



## Resumo para o Currículo dos Alunos

Durante o curso, os alunos aprenderão a desenvolver uma aplicação web completa utilizando:

- **Java com Spring Boot** para o backend
- **PostgreSQL** como banco de dados
- **HTML, CSS e JavaScript** para o frontend
- **GitHub** para versionamento de código
- **Render** para deploy da aplicação
- **Docker** para empacotar e distribuir o projeto



### **com.alunoskesley.professorkesley**

Este é o **pacote principal** do projeto. Dentro dele, temos subpastas que seguem a arquitetura limpa (Clean Architecture):

#### ◆ **business**

- **PessoaService**: Contém a lógica de negócio. Aqui ficam os métodos que tratam os dados antes de salvar ou retornar para o usuário.

#### ◆ **controller**

- **PessoaController**: Controla as rotas da aplicação. Recebe as requisições HTTP (como GET, POST) e chama os serviços adequados.

#### ◆ **exceptions**

- **GlobalExceptionHandler**: Trata erros de forma centralizada.
- **ResourceNotFoundException**: Erro personalizado para quando um recurso (como uma pessoa) não é encontrado.

#### ◆ **infrastructure/entitys**

- **Pessoa**: Classe que representa a entidade "Pessoa". Contém os atributos como nome, disciplina e ano. (Obs: "entitys" está com erro de digitação, o correto seria "entities")

#### ◆ **repository**

- **PessoaRepository**: Interface que comunica com o banco de dados. Usa Spring Data JPA para facilitar operações como salvar, buscar e deletar.

#### ◆ **ProfessorkesleyApplication**

- Classe principal que inicia a aplicação Spring Boot.

## resources/front\_end

Aqui está o **frontend da aplicação**, feito com HTML, CSS e JavaScript:

- **Logo.webp**: Imagem usada na interface.
- **pessoa.html**: Página principal onde o usuário interage.
- **pessoa.css**: Estilos visuais da página.
- **pessoa.js**: Scripts que fazem a comunicação com o backend (por exemplo, usando `fetch()` para enviar dados).

## application.properties

Arquivo de configuração da aplicação. Aqui você define:

- Conexão com o banco de dados
- Porta do servidor
- Configurações de segurança e logs

## Explicando a Classe Pessoa


Esta classe representa a **entidade Pessoa**, que será armazenada no banco de dados PostgreSQL. Ela está localizada em:

```
src/main/java/com/alunoskesley/professorkesley/infrastructure/entitys/Pessoa.java
```

## Anotações do Lombok

Essas anotações reduzem a quantidade de código repetitivo:

- **@Getter** e **@Setter**: Geram automaticamente os métodos `get` e `set` para todos os atributos.
- **@AllArgsConstructor**: Cria um construtor com todos os atributos.
- **@NoArgsConstructor**: Cria um construtor vazio.
- **@Builder**: Permite criar objetos usando o padrão de projeto Builder.

 Essas anotações vêm da biblioteca **Lombok**, que precisa estar adicionada no `pom.xml`.

## Anotações do JPA (Java Persistence API)

- `@Entity`: Indica que esta classe será mapeada para uma tabela no banco de dados.
- `@Table(name = "pessoa")`: Define o nome da tabela como "pessoa".

### Atributos e Mapeamentos

```
1 @Id
2 @GeneratedValue(strategy = GenerationType.AUTO)
3 private Long id;
4
```

- Define o campo `id` como chave primária.
- O valor será gerado automaticamente pelo banco.






```
1 @Column(name = "nome")
2 private String nome;
3
```

- Mapeia o campo `nome` para a coluna "nome" da tabela.

```
1 @Column(name = "email", unique = true)
2 private String email;
3
```






- Mapeia o campo email e garante que ele seja único no banco.

```
1 @Column(name = "curso")
2 private String curso;
3
```

     Java

- Armazena o nome do curso informado pelo aluno.

```
1 @Column(name = "ano")
2 private Integer ano;
3
```

     Java

- Armazena o ano em que o aluno teve aula.

## Relacionamento com o Projeto

Essa classe é usada em conjunto com:

- **PessoaRepository**: Interface que acessa o banco de dados.
- **PessoaService**: Lógica de negócio.
- **PessoaController**: Recebe as requisições HTTP e chama o serviço.

