Projeto*:* Sistema de Agendamento Hospitalar.

Equipe:

**Alexsandro Mourão de Oliveira - rm360062**

**Anderson Napoleão – rm360054**

O projeto se encontra no endereço do github:

<https://github.com/amouraorr/projeto-fiap-hospitalar.git>

Lá, no arquivo README.md, temos informações detalhadas de como executar a aplicação entre outras informações.

**1.Introducão**

**Descrição do problema**

Em um ambiente hospitalar, é essencial contar com sistemas que garantam o agendamento eficaz de consultas, o gerenciamento do histórico de pacientes e o envio de lembretes automáticos para garantir a presença dos pacientes nas consultas. Este sistema deve ser acessível a diferentes tipos de usuários (médicos, enfermeiros e pacientes), com acesso controlado e funcionalidades específicas para cada perfil.

**Objetivo do projeto**

- Desenvolver um backend completo utilizando Spring Boot.

- Implementar operações de gerenciamento de agendamentos, notificações e histórico de consultas.

- Configurar o ambiente com Docker e Docker Compose para facilitar a replicação e escalabilidade.

- Garantir a segurança e a manutenção do sistema.

**2.Arquitetura do Sistema**

**Descricão da Arquitetura**

Arquitetura

A arquitetura do projeto é baseada no padrão MVC (Model-View-Controller), que promove a separação de preocupações e facilita a manutenção e escalabilidade da aplicação. A estrutura do sistema é organizada em pacotes que separam as responsabilidades:

- \*\*Model\*\*: Contém as classes que representam as entidades do sistema, como `Appointment`, `Notification`, e `History`. Essas classes definem a estrutura dos dados que serão manipulados pela aplicação.

- \*\*Controller\*\*: Contém as classes que gerenciam as requisições HTTP e interagem com os serviços, como `AppointmentController`, `NotificationController`, e `HistoryController`. Os controladores são responsáveis por receber as solicitações do cliente, processá-las e retornar as respostas apropriadas.

- \*\*Service\*\*: Contém a lógica de negócios, como `AppointmentService`, `NotificationService`, e `HistoryService`, que manipulam as operações relacionadas a agendamentos, notificações e histórico. Essa camada é responsável por implementar as regras de negócio e orquestrar as interações entre os repositórios e os controladores.

- \*\*Repository\*\*: Contém as interfaces que abstraem a lógica de acesso a dados, como `AppointmentRepository`, `NotificationRepository`, e `HistoryRepository`. Os repositórios são responsáveis por realizar operações de CRUD (Create, Read, Update, Delete) nas entidades.

- \*\*Mapper\*\*: Utiliza o MapStruct para converter entre entidades e DTOs (Data Transfer Objects), facilitando a transferência de dados entre camadas. Isso ajuda a desacoplar a representação interna das entidades da API.

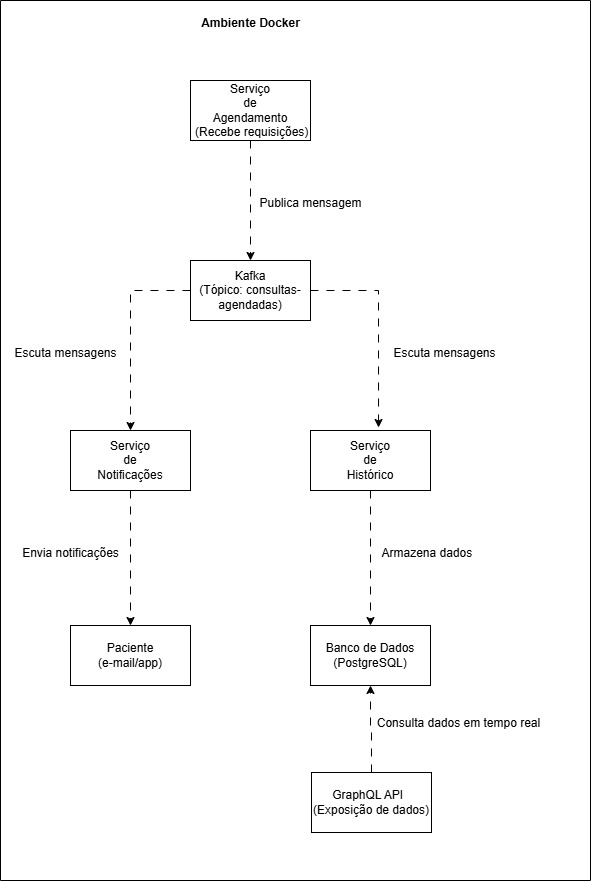
- \*\*Exception\*\*: Contém classes para tratamento de exceções personalizadas. Essa camada é responsável por capturar e gerenciar erros que podem ocorrer durante a execução da aplicação.

