Overview Geral do Projeto

Este documento fornece uma visão geral de dois sistemas distintos: um sistema de log em PHP e um controlador (MedicoController) para uma API em Java (Spring Framework) para gerenciamento de médicos.

Sistema de Log (PHP):

- **Propósito:** Registrar as ações realizadas por usuários no banco de dados.
- **Arquitetura:**
    - Utiliza um sistema de tokens de acesso para identificar usuários.
    - Registra informações detalhadas sobre as modificações (tabela, ID do registro, descrição da ação).
    - Depende do arquivo `db_conn.php` para conexão com o banco de dados.
    - Utiliza as tabelas `access_tokens`, `log_reference` e `system_logs`.
- **Funções Principais:**
    - `registra_modificacoes`: Recebe um array de modificações e as registra individualmente.
    - `registrar_log`: Registra um evento de log no banco de dados.
- **Fluxo de Execução:** Uma ação é realizada > `registra_modificacoes` é chamada > para cada modificação, `registrar_log` é chamada > `registrar_log` consulta o usuário pelo token e insere dados nas tabelas de log.
- **Segurança:** Proteção de tokens de acesso, descrições claras das ações e restrição de acesso às tabelas de log são cruciais.

API MedicoController (Java - Spring Framework):

- **Propósito:** Gerenciar informações de médicos através de uma API RESTful.
- **Arquitetura:**
    - Utiliza o Spring Framework para manipular requisições HTTP.
    - Interage com o `MedicoRepository` para persistência de dados.
    - A API é protegida por autenticação JWT (`@SecurityRequirement(name = "bearer-key")`).
- **Endpoints:**
    - `POST /medicos`: Cadastra um novo médico.
    - `GET /medicos`: Lista médicos ativos (paginado).
    - `PUT /medicos`: Atualiza informações de um médico existente.
    - `DELETE /medicos/{id}`: Exclui um médico (marca como inativo).
    - `GET /medicos/{id}`: Detalha informações de um médico.
- **Classes de Dados (DTOs):**
    - `DadosCadastroMedico`: Dados para cadastrar um médico.
    - `DadosAtualizacaoMedico`: Dados para atualizar um médico.
    - `DadosListagemMedico`: Dados para listagem de médicos.
    - `DadosDetalhamentoMedico`: Dados detalhados de um médico.
    - `DadosEndereco`: Dados de endereço de um médico.
- **Detalhes dos Métodos:**
    - Cada método do controlador (cadastrar, listar, atualizar, deletar, detalhar) corresponde a uma operação CRUD na entidade `Medico`.
- **Validação:** `@Valid` para garantir a integridade dos dados recebidos.
- **Transacionalidade:** `@Transactional` para garantir a consistência das operações no banco de dados.

Visualização da Estrutura de Pastas e Arquivos

Para ajudar na compreensão da estrutura do projeto, segue uma visualização simplificada da organização dos arquivos, baseado nas informações da documentação. É importante notar que essa é uma estrutura *presumida* baseada no texto fornecido, e a estrutura real pode variar.

Estrutura Presumida do Projeto (PHP - Sistema de Log):

```
raiz_do_projeto/
│
├── db_conn.php          # Arquivo de conexão com o banco de dados (PDO)
├── log_function.php   # Arquivo contendo as funções registra_modificacoes e registrar_log
└── [outras pastas e arquivos do sistema PHP]
```

Descrição:

- `db_conn.php`: Contém a lógica para estabelecer a conexão com o banco de dados, utilizando PDO. É provável que contenha informações de configuração como o nome do banco de dados, usuário, senha e host.
- `log_function.php`: Contém as funções principais para registrar as modificações no banco de dados. Ele importa o arquivo `db_conn.php` para poder acessar a conexão com o banco de dados.
- `[outras pastas e arquivos do sistema PHP]`: Representa o restante da estrutura do projeto PHP, que não foi detalhada na documentação. Poderia incluir arquivos para gerenciamento de usuários, tokens, rotas, etc.