## Explicando a Classe PessoaService

### Localização:

```
src/main/java/com/alunoskesley/professorkesley/business/PessoaService.java
```

### Função da Classe

A PessoaService é responsável por **intermediar** a comunicação entre o PessoaController (que recebe as requisições HTTP) e o PessoaRepository (que acessa o banco de dados).

## Explicação dos Componentes

### ◆ @Service

Indica que esta classe é um **componente de serviço** do Spring. Ela será gerenciada automaticamente pelo framework.

### ◆ @Autowired





O Spring injeta automaticamente uma instância de `PessoaRepository` no construtor da classe. Isso permite que o serviço acesse os métodos de persistência de dados.

## Métodos da Classe

### ✓ salvar(Pessoa pessoa)

Salva uma nova pessoa no banco de dados.





```
1 return pessoaRepository.save(pessoa);
2
```

    </> Java

### listarTodas()

Retorna uma lista com todas as pessoas cadastradas.





```
1 return pessoaRepository.findAll();
2
```

    </> Java

### buscarPorId(Long id)

Busca uma pessoa pelo ID. Retorna um `Optional<Pessoa>` que pode estar vazio se não encontrar.

```
1 return pessoaRepository.findById(id);
2
```

    </> Java

### atualizar(Long id, Pessoa novaPessoa)

Atualiza os dados de uma pessoa existente. Se não encontrar, lança uma exceção personalizada.

```
return pessoaRepository.findById(id)
    .map(pessoa -> {
        pessoa.setNome(novaPessoa.getNome());
        pessoa.setEmail(novaPessoa.getEmail());
        pessoa.setCurso(novaPessoa.getCurso());
        pessoa.setAno(novaPessoa.getAno());
        return pessoaRepository.save(pessoa);
    })
    .orElseThrow(() -> new ResourceNotFoundException("Pessoa com ID " + id + " não encontrada"))
```

Remove uma pessoa do banco de dados pelo ID.

```
1 pessoaRepository.deleteById(id);
2
```

## Explicando a Classe PessoaController

### Localização:

```
src/main/java/com/alunoskesley/professorkesley/controller/PessoaController.java
```

### Função da Classe

A PessoaController recebe as requisições HTTP (como GET, POST, PUT, DELETE) e chama os métodos da PessoaService para executar as ações desejadas.

### Anotações Importantes

- `@RestController`: Indica que esta classe é um controlador REST. Os métodos retornam dados diretamente no corpo da resposta.
- `@RequestMapping("/api/pessoas")`: Define o caminho base para todos os endpoints da classe.
- `@CrossOrigin(origins = "*")`: Permite que o frontend acesse a API mesmo se estiver hospedado em outro domínio (útil para integração com HTML/JS).

## Métodos da Classe

### @PostMapping → Criar nova pessoa

```
1 public ResponseEntity<Pessoa> criarPessoa(@RequestBody Pessoa pessoa)
2
```

- Recebe um objeto Pessoa no corpo da requisição.
- Chama o serviço para salvar.
- Retorna o objeto criado com status 201 CREATED.

### @GetMapping → Listar todas as pessoas

```
1 public ResponseEntity<List<Pessoa>> listarTodas()
2
```

- Retorna uma lista com todas as pessoas cadastradas.
- Status 200 OK.

### @GetMapping("/{id}") → Buscar pessoa por ID

```
1 public ResponseEntity<Pessoa> buscarPorId(@PathVariable Long id)
2
```

- Busca uma pessoa pelo ID.
- Se encontrar, retorna com 200 OK; se não, retorna 404 Not Found.

## @PutMapping("/{id}") → Atualizar pessoa

```
1 public ResponseEntity<Pessoa> atualizarPessoa(@PathVariable Long id, @RequestBody Pessoa p
2
```

- Atualiza os dados de uma pessoa existente.
- Se o ID não existir, retorna 404 Not Found.

## @DeleteMapping("/{id}") → Deletar pessoa

```
1 public ResponseEntity<Void> deletarPessoa(@PathVariable Long id)
2
```

- Remove a pessoa do banco de dados.
- Retorna 204 No Content (sem corpo na resposta).

