorna `200 OK`.

---

6. `@DeleteMapping("/{id}")` - **Excluir um aluno**
    - **Endpoint:** `DELETE /aluno/{id}`
    - **Descrição:** Exclui um aluno do banco de dados pelo `id`.
    - **Lógica:**
        - Verifica se o aluno existe chamando `buscarPorId(id)`.
        - Se não encontrado, retorna `404 Not Found`.
        - Caso encontrado, chama o método `excluir(id)` no serviço e retorna `200 OK`.

---

7. **`@DeleteMapping("/inactive/{id}")` - Desativar um aluno**
    - **Endpoint:** `DELETE /aluno/inactive/{id}`
    - **Descrição:** Marca um aluno como inativo no banco de dados.
    - **Lógica:**
        - Verifica se o aluno existe chamando `buscarPorId(id)`.
        - Se não encontrado, retorna `404 Not Found`.
        - Caso encontrado, chama o método `desativar(aluno)` no serviço e retorna `200 OK`.

Fluxo de Funcionamento

1. Cliente faz uma requisição HTTP (GET, POST, PUT, DELETE).
2. O controller interpreta a requisição e chama o método correspondente no serviço `AlunoService`.
3. O serviço realiza as operações necessárias (por exemplo, consultar o banco de dados via repositório) e retorna o resultado.
4. O controller devolve a resposta ao cliente no formato esperado (por exemplo, JSON) e com o código HTTP apropriado.

Resumo

O `AlunoController` é responsável por:

- Receber as requisições HTTP relacionadas a alunos.
- Delegar a lógica de negócios para o serviço `AlunoService`.
- Retornar respostas adequadas para o cliente (dados ou mensagens de erro).

Ele segue o padrão arquitetural MVC (Model-View-Controller), sendo a camada responsável pela comunicação com o cliente.

Código:

```java
package com.example.universidadeESN3.controller;

import com.example.universidadeESN3.entity.Aluno;
import com.example.universidadeESN3.service.AlunoService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

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

@RestController
@RequestMapping("/aluno")
@Slf4j
public class AlunoController {

    @Autowired
    private AlunoService alunoService;

    @GetMapping
    public ResponseEntity<List<Aluno>> buscarTodos() {
        return ResponseEntity.ok(alunoService.buscarTodos());
    }

    @GetMapping(path = "/{id}")
    public ResponseEntity<Aluno> buscarPorId(@PathVariable Long id){

        Aluno response = alunoService.buscarPorId(id);
        if (response != null) {
            return ResponseEntity.ok(response);
        }
        return ResponseEntity.notFound().build();
    }

    @GetMapping(path = "/nome/{nome}")
    public ResponseEntity<List<Aluno>> buscarPorNome(@PathVariable String nome){

        List<Aluno> response = alunoService.buscarPorNome(nome);
        if (response != null && !response.isEmpty()) {
            return ResponseEntity.ok(response);
        }
        return ResponseEntity.ok(new ArrayList<>());
    }

    @PostMapping
    public ResponseEntity<Aluno> salvarAluno(@RequestBody Aluno aluno){
        log.info("salvarAluno() - aluno:{}", aluno );
        return ResponseEntity.ok(alunoService.salvar(aluno));
    }

    @PutMapping()
    public ResponseEntity<?> update(@RequestBody Aluno aluno){

        Aluno response = alunoService.buscarPorId(aluno.getId());
        if (response == null) {
            return ResponseEntity.notFound().build();
        }
        alunoService.atualizar(aluno);
        return ResponseEntity.ok(null);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> delete(@PathVariable Long id) {
        Aluno response = alunoService.buscarPorId(id);
        if (response == null) {
            return ResponseEntity.notFound().build();
        }
        alunoService.excluir(id);
        return ResponseEntity.ok(null);
    }

    @DeleteMapping("/inactive/{id}")
    public ResponseEntity<?> desativar(@PathVariable Long id) {
        Aluno response = alunoService.buscarPorId(id);
        if (response == null) {
            return ResponseEntity.notFound().build();
        }
        alunoService.desativar(response);
        return ResponseEntity.ok(null);
    }


}
```

- **ProfessorController:**

O código define o `ProfessorController`, uma classe de controle responsável por

gerenciar as requisições HTTP relacionadas à entidade `Professor` em uma aplicação Spring Boot. Essa classe faz a interface entre os clientes da aplicação (como front-ends ou outros serviços) e a camada de serviço (`ProfessorService`) que executa a lógica de negócios.

Função Geral

O `ProfessorController` é um ponto de entrada para as operações CRUD relacionadas à entidade `Professor`. Ele expõe uma API RESTful para buscar, salvar, atualizar e excluir registros de professores.

Anotações no Código

1. **`@RestController`:**
   - Indica que esta classe é um controller RESTful, retornando respostas HTTP no formato JSON.
2. **`@RequestMapping("/professor")`:**
   - Define o prefixo `/professor` como o caminho base para todos os endpoints dessa classe.
3. **`@Slf4j`:**
   - Habilita o uso do logger da biblioteca Lombok para registrar logs de eventos.
4. **`@Autowired`:**
   - Realiza a injeção de dependência para o serviço `ProfessorService`, permitindo que ele seja utilizado pelos métodos do controlador.