Estrutura Presumida do Projeto (Java - API MedicoController):

```
raiz_do_projeto/
│
└── src/
    │
    └── main/
        │
        └── java/
            │
            └── med/
                │
                └── voll/
                    │
                    └── api/
                        │
                        ├── controller/
                        │   │
                        │   └── MedicoController.java  # Controlador REST para médicos
                        │
                        ├── domain/
                        │   │
                        │   └── medico/
                        │       │
                        │       ├── Medico.java                  # Entidade Medico (classe de modelo)
                        │       ├── MedicoRepository.java        # Interface para acesso ao banco de dados (JPA Repository)
                        │       ├── DadosCadastroMedico.java     # DTO para cadastro de médicos
                        │       ├── DadosAtualizacaoMedico.java    # DTO para atualização de médicos
                        │       ├── DadosListagemMedico.java    # DTO para listagem de médicos
                        │       ├── DadosDetalhamentoMedico.java   # DTO para detalhamento de médicos
                        │       └── Especialidade.java          # Enum para especialidades médicas
                        │
                        └── DadosEndereco.java              # DTO para informações de endereço
```

Descrição:

- `src/main/java/med/voll/api/controller/MedicoController.java`: Este é o arquivo principal da API, contendo a classe `MedicoController` que define os endpoints REST para gerenciamento de médicos.
- `src/main/java/med/voll/api/domain/medico/*`: Esta pasta contém as classes relacionadas à entidade `Medico`. Isso inclui a classe `Medico` (que representa a tabela de médicos no banco de dados), a interface `MedicoRepository` (que fornece métodos para acessar e manipular os dados dos médicos no banco de dados), e as classes DTO (Data Transfer Objects) que são usadas para transferir dados entre a API e o cliente.
- `src/main/java/med/voll/api/DadosEndereco.java`: Contém a classe `DadosEndereco`, que é um DTO para representar informações de endereço.
- `[outras pastas e arquivos do projeto Java]`: Representa o restante da estrutura do projeto Java, que não foi detalhada na documentação. Isso inclui arquivos de configuração do Spring, classes de serviço, exceptions, configurações de segurança, etc.

Considerações:

- **Nomes de Pastas e Pacotes:** A estrutura mostrada é uma convenção comum para projetos Java usando Maven ou Gradle, onde as pastas refletem a estrutura dos pacotes Java.
- **Frameworks e Bibliotecas:** O projeto Java utiliza o Spring Framework, JPA (Hibernate provavelmente), e Swagger/OpenAPI para documentação.
- **Dados Faltantes:** Essa é uma representação simplificada. Um projeto real teria mais arquivos e pastas para lidar com configurações, testes, recursos estáticos, etc.

Essa visualização combinada oferece um entendimento da organização dos arquivos em ambos os projetos, facilitando a localização e a compreensão dos componentes principais.

Documentação Técnica: Sistema de Log

Este documento descreve o funcionamento do sistema de log, incluindo as funções principais e como elas interagem para registrar modificações em um banco de dados.

Visão Geral

O sistema de log foi projetado para registrar as ações realizadas por usuários em tabelas específicas do banco de dados. Ele utiliza tokens de acesso para identificar o usuário que realizou a ação e armazena informações detalhadas sobre a modificação, como a tabela afetada, o ID do registro e a descrição da ação.

Arquivo `log_function.php`

Este arquivo contém as funções principais para registrar as modificações. Ele depende do arquivo `db_conn.php` para estabelecer a conexão com o banco de dados.

Dependências

- **`db_conn.php`**: Responsável por estabelecer a conexão com o banco de dados usando PDO.

Funções

`registra_modificacoes(array $modificacoes, $token, $id, $table)`

Esta função recebe um array de modificações e registra cada uma delas individualmente.

Descrição: Itera sobre o array de modificações e chama a função `registrar_log` para cada modificação.

Parâmetros:

Parâmetro	Tipo	Descrição
`$modificacoes`	`array`	Um array contendo as descrições das modificações a serem registradas.
`$token`	`string`	O token de acesso do usuário que realizou as modificações.
`$id`	`int`	O ID do registro na tabela que foi modificado.
`$table`	`string`	O nome da tabela que foi modificada.

Retorno:

- `true`: Se todas as modificações foram registradas com sucesso.
- `false`: Se o array de modificações estiver vazio.

Exemplo de Uso:

```
$modificacoes = [
    "Nome alterado de 'João' para 'José'",
    "Email alterado de 'joao@example.com' para 'jose@example.com'"
];
$token = "seu_token_de_acesso";
$id = 123;
$table = "usuarios";

$resultado = registra_modificacoes($modificacoes, $token, $id, $table);

if ($resultado) {
    echo "Modificações registradas com sucesso!";
} else {
    echo "Nenhuma modificação para registrar.";
}
```

`registrar_log($token, $id, $table, $action)`

Esta função registra um evento de log no banco de dados.