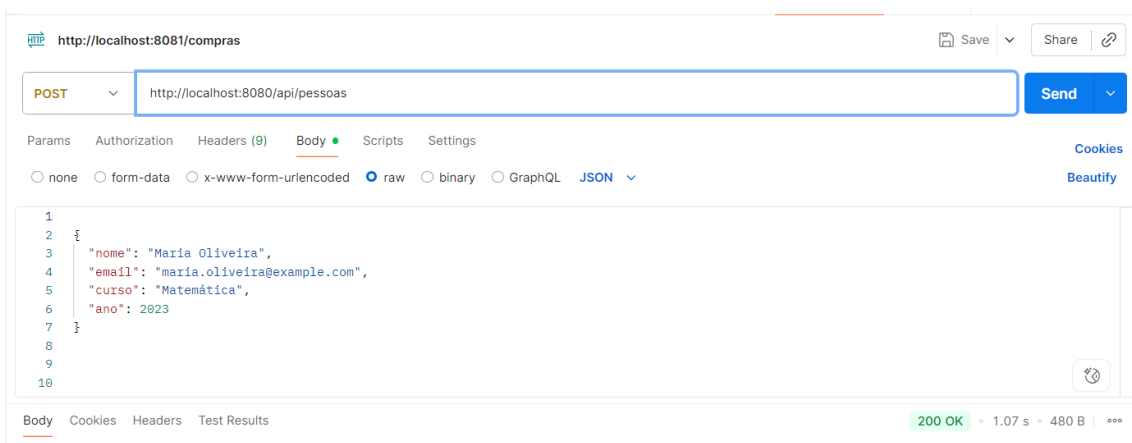
## Diagrama de Fluxo da API - Explicação em Texto

```
1 graph TD
2   A[Usuário envia requisição HTTP] --> B[PessoaController]
3   B --> C[PessoaService]
4   C --> D[PessoaRepository]
5   D --> E[Banco de Dados PostgreSQL]
6   C --> F[Validação e Lógica de Negócio]
7   B --> G[Resposta HTTP com dados ou erro]
8   C --> H[ResourceNotFoundException (se necessário)]
```



## Etapas do Fluxo

1. **Usuário** envia **requisição** via frontend (HTML + JS) ou Postman.
2. **PessoaController** recebe a **requisição** e chama o **serviço**.
3. **PessoaService** executa a **lógica de negócio**:
  - Salvar, buscar, atualizar ou deletar **pessoa**.
  - Valida os **dados**.
  - Lança **exceções** se necessário.
4. **PessoaRepository** acessa o **banco de dados PostgreSQL**.
5. O **banco** retorna os **dados** ou **confirma** a **operação**.
6. O **serviço** retorna os **dados** para o **controller**.
7. O **controller** envia a **resposta HTTP** para o **usuário**.



## Classe **ResourceNotFoundException**





### Função

Essa classe define uma **exceção personalizada** que será lançada quando o sistema não encontrar um recurso solicitado — por exemplo, ao tentar atualizar ou deletar uma pessoa que não existe no banco de dados.

### Explicação do Código





```
1 @ResponseStatus(HttpStatus.NOT_FOUND)
2
```

- Indica que, ao lançar essa exceção, o sistema deve retornar o **status HTTP 404 (Not Found)** para o cliente.

    </> Java

```
1 public class ResourceNotFoundException extends RuntimeException
2
```

- A classe herda de `RuntimeException`, permitindo que seja lançada sem necessidade de tratamento obrigatório (try/catch).

    </> Java

```
1 public ResourceNotFoundException(String mensagem) {
2     super(mensagem);
3 }
4
```

- Construtor que recebe uma mensagem personalizada, que será exibida no erro.

## Classe `GlobalExceptionHandler`

### Localização:

```
src/main/java/com/alunoskesley/professoreskesley/exceptions/GlobalExceptionHandler.java
```

### Função

Essa classe intercepta erros que ocorrem na aplicação e retorna respostas HTTP personalizadas para o cliente. Ela evita que mensagens técnicas ou confusas sejam exibidas diretamente.

### Anotação Principal

    </> Java

```
1 @RestControllerAdvice
2
```

- Indica que esta classe **observa todas as exceções** lançadas pelos controladores (`@RestController`) e pode tratá-las de forma personalizada.

## Métodos de Tratamento de Erros

### 1. `handleResourceNotFound`

Trata exceções do tipo `ResourceNotFoundException`.

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<Map<String, String>> handleResourceNotFound(ResourceNotFoundException ex)
{
    Map<String, String> erro = new HashMap<>();
    erro.put("erro", "Recurso não encontrado");
    erro.put("mensagem", ex.getMessage());
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(erro);
}
```

- Retorna **status 404 (Not Found)**.
- Envia uma mensagem amigável e a descrição do erro.

### 2. `handleGenericException`

Trata qualquer outra exceção genérica que não tenha sido capturada.