- \*\*Security\*\*: Contém configurações e classes relacionadas à segurança da aplicação, como autenticação e autorização. Isso pode incluir a configuração do Spring Security para proteger endpoints e gerenciar usuários.

- \*\*Swagger\*\*: Configurações para a documentação da API usando Swagger/OpenAPI. Isso permite que os desenvolvedores visualizem e testem os endpoints da API de forma interativa.

A aplicação é conteinerizada usando Docker, permitindo uma implantação fácil e consistente em diferentes ambientes. O sistema utiliza o PostgreSQL como banco de dados relacional, escolhido por sua robustez e suporte a transações complexas. As requisições do cliente são recebidas pelos controladores, que delegam a lógica de negócios para os serviços. Os serviços, por sua vez, interagem com os repositórios para acessar e manipular os dados no banco de dados.

**Diagrama da Arquitetura**



**3.Descricão dos Endpoints da API**

**Tabela de Endpoints**

|  |  |  |
| --- | --- | --- |
| Endpoint | Método | Descrição |
| /appointments/{id} | GET | Busca consulta médica por ID |
| /appointments/{id} | DELETE | Excluir consulta médica e todos os dados relacionados |
| /appointments/{id} | PUT | Atualiza consulta médica |
| /appointments | GET | Buscar todas consultas médicas |
| /appointments | POST | Criação de nova consulta médica |

**Exemplos de requisição e resposta**

1. \*\*Criar Consulta Médica\*\*

- \*\*Endpoint:\*\* POST /appointments

- \*\*Descrição:\*\* Cria uma nova consulta médica.

- \*\*Requisição:\*\*

```http

POST /appointments HTTP/1.1

Host: localhost:8080

Content-Type: application/json

{

"paciente": "Nome do Paciente",

"data": "2023-10-01T10:00:00",

"medicoId": 1

}

```

- \*\*Resposta:\*\*

```http

HTTP/1.1 201 Created

Content-Type: application/json

{

"data": {

"id": 1,

"paciente": "Nome do Paciente",

"data": "2023-10-01T10:00:00",

"medicoId": 1

}

}

```

2. \*\*Buscar Consulta Médica por ID\*\*

- \*\*Endpoint:\*\* GET /appointments/{id}

- \*\*Descrição:\*\* Busca consulta médica por ID.

- \*\*Requisição:\*\*

```http

GET /appointments/1 HTTP/1.1

Host: localhost:8080

```

- \*\*Resposta:\*\*

```http

HTTP/1.1 200 OK

Content-Type: application/json

{

"data": {

"id": 1,

"paciente": "Nome do Paciente",

"data": "2023-10-01T10:00:00",

"medicoId": 1

}

}

3. \*\*Excluir Consulta Médica\*\*

- \*\*Endpoint:\*\* DELETE /appointments/{id}

- \*\*Descrição:\*\* Excluir consulta médica e todos os dados relacionados.

- \*\*Requisição:\*\*

```http

DELETE /appointments/1 HTTP/1.1

Host: localhost:8080

```

- \*\*Resposta:\*\*

```http

HTTP/1.1 204 No Content

4. \*\*Atualizar Consulta Médica\*\*

- \*\*Endpoint:\*\* PUT /appointments/{id}

- \*\*Descrição:\*\* Atualiza consulta médica.

