Documentação Técnica Integrada: Overview do Sistema

Esta documentação consolida a descrição de dois componentes distintos: um sistema de log em PHP e um controlador de médicos (`MedicoController`) em Java (Spring Boot). O objetivo é fornecer uma visão geral clara e concisa do funcionamento de cada sistema, suas dependências, funcionalidades e como eles se encaixam no contexto geral do projeto (embora não estejam diretamente relacionados).

Componentes Principais:

1. **Sistema de Log (PHP):** Responsável por registrar as alterações realizadas no banco de dados, incluindo informações sobre o usuário que realizou a modificação, a tabela afetada e a ação executada. Ele utiliza funções PHP (`registra_modificacoes` e `registrar_log`), dependendo de um arquivo de conexão ao banco de dados (`db_conn.php`) e assume a existência de tabelas específicas no banco de dados (`access_tokens`, `log_reference` e `system_logs`).
2. **`MedicoController` (Java/Spring Boot):** Um componente REST API que gerencia operações relacionadas a médicos, como cadastro, listagem, atualização, detalhamento e exclusão (lógica). Utiliza Spring Boot, Spring Data JPA, DTOs, Bean Validation e Swagger para documentação. É protegido por autenticação via Bearer Token.

Relação entre os Componentes:

Embora descritos no mesmo documento, esses componentes *não compartilham dependências diretas*. O sistema de log é um módulo PHP autônomo que registra informações de auditoria. O `MedicoController` é uma API Java que gerencia dados de médicos. É possível que o `MedicoController` *use* o sistema de log para registrar ações como a criação, atualização ou exclusão de médicos, mas isso não é explicitamente mostrado no código fornecido. A implementação da integração dependeria de como a aplicação Java (via Spring Boot) interage com o sistema de log em PHP, possivelmente através de chamadas a APIs ou escrita em um banco de dados compartilhado.

Público-Alvo:

Esta documentação é destinada a desenvolvedores, arquitetos de software e qualquer pessoa envolvida no projeto que precise entender como os diferentes componentes funcionam e como eles se encaixam na arquitetura geral do sistema.

Visualização da Estrutura de Pastas e Arquivos (Simulação)

Dado que a documentação não fornece explicitamente a estrutura de diretórios, esta seção apresenta uma possível organização, baseada nas informações disponíveis.

```
├── php_modules/          # (Possível diretório para módulos PHP)
│   └── system_log/      # Diretório específico para o sistema de log
│       ├── db_conn.php   # Arquivo de conexão com o banco de dados
│       └── log.php       # Arquivo contendo as funções registra_modificacoes e registrar_log
│
├── java_backend/         # (Possível diretório para o backend Java)
│   ├── src/main/java/med/voll/api/  # Estrutura de pacotes Java
│   │   ├── controller/
│   │   │   └── MedicoController.java  # Controlador REST para médicos
│   │   ├── domain/
│   │   │   ├── medico/
│   │   │   │   ├── Medico.java         # Entidade Medico
│   │   │   │   ├── MedicoRepository.java # Repositório JPA para Medico
│   │   │   │   ├── DadosCadastroMedico.java  # DTO para cadastro de medico
│   │   │   │   ├── DadosListagemMedico.java  # DTO para listagem de medico
│   │   │   │   ├── DadosAtualizacaoMedico.java # DTO para atualização de medico
│   │   │   │   └── DadosDetalhamentoMedico.java # DTO para detalhamento de medico
│   │   │   └── ...
│   │   └── ...
│   └── pom.xml           # Arquivo de configuração do Maven (dependências, build, etc.)
│
└── docs/               # (Possível diretório para documentação)
    └── README.md         # Documento principal (onde esta informação pode residir)
```

Explicação:

- **`php_modules/`:** Uma possível pasta para conter módulos PHP. Dentro dela, `system_log/` guarda os arquivos PHP relacionados ao sistema de log.
- **`java_backend/`:** Uma possível pasta para o backend Java. A estrutura de diretórios segue as convenções do Spring Boot, com pacotes organizando o código Java. O `pom.xml` é o arquivo de configuração do Maven, gerenciando as dependências e o build da aplicação Java.
- **`docs/`:** Uma pasta para a documentação geral do projeto.

Observações:

- Esta é uma *estrutura de diretórios hipotética*. A estrutura real pode variar dependendo das decisões de organização do projeto.
- A documentação não detalha a estrutura completa do projeto, apenas os arquivos relevantes para os sistemas de log e o `MedicoController`.
- A relação entre os sistemas (se houver) não está visível na estrutura de diretórios, pois eles parecem ser módulos independentes.