Métodos Disponíveis e Suas Funções

1. **`@GetMapping` - Buscar todos os professores**
   - **Endpoint:** `GET /professor`
   - **Descrição:** Retorna uma lista de todos os professores cadastrados.
   - Chama o método `buscarTodos()` no serviço e retorna o resultado com código HTTP 200 (`OK`).

---

2. **`@GetMapping("/{id}")` - Buscar professor por ID**
   - **Endpoint:** `GET /professor/{id}`
   - **Descrição:** Retorna um professor específico com base no seu `id`.

- **Lógica:**
  - Chama o método `buscarPorId(id)` no serviço.
  - Se o professor for encontrado, retorna o objeto e o código HTTP 200 (`ResponseEntity.ok()`).
  - Caso contrário, retorna o código HTTP 404 (`ResponseEntity.notFound().build()`).

---

3. `@PostMapping` - Salvar um novo professor

- **Endpoint:** `POST /professor`
- **Descrição:** Salva um novo registro de professor no banco de dados.
- **Lógica:**
  - Recebe o objeto `Professor` no corpo da requisição (`@RequestBody`).
  - Chama o método `salvar(professor)` no serviço e retorna o objeto salvo com código HTTP 200.

---

4. `@PutMapping` - Atualizar um professor existente

- **Endpoint:** `PUT /professor`
- **Descrição:** Atualiza os dados de um professor no banco de dados.
- **Lógica:**
  - Verifica se o professor existe utilizando `buscarPorId(professor.getId())`.
  - Se não encontrado, retorna código HTTP 404 (`ResponseEntity.notFound().build()`).
  - Caso encontrado, chama o método `atualizar(professor)` no serviço e retorna o código HTTP 200.

---

5. `@DeleteMapping("/{id}")` - Excluir um professor

- **Endpoint:** `DELETE /professor/{id}`
- **Descrição:** Exclui um professor do banco de dados pelo seu `id`.
- **Lógica:**
  - Verifica se o professor existe utilizando `buscarPorId(id)`.
  - Se não encontrado, retorna código HTTP 404.
  - Caso encontrado, chama o método `excluir(id)` no serviço e retorna código HTTP 200.

Fluxo de Funcionamento

1. **Requisição HTTP:** O cliente faz uma requisição (GET, POST, PUT ou DELETE) para os endpoints do controlador.
2. **Processamento no Controller:** O método apropriado do controlador é chamado com base na rota e no método HTTP.
3. **Chamadas à Camada de Serviço:** O controlador delega a lógica de negócios ao `ProfessorService`.
4. **Resposta:** O controlador retorna uma resposta HTTP para o cliente, incluindo os dados processados ou mensagens de erro com o código de status apropriado.

Resumo

O `ProfessorController`:

- Recebe requisições relacionadas a professores.
- Chama o serviço `ProfessorService` para executar as operações necessárias.
- Retorna respostas no formato JSON com o status apropriado (200, 404, etc.).

Ele segue o padrão arquitetural MVC (Model-View-Controller) e atua como o ponto de comunicação entre os clientes e a lógica de negócios da aplicação.

Código:

```java
package com.example.universidadeCSN3.controller;

import com.example.universidadeCSN3.entity.Professor;
import com.example.universidadeCSN3.service.ProfessorService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

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

@RestController
@RequestMapping("/professor")
@Slf4j
public class ProfessorController {

    @Autowired
    private ProfessorService professorService;

    @GetMapping
    public ResponseEntity<List<Professor>> buscarTodos() {
        return ResponseEntity.ok(professorService.buscarTodos());
    }

    @GetMapping(path = "/{id}")
    public ResponseEntity<Professor> buscarPorId(@PathVariable Long id){

        Professor response = professorService.buscarPorId(id);
        if (response != null) {
            return ResponseEntity.ok(response);
        }
        return ResponseEntity.notFound().build();
    }


    @PostMapping
    public ResponseEntity<Professor> salvarProfessor(@RequestBody Professor professor){
        log.info("salvarProfessor() - professor:{}", professor );
        return ResponseEntity.ok(professorService.salvar(professor));
    }

    @PutMapping()
    public ResponseEntity<?> update(@RequestBody Professor professor){

        Professor response = professorService.buscarPorId(professor.getId());
        if (response == null) {
            return ResponseEntity.notFound().build();
        }
        professorService.atualizar(professor);
        return ResponseEntity.ok(null);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> delete(@PathVariable Long id) {
        Professor response = professorService.buscarPorId(id);
        if (response == null) {
            return ResponseEntity.notFound().build();
        }
        professorService.excluir(id);
        return ResponseEntity.ok(null);
    }


}
```

4. Código Fonte Backend

```java
package com.example.universidadeESN3;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class UniversidadeEsn3Application {

    public static void main(String[] args) {
        SpringApplication.run(UniversidadeEsn3Application.class, args);
    }

}
```

5. Conclusão do projeto:

Esse PDS é semelhante ao primeiro PDS porém com mais detalhes que eu fiz sem saber o que era o PDS que eu vi com os meninos não ajudou tanto a saber o que fazer nele.

6. Qr Code do Github: