

CHECKPOINT 3

ARQUITETURA ORIENTADA A SERVIÇOS – 3ESPW

Integrantes:

- Julia Azevedo Lins – RM 98690
- Luís Gustavo Barreto Garrido – RM 99210
- Victor Hugo Aranda Forte – RM 99750

Sumário

Introdução.....	2
Estrutura do Projeto.....	2
Controller	2
Service.....	2
DTO (Data Transfer Object)	2
Model.....	2
Repository	2
Testes via Swagger	3
Endpoints Explicados	3
Criar um Paciente (POST /pacientes)	3
Listar todos os Pacientes (GET /pacientes)	4
Buscar Paciente por ID (GET /pacientes/{id})	4
Atualizar Paciente (PUT /pacientes/{id}).....	5
Deletar Paciente (DELETE /pacientes/{id}).....	6

Introdução

Este projeto consiste em uma API RESTful desenvolvida em Java com Spring Boot, utilizando arquitetura em camadas para promover organização, manutenção e escalabilidade do código. O objetivo principal é permitir o cadastro, consulta, atualização e exclusão de pacientes, seguindo boas práticas de desenvolvimento orientado a objetos, persistência de dados e documentação de API via Swagger UI.

A aplicação está integrada ao banco de dados MySQL, containerizado com Docker, e segue os padrões estabelecidos para o checkpoint da FIAP.

Estrutura do Projeto

Controller

- **Função:** Responsável por receber e responder às requisições HTTP.
- **Classe principal:** PacienteController
- **Explicação:** Aqui estão mapeados os endpoints (como GET, POST, PUT e DELETE), que chamam os métodos da camada de serviço. Essa camada é a "porta de entrada" para o sistema.

Service

- **Função:** Implementa a lógica de negócio da aplicação.
- **Classe principal:** PacienteService
- **Explicação:** Contém a inteligência da aplicação, como validações de dados, tratamento de erros e chamadas ao repositório. Garante que regras como "nome e email não podem ser nulos" sejam respeitadas.

DTO (Data Transfer Object)

- **Função:** Definem os formatos dos dados recebidos e enviados.
- **Classes utilizadas:**
 - PacienteRequestCreate: dados para criar paciente.
 - PacienteRequestUpdate: dados para atualização.
 - PacienteResponse: dados retornados ao usuário.
- **Explicação:** Usamos os DTOs para evitar expor diretamente a estrutura do banco de dados e para facilitar a validação e a segurança.

Model

- **Função:** Representa a entidade que será persistida no banco de dados.
- **Classe principal:** Paciente
- **Explicação:** Contém os atributos mapeados com anotações JPA como @Entity, @Id, @GeneratedValue, etc. Esta classe reflete a tabela no banco.

Repository

- **Função:** Acesso ao banco de dados via JPA.
- **Interface:** PacienteRepository
- **Explicação:** Estende JpaRepository, fornecendo métodos prontos como save, findById, deleteById e findAll.

Testes via Swagger

- Criar um paciente (POST /pacientes)
- Listar todos (GET /pacientes)
- Buscar por ID (GET /pacientes/{id})
- Atualizar (PUT /pacientes/{id})
- Excluir (DELETE /pacientes/{id})

Endpoints Explicados

Criar um Paciente (POST /pacientes)

Este endpoint permite o cadastro de um novo paciente. É necessário informar nome, CPF e email. Todos os campos são obrigatórios.

Exemplo de uso no Swagger:

- Preencha os dados no corpo da requisição e clique em Execute.
- Você receberá de volta os dados do paciente criado com seu respectivo ID.

The screenshot displays the Swagger UI interface for the POST /pacientes endpoint. The top bar shows the method 'POST' and the path '/pacientes'. Below this, the 'Parameters' section is empty, with 'No parameters' displayed. The 'Request body' section is marked as 'required' and has a dropdown menu set to 'application/json'. The request body is a JSON object:

```
{  "nome": "Carlos",  "cpf": "98765432100",  "email": "carlos@exemplo.com"}
```

. The bottom section shows the 'Server response' with a status code of '201' and a 'Response body' containing the created patient data:

```
{  "id": 2,  "nome": "Carlos",  "cpf": "98765432100",  "email": "carlos@exemplo.com"}
```

. A 'Download' button is visible next to the response body.

Listar todos os Pacientes (GET /pacientes)

Retorna uma lista com todos os pacientes cadastrados no sistema. Pode ser usado para verificar o conteúdo atual da base de dados.

Exemplo de uso:

- Clique em Execute e confira os pacientes listados.

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** /pacientes
- Parameters:** No parameters
- Buttons:** Execute (blue), Clear (grey)
- Responses:**
 - Curl:**

```
curl -X 'GET' \
  'http://localhost:8080/pacientes' \
  -H 'accept: */*'
```
 - Request URL:** http://localhost:8080/pacientes
 - Server response:**

Code	Details
200	<div>Response body</div> <pre>{ "id": 2, "nome": "Carlos", "cpf": "98765432100", "email": "carlos@exemplo.com" }</pre>

Buscar Paciente por ID (GET /pacientes/{id})

Busca um paciente específico pelo seu identificador único (ID). Se o ID não existir, uma mensagem de erro será exibida.

Exemplo de validação:

- Realizamos um teste buscando o ID 2 com sucesso.

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** /pacientes/{id}
- Parameters:**

Name	Description
id * required	
integer(\$int64)	2
(path)	
- Buttons:** Execute (blue), Clear (grey)
- Responses:**
 - Curl:**

```
curl -X 'GET' \
  'http://localhost:8080/pacientes/2' \
  -H 'accept: */*'
```
 - Request URL:** http://localhost:8080/pacientes/2
 - Server response:**

Code	Details
200	<div>Response body</div> <pre>{ "id": 2, "nome": "Carlos", "cpf": "98765432100", "email": "carlos@exemplo.com" }</pre>

Atualizar Paciente (PUT /pacientes/{id})

Atualiza os dados de um paciente existente. É necessário informar o ID na URL e os novos valores de nome e email no corpo da requisição.

Atenção:

- O CPF não pode ser alterado após a criação.

PUT /pacientes/{id}

Parameters

Name	Description
id * required	
integer(\$int64)	2
(path)	

Cancel

Reset

Request body * required

application/json

```
{  "nome": "Carlos Silva",  "email": "silva@exemplo.com"}
```

Request URL

http://localhost:8080/pacientes/2

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "id": 2, "nome": "Carlos Silva", "cpf": "98765432100", "email": "silva@exemplo.com"}</pre></div><div><div>Download</div></div></div> <div><div>Response headers</div><div><pre>connection: keep-alivecontent-type: application/jsondate: Mon, 05 May 2025 23:03:36 GMTkeep-alive: timeout=60transfer-encoding: chunked</pre></div></div>

Deletar Paciente (DELETE /pacientes/{id})

Remove um paciente do sistema com base no ID informado. Se o ID for válido, o paciente será removido permanentemente.

The screenshot shows a REST client interface with a red header bar. The method is **DELETE** and the URL is `/pacientes/{id}`. The **Parameters** tab is active, showing a required parameter `id` of type `integer($int64)` with the value `2`. Below the parameters are **Execute** and **Clear** buttons. The **Responses** section shows a **curl** command: `curl -X 'DELETE' \n 'http://localhost:8080/pacientes/2' \n -H 'accept: */*' \n`. The **Request URL** is `http://localhost:8080/pacientes/2`. The **Server response** shows a **Code** of `204` and **Details** including **Response headers**: `connection: keep-alive`, `date: Mon, 05 May 2025 23:04:42 GMT`, and `keep-alive: timeout=60`.

Para validar o delete, executamos um get no ID 2:

The screenshot shows the same REST client interface, but the method is **GET**. The **Parameters** tab is active, showing the same required parameter `id` of type `integer($int64)` with the value `2`. Below the parameters are **Execute** and **Clear** buttons. The **Responses** section shows a **curl** command: `curl -X 'GET' \n 'http://localhost:8080/pacientes/2' \n -H 'accept: */*' \n`. The **Request URL** is `http://localhost:8080/pacientes/2`. The **Server response** shows a **Code** of `404` and **Details** including **Response headers**: `connection: keep-alive`, `content-length: 0`, `date: Mon, 05 May 2025 23:04:49 GMT`, and `keep-alive: timeout=60`.