- \*\*Requisição:\*\*

```http

PUT /appointments/1 HTTP/1.1

Host: localhost:8080

Content-Type: application/json

{

"paciente": "Nome do Paciente Atualizado",

"data": "2023-10-02T10:00:00",

"medicoId": 1

}

```

- \*\*Resposta:\*\*

```http

HTTP/1.1 200 OK

Content-Type: application/json

{

"data": {

"id": 1,

"paciente": "Nome do Paciente Atualizado",

"data": "2023-10-02T10:00:00",

"medicoId": 1

}

}

5. \*\*Listar Todas as Consultas Médicas\*\*

- \*\*Endpoint:\*\* GET /appointments

- \*\*Descrição:\*\* Buscar todas as consultas médicas.

- \*\*Requisição:\*\*

```http

GET /appointments HTTP/1.1

Host: localhost:8080

```

- \*\*Resposta:\*\*

```http

HTTP/1.1 200 OK

Content-Type: application/json

{

"data": [

{

"id": 1,

"paciente": "Nome do Paciente",

"data": "2023-10-01T10:00:00",

"medicoId": 1

},

{

"id": 2,

"paciente": "Outro Paciente",

"data": "2023-10-02T11:00:00",

"medicoId": 2

}

]

}

1. **Utilização do GraphiQL**

O que é GraphiQL?

GraphiQL é uma ferramenta de interface gráfica que permite aos desenvolvedores interagir com APIs GraphQL de forma intuitiva. Com ela, é possível realizar consultas, mutações e explorar o esquema da API de maneira visual.

Acessando o GraphiQL

Para acessar a interface do GraphiQL, inicie o servidor da aplicação e navegue até o seguinte endereço no seu navegador:

http://localhost:8081/graphiql.html

Realizando Consultas

Na interface do GraphiQL, você pode realizar consultas para obter dados do histórico de consultas. Aqui estão alguns exemplos de consultas que podem ser realizadas:

# 1. Consulta para retornar todos os históricos de consultas

query {

getAllHistories {

id

paciente

medico

enfermeiro

dataHora

}

}

\*\*Descrição:\*\* Esta consulta retorna todos os registros do histórico de consultas, incluindo informações sobre paciente, médico, enfermeiro e data/hora.

# 2. Consulta para retornar o histórico de um paciente específico

query {

getPatientHistory(paciente: "Nome do Paciente") {

id

medico

enfermeiro

dataHora

}

}

\*\*Descrição:\*\* Esta consulta retorna o histórico de consultas para um paciente específico, permitindo que você veja todas as consultas realizadas por ele.

# 3. Consulta para retornar históricos de um médico específico

query {

getHistoriesByMedico(medico: "Nome do Medico") {

id

paciente

enfermeiro

dataHora

}

}

\*\*Descrição:\*\* Esta consulta retorna todas as consultas realizadas por um médico específico.

# 4. Consulta para retornar históricos de um enfermeiro específico

query {

getHistoryByEnfermeiro(enfermeiro: "Nome do Enfermeiro") {

id

paciente

medico

dataHora

}

}

\*\*Descrição:\*\* Esta consulta retorna todas as consultas realizadas por um enfermeiro específico.

# 5. Mutation para adicionar um novo histórico de consulta

mutation {

addHistory(

paciente: "Nome do Paciente"

medico: "Nome do Medico"

enfermeiro: "Nome do Enfermeiro"

dataHora: "2024-06-01T10:00:00"

) {

id

paciente

medico

enfermeiro

dataHora

}

}

\*\*Descrição:\*\* Adiciona um novo registro ao histórico de consultas.

# 6. Consulta simples de teste

query {

hello

}

\*\*Descrição:\*\* Consulta de teste para verificar se a API GraphQL está funcionando.

### Explorando o Esquema

A interface do GraphiQL também permite que você explore o esquema GraphQL da API. Você pode visualizar os tipos de dados disponíveis, as consultas e mutações que podem ser realizadas, e os campos que cada tipo contém. Isso facilita a compreensão da estrutura da API e ajuda na construção de consultas mais complexas.

### Conclusão

A utilização do GraphiQL proporciona uma maneira prática e eficiente de interagir com a API GraphQL do serviço de histórico, permitindo que desenvolvedores e usuários testem e explorem as funcionalidades disponíveis de forma intuitiva.