Este documento visa fornecer uma compreensão inicial do projeto, seus componentes e sua possível organização. Para uma visão mais detalhada, é fundamental examinar o código-fonte, os arquivos de configuração e a documentação específica de cada componente.

Documentação Técnica: Sistema de Log

Este documento descreve o funcionamento do sistema de log implementado no projeto, detalhando as funções `registra_modificacoes` e `registrar_log`. O sistema permite rastrear alterações em tabelas do banco de dados, registrando informações sobre o usuário que realizou a modificação, a tabela afetada e a ação executada.

Visão Geral

O sistema de log é composto por duas funções principais:

1. **`registra_modificacoes`**: Recebe um array de modificações, um token de acesso, o ID da entidade modificada e o nome da tabela. Itera sobre o array de modificações e chama a função `registrar_log` para cada uma delas.
2. **`registrar_log`**: Recebe um token de acesso, o ID da entidade modificada, o nome da tabela e a descrição da ação realizada. Valida o token, registra a referência à modificação e, finalmente, registra o log da ação.

Dependências

O sistema de log depende do arquivo `db_conn.php`, que deve conter a configuração e a conexão com o banco de dados. Assume-se que o banco de dados seja compatível com PDO e que possua as tabelas `access_tokens`, `log_reference` e `system_logs`.

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;
}
?>
```

Funções Detalhadas

`registra_modificacoes`

Esta função é responsável por iterar sobre uma lista de modificações e registrar cada uma delas no sistema de log.

Parâmetro	Tipo	Descrição
`$modificacoes`	`array`	Um array de strings, onde cada string representa uma modificação realizada. Exemplo: `["Campo X alterado de 'A' para 'B'", "Status alterado para 'Ativo'"]`.
`$token`	`string`	O token de acesso do usuário que realizou as modificações. Este token é usado para identificar o usuário no banco de dados.
`$id`	`int`	O ID da entidade (registro) que foi modificada na tabela.
`$table`	`string`	O nome da tabela onde a entidade foi modificada.

Retorno:

- `true` se todas as modificações forem registradas com sucesso.
- `false` se o array de modificações estiver vazio.

Exemplo de Uso:

```php
$modificacoes = ["Nome alterado de 'Produto A' para 'Produto B'", "Preço alterado de 10.00 para 12.00"];
$token = "seu_token_de_acesso";
$id_produto = 123;
$tabela = "produtos";

$resultado = registra_modificacoes($modificacoes, $token, $id_produto, $tabela);

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

`registrar_log`

Esta função é responsável por registrar uma única ação no sistema de log. Ela valida o token de acesso, cria uma referência para a modificação na tabela `log_reference` e, finalmente, registra a ação na tabela `system_logs`.

Parâmetro	Tipo	Descrição
`$token`	`string`	O token de acesso do usuário que realizou a modificação. Usado para identificar o usuário.
`$id`	`int`	O ID da entidade (registro) que foi modificada na tabela.
`$table`	`string`	O nome da tabela onde a entidade foi modificada.
`$action`	`string`	Uma descrição da ação realizada. Exemplo: `"Usuário deletado do sistema"`, `"Senha alterada"`.

Retorno:

- `true` se a ação for registrada com sucesso.
- `false` se o token de acesso for inválido.

Lógica:

1. **Validação do Token:** A função primeiro consulta a tabela `access_tokens` para verificar se o token fornecido é válido e para obter o nome de usuário associado ao token. Se o token não for encontrado, a função retorna `false`.
2. **Registro da Referência:** A função insere um registro na tabela `log_reference`, contendo o ID da entidade modificada (`reference_id`) e o nome da tabela (`table_name`). O ID gerado por esta inserção é usado como chave estrangeira na tabela `system_logs`.
3. **Registro do Log:** A função insere um registro na tabela `system_logs`, contendo o nome de usuário, o ID da referência ao log (`log_reference_id`) e a descrição da ação (`action`).