```
1 @ExceptionHandler(Exception.class)
2 public ResponseEntity<Map<String, String>> handleGenericException(Exception ex) {
3     Map<String, String> erro = new HashMap<>();
4     erro.put("erro", "Erro interno");
5     erro.put("mensagem", ex.getMessage());
6     return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(erro);
7 }
```

- Retorna **status 500 (Internal Server Error)**.
- Evita que o usuário veja mensagens técnicas ou stack traces.

## Cadastro de Alunos

Nome

Email

Curso

Ano

Cadastrar

Pessoas cadastradas: 0



### Arquivo `pessoa.html` – Interface Web

#### Localização:

```
src/main/resources/front_end/pessoa.html
```

#### Função

Este arquivo define a **estrutura visual da página** onde os usuários podem cadastrar alunos. Ele utiliza HTML para a estrutura, CSS para o estilo (via `pessoa.css`) e JavaScript para a lógica (via `pessoa.js`).

## Explicação dos Componentes

### ◆ <head>

```
1 <link rel="stylesheet" href="pessoa.css">
2
```

- Importa o arquivo CSS que define o estilo da página.

### ◆ <form id="cadastroForm">

Formulário de cadastro com os seguintes campos:

```
1 <input type="text" id="nome" placeholder="Nome" required>
2 <input type="email" id="email" placeholder="Email" required>
3 <input type="text" id="curso" placeholder="Curso" required>
4 <input type="number" id="ano" placeholder="Ano" required>
5 <button type="submit">Cadastrar</button>
6
```

- Os campos são obrigatórios (required).
- O botão envia os dados para o backend via JavaScript.

### ◆ <div id="message">

Área onde será exibida a mensagem de sucesso ou erro após o envio do formulário.

### ◆ <div id="contador">

Exibe o número de pessoas cadastradas. Este valor pode ser atualizado dinamicamente via JavaScript.

### ◆ 

Imagem decorativa da página, com o logo da aplicação.

## Arquivo `pessoa.css` – Estilização da Interface

### Localização:

```
src/main/resources/front_end/pessoa.css
```

### Função

Define o visual da página HTML, como cores, espaçamento, fontes, botões e responsividade para dispositivos móveis.

## Principais Estilos Definidos

### `body`

```
1  body {  
2      font-family: Arial, sans-serif;  
3      background-color: #f4f4f4;  
4      display: flex;  
5      flex-direction: column;  
6      align-items: center;  
7  }  
8
```

- Define a fonte padrão.
- Centraliza o conteúdo verticalmente.
- Aplica cor de fundo suave.

## ◆ h1

```
1 h1 {  
2     margin-top: 30px;  
3     color: #333;  
4 }  
5
```

- Define o título da página com espaçamento e cor escura.

## ◆ form

```
1 form {  
2     background-color: #fff;  
3     padding: 20px 30px;  
4     border-radius: 8px;  
5     box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);  
6 }  
7
```

- Cria um cartão branco com sombra e bordas arredondadas.
- Dá destaque ao formulário de cadastro.

## ◆ input e button

```
1  form input {  
2      width: 100%;  
3      padding: 8px;  
4      border-radius: 4px;  
5  }  
6  button {  
7      background-color: #28a745;  
8      color: white;  
9      cursor: pointer;  
10 }  
11 button:hover {  
12     background-color: #218838;  
13 }
```

- Os campos de entrada são estilizados para serem amigáveis.
- O botão tem cor verde e muda ao passar o mouse.



## ◆ #mensagem e #contador

```
1  #mensagem {  
2      font-weight: bold;  
3      color: #333;  
4  }  
5  #contador {  
6      font-size: 18px;  
7      font-weight: bold;  
8      color: #444;  
9      display: flex;  
10     gap: 10px;  
11 }  
12
```

- Estiliza a área de mensagens e o contador de cadastros.





## ◆ .lion-image

```
1  .lion-image {  
2      width: 200px;  
3      height: 200px;  
4      object-fit: contain;  
5  }  
6
```

- Define o tamanho da imagem do leão e garante que ela se encaixe bem na tela.

## Responsividade