**5. Configuracão do Projeto**

**Configuração do Docker Compose**

Arquivo docker-compose.yml

O arquivo `docker-compose.yml` é responsável por orquestrar a aplicação e o banco de dados, definindo os serviços necessários para o funcionamento do sistema.

**Instruções para execução local**

Consultar arquivo README.md no repositório do projeto

**6.Qualidade do Código**

**Boas Práticas Utilizadas**

O projeto adota várias boas práticas e padrões de design que garantem a qualidade e a manutenibilidade do código. Abaixo estão algumas das principais práticas seguidas:

1. \*\*DRY (Don't Repeat Yourself)\*\*: O princípio DRY é aplicado para evitar a duplicação de código.

2. \*\*SOLID\*\*: Os princípios SOLID são seguidos para garantir um design de software robusto e flexível.

3. \*\*Padrões de Projeto\*\*: O projeto utiliza padrões de design como MVC, Repository Pattern e Service Layer Pattern.

4. \*\*Tratamento de Exceções\*\*: O tratamento de exceções é centralizado usando `@ControllerAdvice`.

5. \*\*Documentação\*\*: A documentação da API é gerada automaticamente usando Swagger/OpenAPI.

Conclusão

Essas boas práticas e padrões de design não apenas melhoram a qualidade do código, mas também facilitam a colaboração entre desenvolveres, a manutenção do sistema e a adição de novas funcionalidades ao longo do tempo.

**7.Collections para Teste**

**Link para a Collection do Postman**

<https://github.com/amouraorr/projeto-fiap-hospitalar/blob/main/postman-collections/pos-fiap-2025-agendamento-hospitalar.postman_collection.json>

**Descricão dos Testes Manuais**

Para validar os endpoints da API, utilizamos Collections do Postman, que contêm requisições pré-configuradas para cada um dos endpoints disponíveis. Abaixo estão as etapas para realizar os testes manuais usando essas Collections.

Passos para Validar os Endpoints

1. \*\*Importar a Collection\*\*:

- Abra o Postman.

- Clique em "Import" no canto superior esquerdo.

- Se você tiver um arquivo JSON da Collection, arraste e solte o arquivo ou selecione-o. Se você tiver um link, cole-o na aba "Link" e clique em "Import".

2. \*\*Selecionar a Collection\*\*:

- Após a importação, localize a Collection na barra lateral esquerda do Postman.

- Clique na Collection para expandi-la e visualizar todos os endpoints disponíveis.

3. \*\*Executar os Testes\*\*:

- Para cada endpoint, você verá uma lista de requisições. Selecione a requisição que deseja testar (por exemplo, `POST /users` para criar um usuário).

- Clique na requisição para abrir os detalhes.

4. \*\*Configurar os Parâmetros\*\*:

- Se a requisição requer parâmetros ou um corpo (body), preencha os campos necessários. Por exemplo, para criar um usuário, você deve fornecer um JSON com os dados do usuário no corpo da requisição.

- Exemplo de corpo para a requisição `POST /users`:

```json

{

"nome": "Nome do Usuário",

"email": "usuario@example.com",

}

5. Enviar a Requisição:

Clique no botão "Send" para enviar a requisição ao servidor.

O Postman exibirá a resposta do servidor na parte inferior da tela.

6. Validar a Resposta:

Verifique o código de status da resposta. Por exemplo, para uma criação bem-sucedida, você deve receber um código 201 Created.

Analise o corpo da resposta para garantir que os dados retornados estão corretos e correspondem ao que foi enviado.

Exemplo de resposta esperada:

{

"data": {

"id": 1,

"nome": "Nome do Usuário"

}

}

7. Repetir para Outros Endpoints:

Repita os passos 3 a 6 para todos os outros endpoints que você deseja testar, como GET /users, PUT /users/{id}, DELETE /users/{id}, etc.

**8.Repositório do Código**

**URL do Repositório maven**

1. Clone o repositório:

bash

git clone https://github.com/amouraorr/projeto-fiap-hospitalar.git

branch main