Descrição:

1. Consulta o banco de dados para obter o nome de usuário associado ao token de acesso fornecido.
2. Insere uma entrada na tabela `log_reference` com o ID da referência e o nome da tabela.
3. Obtém o ID gerado automaticamente para a entrada na tabela `log_reference`.
4. Insere uma entrada na tabela `system_logs` com o nome de usuário, o ID da referência do log e a descrição da ação.

Parâmetros:

Parâmetro	Tipo	Descrição
`$token`	`string`	O token de acesso do usuário que realizou a ação.
`$id`	`int`	O ID do registro na tabela que foi modificado.
`$table`	`string`	O nome da tabela que foi modificada.
`$action`	`string`	Uma descrição da ação realizada (ex: "Usuário criado", "Registro atualizado", "Registro excluído").

Retorno:

- `true`: Se o log foi registrado com sucesso.
- `false`: Se o token de acesso não for encontrado no banco de dados.

Código-fonte Relevante:

```php
<?php
require_once 'db_conn.php';

function registra_modificacoes(array $modificacoes, $token, $id, $table){
    if (count($modificacoes) == 0){
        return false;
    }
    foreach ($modificacoes as $modificacao) {
        registrar_log($token, $id, $table, $modificacao);
    }
    return true;
}

function registrar_log($token, $id, $table, $action){
    global $pdo;

    $sql = "SELECT username from access_tokens where access_token = :access_token";
    $stmt = $pdo->prepare($sql);
    $stmt->bindParam(':access_token', $token);
    $stmt->execute();
    $query_token = $stmt->fetch(PDO::FETCH_OBJ);

    if (!$query_token) {
        return false;
    }
    $username = $query_token->username;

    $sql = "INSERT INTO log_reference (reference_id, table_name)
            VALUES (:reference_id, :table_name) RETURNING id";

    $stmt = $pdo->prepare($sql);
    $stmt->bindParam(':reference_id', $id, PDO::PARAM_INT);
    $stmt->bindParam(':table_name', $table, PDO::PARAM_STR);
    $stmt->execute();
    $insert_id = $stmt->fetchColumn();

    $sql = "INSERT INTO system_logs (username, log_reference_id, action)
            VALUES (:username, :log_reference_id, :action)";

    $stmt = $pdo->prepare($sql);
    $stmt->bindParam(':username', $username, PDO::PARAM_STR);
    $stmt->bindParam(':log_reference_id', $insert_id, PDO::PARAM_INT);
    $stmt->bindParam(':action', $action, PDO::PARAM_STR);
    $stmt->execute();
    return true;
}
?>
```

Tabelas do Banco de Dados

O sistema de log utiliza as seguintes tabelas no banco de dados:

- **`access_tokens`**: Armazena os tokens de acesso dos usuários e seus respectivos nomes de usuário.
    - `access_token` (VARCHAR): O token de acesso.
    - `username` (VARCHAR): O nome de usuário associado ao token.
- **`log_reference`**: Armazena a referência a qual tabela e ID o log se refere.
    - `id` (SERIAL PRIMARY KEY): O ID único do log de referência.
    - `reference_id` (INT): O ID do registro na tabela que foi modificada.
    - `table_name` (VARCHAR): O nome da tabela que foi modificada.
- **`system_logs`**: Armazena os logs do sistema.
    - `id` (SERIAL PRIMARY KEY): O ID único do log.
    - `username` (VARCHAR): O nome de usuário que realizou a ação.
    - `log_reference_id` (INT): O ID da referência do log na tabela `log_reference`.
    - `action` (VARCHAR): A descrição da ação realizada.

Fluxo de Execução

1. Uma ação é realizada no sistema que requer registro (ex: atualização de um registro).
2. A função `registra_modificacoes` é chamada, recebendo um array com as descrições de cada modificação.
3. Para cada modificação, a função `registrar_log` é chamada.
4. `registrar_log` consulta a tabela `access_tokens` para obter o nome de usuário associado ao token fornecido.
5. `registrar_log` insere uma nova entrada na tabela `log_reference` com o ID do registro modificado e o nome da tabela.
6. `registrar_log` insere uma nova entrada na tabela `system_logs` com o nome de usuário, o ID da referência do log e a descrição da ação.