Exemplo de Uso:

```
$token = "seu_token_de_acesso";
$id_usuario = 456;
$tabela = "usuarios";
$acao = "Usuário editado: email alterado de 'antigo@email.com' para 'novo@email.com'";

$resultado = registrar_log($token, $id_usuario, $tabela, $acao);

if ($resultado) {
    echo "Log registrado com sucesso!";
} else {
    echo "Falha ao registrar o log.";
}
```

Estrutura do Banco de Dados

Para o correto funcionamento do sistema de log, as seguintes tabelas devem estar presentes no banco de dados:

- **`access_tokens`**: Armazena os tokens de acesso e os nomes de usuário associados.
    - `access_token` (VARCHAR, PRIMARY KEY)
    - `username` (VARCHAR)
- **`log_reference`**: Armazena a referência às entidades modificadas.
    - `id` (SERIAL PRIMARY KEY)
    - `reference_id` (INT)
    - `table_name` (VARCHAR)
- **`system_logs`**: Armazena os logs das ações realizadas.
    - `id` (SERIAL PRIMARY KEY)
    - `username` (VARCHAR)
    - `log_reference_id` (INT, FOREIGN KEY referencing `log_reference.id`)
    - `action` (TEXT)

Considerações de Segurança

- É crucial garantir a segurança dos tokens de acesso. Os tokens devem ser armazenados e transmitidos de forma segura.
- As descrições das ações (`action`) devem ser cuidadosamente elaboradas para evitar a inclusão de informações sensíveis.
- O acesso às tabelas de log deve ser restrito a usuários autorizados.

Possíveis Melhorias

- Implementar um sistema de paginação para a visualização dos logs.
- Adicionar filtros de pesquisa para facilitar a busca por logs específicos.
- Implementar um sistema de alertas para notificar administradores sobre eventos críticos.
- Considerar a utilização de um sistema de log mais robusto, como o ELK Stack (Elasticsearch, Logstash, Kibana), para ambientes de produção com alto volume de logs.

Documentação Técnica: MedicoController

Este documento descreve o funcionamento do `MedicoController`, responsável por gerenciar as operações relacionadas a médicos na API. O controller expõe endpoints para cadastrar, listar, atualizar, detalhar e excluir informações de médicos.

Visão Geral

O `MedicoController` utiliza as seguintes tecnologias e padrões:

- **Spring Boot:** Framework para construção de aplicações Java.
- **Spring Data JPA:** Facilita o acesso e a persistência de dados no banco de dados.

- **REST API:** Arquitetura para comunicação entre o cliente e o servidor.
- **DTOs (Data Transfer Objects):** Objetos para transportar dados entre as camadas da aplicação.
- **Bean Validation (Jakarta Validation):** Para validar os dados de entrada.
- **Swagger (OpenAPI):** Para documentação da API.

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 seguir, uma descrição detalhada de cada endpoint do `MedicoController`.

1. Cadastrar Médico (`POST /medicos`)

Este endpoint permite cadastrar um novo médico no sistema.

- **Método:** `POST`
- **URL:** `/medicos`
- **Request Body:** `DadosCadastroMedico` (JSON)
- **Response:**
    - `201 Created`: Médico cadastrado com sucesso. Retorna um `DadosDetalhamentoMedico` no corpo da resposta e o header `Location` com a URL do

novo recurso.
    - `400 Bad Request`: Erro de validação nos dados de entrada.

Request Body (`DadosCadastroMedico`)

Atributo	Tipo	Descrição	Validação
nome	String	Nome completo do médico.	Obrigatório, não pode ser vazio.
email	String	Endereço de e-mail do médico.	Obrigatório, deve ser um e-mail válido.
crm	String	CRM do médico.	Obrigatório, deve seguir um padrão específico.
especialidade	Enum	Especialidade do médico.	Obrigatório, deve ser um valor válido do Enum `Especialidade`.
endereco	Objeto	Endereço do médico (ver tabela abaixo).	Obrigatório, todos os campos devem ser válidos.

Request Body (`Endereco`)

Atributo	Tipo	Descrição	Validação
logradouro	String	Logradouro do endereço.	Obrigatório, não pode ser vazio.
bairro	String	Bairro do endereço.	Obrigatório, não pode ser vazio.
cep	String	CEP do endereço.	Obrigatório, deve seguir um padrão específico.
cidade	String	Cidade do endereço.	Obrigatório, não pode ser vazio.
uf	String	Unidade Federativa do endereço.	Obrigatório, deve ser uma UF válida.
numero	String	Número do endereço.	Obrigatório, não pode ser vazio.
complemento	String	Complemento do endereço (opcional).	Opcional.