```
1 @media (max-width: 480px) {  
2     form {  
3         padding: 15px 20px;  
4     }  
5     button {  
6         font-size: 14px;  
7     }  
8 }  
9
```

    CSS

- Adapta o layout para telas pequenas (como celulares).
- Reduz o tamanho de fontes e espaçamentos.

## Arquivo `pessoa.js` – Lógica do Frontend

### Localização:

```
src/main/resources/front_end/pessoa.js
```

## Função

Este script JavaScript:

- Captura os dados do formulário HTML.
- Envia os dados para a API REST usando `fetch`.
- Atualiza a interface com mensagens e contador de cadastros.

## Explicação do Código

### ◆ DOMContentLoaded

```
1 document.addEventListener("DOMContentLoaded", function () {  
2     atualizarContador();  
3 });  
4
```

- Quando a página é carregada, chama a função `atualizarContador()` para mostrar quantas pessoas já estão cadastradas.

### ◆ Envio do Formulário

```
1 document.getElementById("cadastroForm").addEventListener("submit", function (e) {  
2     e.preventDefault();  
3     ...  
4 });  
5
```

- Intercepta o envio do formulário para evitar o recarregamento da página.
- Captura os dados digitados pelo usuário.

### ◆ Envio para a API

```
1 fetch("http://localhost:8080/api/pessoas", {  
2     method: "POST",  
3     headers: {  
4         "Content-Type": "application/json"  
5     },  
6     body: JSON.stringify(pessoa)  
7 })  
8
```

- Envia os dados para o backend via método POST.
- O backend recebe e salva a nova pessoa no banco de dados.

### ◆ Tratamento de Resposta

```
1 .then(response => {
2   if (response.ok) {
3     document.getElementById("mensagem").textContent = "Cadastro realizado com sucesso!";
4     document.getElementById("cadastroForm").reset();
5     atualizarContador();
6   } else {
7     ...
8   }
9 })
0 .catch(error => {
1   document.getElementById("mensagem").textContent = error.message;
2 });
3
```

- Se o cadastro for bem-sucedido, exibe uma mensagem e limpa o formulário.
- Se houver erro, exibe a mensagem de erro.

### ◆ Atualização do Contador

```
function atualizarContador() {
  fetch("http://localhost:8080/api/pessoas")
    .then(response => response.json())
    .then(data => {
      document.getElementById("contador").textContent = `Pessoas cadastradas: ${data.length}`;
    })
    .catch(error => {
      console.error("Erro ao buscar pessoas:", error);
    });
}
```

- Faz uma requisição GET para buscar todas as pessoas cadastradas.
- Atualiza o contador na interface com o número total.



## 1. Dockerfile – Empacotando a Aplicação

Crie um arquivo chamado Dockerfile na raiz do projeto:

```
1  # Usa uma imagem base do Java
2  FROM openjdk:17-jdk-slim
3
4  # Define o diretório de trabalho dentro do container
5  WORKDIR /app
6
7  # Copia o arquivo JAR gerado pelo Spring Boot
8  COPY target/professoreskesley.jar app.jar
9
10 # Expõe a porta usada pela aplicação
11 EXPOSE 8080
12
13 # Comando para rodar a aplicação
14 ENTRYPOINT ["java", "-jar", "app.jar"]
15
```

💡 Certifique-se de que o JAR foi gerado com `mvn clean package` ou `./gradlew build`.

## 🔧 2. docker-compose.yml – Aplicação + PostgreSQL

Crie um arquivo `docker-compose.yml` na raiz do projeto:

```
1  version: '3.8'
2
3  services:
4    postgres:
5      image: postgres:15
6      container_name: postgres_db
7      environment:
8        POSTGRES_DB: professoreskesley
9        POSTGRES_USER: kesley
10       POSTGRES_PASSWORD: senha123
11      ports:
12        - "5432:5432"
13      volumes:
14        - pgdata:/var/lib/postgresql/data
```

```
15
16 app:
17   build: .
18   container_name: professoreskesley_app
19   ports:
20     - "8080:8080"
21   depends_on:
22     - postgres
23   environment:
24     SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/professoreskesley
25     SPRING_DATASOURCE_USERNAME: kesley
26     SPRING_DATASOURCE_PASSWORD: senha123
27
28 volumes:
29   pgdata:
```



## Como Subir a Aplicação com Docker

### 1. Gere o JAR da aplicação:

```
1 ./gradlew build
2
```

### 2. Execute o Docker Compose:

```
1 docker-compose up --build
2
```

### 3. Acesse a aplicação em:

`http://localhost:8080`