Considerações de Segurança

- É crucial proteger os tokens de acesso para evitar o registro de ações em nome de outros usuários.
- As descrições das ações devem ser claras e concisas para facilitar a análise dos logs.
- O acesso às tabelas de log deve ser restrito a usuários autorizados.

Documentação Técnica: MedicoController

Este documento descreve o `MedicoController`, um componente crucial da API para gerenciamento de médicos. Ele expõe endpoints para cadastrar, listar, atualizar, deletar e detalhar informações sobre médicos.

Visão Geral

O `MedicoController` é um controlador REST que utiliza o Spring Framework para manipular requisições HTTP relacionadas a entidades `Medico`. Ele interage com o `MedicoRepository` para persistir e recuperar dados do banco de dados. A API é protegida por autenticação via token JWT, conforme definido pela anotação `@SecurityRequirement(name = "bearer-key")`.

Código-Fonte Relevante

```java
package med.voll.api.controller;

import io.swagger.v3.oas.annotations.security.SecurityRequirement;
import jakarta.validation.Valid;
import med.voll.api.domain.medico.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.web.PageableDefault;
import org.springframework.http.ResponseEntity;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponentsBuilder;

@RestController
@RequestMapping("medicos")
@SecurityRequirement(name = "bearer-key")
public class MedicoController {

    @Autowired
    private MedicoRepository repository;

    @PostMapping
    @Transactional
    public ResponseEntity cadastrar(@RequestBody @Valid DadosCadastroMedico dados, UriComponentsBuilder uriBuilder) {
        var medico = new Medico(dados);
        repository.save(new Medico(dados));

        var uri = uriBuilder.path("/medicos/{id}").buildAndExpand(medico.getId()).toUri();

        return ResponseEntity.created(uri).body(new DadosDetalhamentoMedico(medico));
    }

    @GetMapping
    public ResponseEntity<Page<DadosListagemMedico>> listar(@PageableDefault(size = 10, sort = {"nome"}) Pageable paginacao) {
        var page = repository.findAllByAtivoTrue(paginacao).map(DadosListagemMedico::new);
        return ResponseEntity.ok(page);
    }

    @PutMapping
    @Transactional
    public ResponseEntity atualizar(@RequestBody @Valid DadosAtualizacaoMedico dados) {
        var medico = repository.getReferenceById(dados.id());
        medico.atualizarInformacoes(dados);

        return ResponseEntity.ok(new DadosDetalhamentoMedico(medico));
    }

    @DeleteMapping("/{id}")
    @Transactional
    public ResponseEntity deletar(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();

        return ResponseEntity.noContent().build();
    }

    @GetMapping("/{id}")
    public ResponseEntity detalhar(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);

        return ResponseEntity.ok(new DadosDetalhamentoMedico(medico));
    }
}
```

Endpoints

A tabela a seguir descreve os endpoints expostos pelo `MedicoController`:

Método HTTP	Endpoint	Descrição	Request Body	Response Body
POST	`/medicos`	Cadastra um novo médico.	`DadosCadastroMedico`	`DadosDetalhamentoMedico`
GET	`/medicos`	Lista médicos ativos de forma paginada.	N/A	`Page<DadosListagemMedico>`
PUT	`/medicos`	Atualiza as informações de um médico existente.	`DadosAtualizacaoMedico`	`DadosDetalhamentoMedico`

DELETE	`/medicos/{id}`	Exclui um médico (marca como inativo).	N/A	`ResponseEntity.noContent()`
GET	`/medicos/{id}`	Detalha as informações de um médico específico.	N/A	`DadosDetalhamentoMedico`

DadosCadastroMedico

Estrutura de dados utilizada no corpo da requisição para cadastrar um novo médico.

Atributo	Tipo	Descrição	Validação
nome	String	Nome completo do médico.	`@NotBlank`
email	String	Endereço de email do médico.	`@NotBlank`, `@Email`
telefone	String	Número de telefone do médico.	`@NotBlank`
crm	String	Número do CRM (Conselho Regional de Medicina) do médico.	`@NotBlank`, Validação específica (ex: único)
especialidade	Especialidade	Especialidade médica do médico (enum).	`@NotNull`
endereco	DadosEndereco	Objeto contendo informações sobre o endereço do médico.	`@Valid`

DadosEndereco

Estrutura de dados que representa o endereço do médico.