Response Body (`DadosDetalhamentoMedico`)

Atributo	Tipo	Descrição
id	Long	ID do médico.
nome	String	Nome completo do médico.
email	String	Endereço de e-mail do médico.
crm	String	CRM do médico.
especialidade	Enum	Especialidade do médico.
endereco	Objeto	Endereço do médico.

Exemplo de Requisição (JSON):

```
{
  "nome": "Dr. João Silva",
  "email": "joao.silva@voll.med",
  "crm": "123456",
  "especialidade": "CARDIOLOGIA",
  "endereco": {
    "logradouro": "Rua das Flores",
    "bairro": "Jardim",
    "cep": "12345678",
    "cidade": "São Paulo",
    "uf": "SP",
    "numero": "123",
    "complemento": "Apto 101"
  }
}
```

2. Listar Médicos (`GET /medicos`)

Este endpoint permite listar os médicos cadastrados no sistema, com suporte a paginação e ordenação.

- **Método:** `GET`
- **URL:** `/medicos`
- **Query Parameters:**
    - `page`: Número da página (padrão: 0).
    - `size`: Tamanho da página (padrão: 10).
    - `sort`: Critério de ordenação (padrão: "nome").
    - `direction`: Direção da ordenação (padrão: ASC).
- **Response:**
    - `200 OK`: Lista de médicos paginada. Retorna um `Page<DadosListagemMedico>` no corpo da resposta.

Response Body (`DadosListagemMedico`)

Atributo	Tipo	Descrição
id	Long	ID do médico.
nome	String	Nome completo do médico.
email	String	Endereço de e-mail do médico.
crm	String	CRM do médico.
especialidade	Enum	Especialidade do médico.

Exemplo de Requisição:

```
/medicos?page=0&size=20&sort=nome,asc
```

3. Atualizar Médico (`PUT /medicos`)

Este endpoint permite atualizar as informações de um médico existente.

- **Método:** `PUT`
- **URL:** `/medicos`

- **Request Body:** `DadosAtualizacaoMedico` (JSON)
- **Response:**
    - `200 OK`: Médico atualizado com sucesso. Retorna um `DadosDetalhamentoMedico` no corpo da resposta.
    - `400 Bad Request`: Erro de validação nos dados de entrada.
    - `404 Not Found`: Médico não encontrado.

Request Body (`DadosAtualizacaoMedico`)

Atributo	Tipo	Descrição	Validação
id	Long	ID do médico a ser atualizado.	Obrigatório.
nome	String	Nome completo do médico (opcional).	Opcional.
email	String	Endereço de e-mail do médico (opcional).	Opcional, deve ser um e-mail válido.
crm	String	CRM do médico (opcional).	Opcional, deve seguir um padrão específico.
endereco	Objeto	Endereço do médico (opcional, ver tabela abaixo).	Opcional, todos os campos devem ser válidos.

Exemplo de Requisição (JSON):

```
{
  "id": 1,
  "nome": "Dr. João Silva Atualizado",
  "endereco": {
    "logradouro": "Nova Rua das Flores",
    "numero": "456"
  }
}
```

4. Excluir Médico (`DELETE /medicos/{id}`)

Este endpoint permite desativar um médico existente. A exclusão é lógica, ou seja, o médico não é removido do banco de dados, mas seu status é alterado para inativo.

- **Método:** `DELETE`
- **URL:** `/medicos/{id}`
- **Path Variable:**
    - `id`: ID do médico a ser excluído.
- **Response:**
    - `204 No Content`: Médico excluído com sucesso.
    - `404 Not Found`: Médico não encontrado.

Exemplo de Requisição:

```
/medicos/1
```

5. Detalhar Médico (`GET /medicos/{id}`)

Este endpoint permite obter os detalhes de um médico específico.

- **Método:** `GET`
- **URL:** `/medicos/{id}`
- **Path Variable:**
    - `id`: ID do médico a ser detalhado.
- **Response:**
    - `200 OK`: Detalhes do médico. Retorna um `DadosDetalhamentoMedico` no corpo da resposta.
    - `404 Not Found`: Médico não encontrado.

Exemplo de Requisição:

```
/medicos/1
```

Autenticação

Todos os endpoints do `MedicoController` requerem autenticação via Bearer Token. A anotação `@SecurityRequirement(name = "bearer-key")` indica essa exigência.

Tratamento de Exceções

O `MedicoController` não implementa tratamento de exceções diretamente. Espera-se que um handler global capture exceções e retorne respostas de erro apropriadas.

Considerações Finais

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