Atributo	Tipo	Descrição	Validação
logradouro	String	Rua ou avenida do endereço.	`@NotBlank`
numero	String	Número do endereço.	`@NotBlank`
complemento	String	Complemento do endereço (opcional).	N/A
bairro	String	Bairro do endereço.	`@NotBlank`
cidade	String	Cidade do endereço.	`@NotBlank`
uf	String	Unidade Federativa (UF) do endereço.	`@NotBlank`, Tamanho 2
cep	String	Código de Endereçamento Postal (CEP).	`@NotBlank`

DadosListagemMedico

Estrutura de dados utilizada para listar médicos, retornada no endpoint GET `/medicos`.

Atributo	Tipo	Descrição
id	Long	ID do médico.
nome	String	Nome do médico.
email	String	Email do médico.
crm	String	CRM do médico.
especialidade	Especialidade	Especialidade do médico.

DadosAtualizacaoMedico

Estrutura de dados utilizada para atualizar as informações de um médico existente.

Atributo	Tipo	Descrição	Validação
id	Long	ID do médico a ser atualizado.	`@NotNull`
nome	String	Novo nome do médico (opcional).	N/A
telefone	String	Novo número de telefone do médico (opcional).	N/A
endereco	DadosEndereco	Novas informações de endereço do médico (opcional).	`@Valid`

DadosDetalhamentoMedico

Estrutura de dados retornada para detalhar as informações de um médico específico (GET `/medicos/{id}`).

Atributo	Tipo	Descrição
id	Long	ID do médico.
nome	String	Nome completo do médico.
email	String	Endereço de email do médico.
telefone	String	Número de telefone do médico.
crm	String	Número do CRM (Conselho Regional de Medicina) do médico.
especialidade	Especialidade	Especialidade médica do médico (enum).
endereco	DadosEndereco	Objeto contendo informações sobre o endereço do médico.
ativo	Boolean	Indica se o médico está ativo (true) ou inativo (false).

Detalhes dos Métodos

`cadastrar`

Este método recebe um objeto `DadosCadastroMedico` no corpo da requisição, cria uma nova instância de `Medico` com base nesses dados, e salva o médico no banco de dados através do `MedicoRepository`. Ele retorna um `ResponseEntity` com status `201 Created` e um cabeçalho `Location` contendo a URI do novo médico criado. O corpo da resposta contém um objeto `DadosDetalhamentoMedico` com as informações detalhadas do médico criado.

`listar`

Este método retorna uma lista paginada de médicos ativos. Ele utiliza o `MedicoRepository` para buscar todos os médicos com o atributo `ativo` igual a `true`. A paginação é configurada através do objeto `Pageable`, que pode ser customizado através dos parâmetros da requisição (ex: `size`, `page`, `sort`). O resultado é mapeado para uma lista de objetos `DadosListagemMedico` e retornado em um `ResponseEntity` com status `200 OK`.

`atualizar`

Este método recebe um objeto `DadosAtualizacaoMedico` no corpo da requisição e atualiza as informações de um médico existente. Ele busca o médico pelo ID através do `MedicoRepository`, atualiza seus atributos com base nos dados recebidos, e retorna um `ResponseEntity` com status `200 OK` e um objeto `DadosDetalhamentoMedico` com as informações atualizadas do médico.

`deletar`

Este método recebe o ID de um médico como parâmetro na URL e marca o médico como inativo no banco de dados. Ele busca o médico pelo ID através do `MedicoRepository` e chama o método `excluir()` para atualizar o atributo `ativo` para `false`. Retorna um `ResponseEntity` com status `204 No Content`.

`detalhar`

Este método recebe o ID de um médico como parâmetro na URL e retorna as informações detalhadas do médico. Ele busca o médico pelo ID através do `MedicoRepository` e retorna um `ResponseEntity` com status `200 OK` e um objeto `DadosDetalhamentoMedico` com as informações detalhadas do médico.

Considerações Adicionais

- **Validação:** O uso da anotação `@Valid` garante que os dados recebidos nas requisições sejam validados de acordo com as restrições definidas nas classes de DTO (Data Transfer Object).
- **Transacionalidade:** A anotação `@Transactional` garante que as operações de banco de dados sejam executadas dentro de uma transação, garantindo a consistência dos dados.
- **Segurança:** A API é protegida por autenticação via token JWT, garantindo que apenas usuários autorizados possam acessar os endpoints.

Este documento fornece uma visão geral do `MedicoController` e seus endpoints. Ele deve ser usado como referência para entender o funcionamento do componente e como interagir com ele.