



UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

Relatório - Trabalho Final

Universidade Federal do Ceará - Campus Quixadá

Disciplina: Sistemas Distribuídos

Autor: Robson Mendes

Professor: Rafael Braga

1. Introdução

Este relatório detalha o desenvolvimento do Trabalho 3 da disciplina de Sistemas Distribuídos, cujo objetivo principal é a reimplementação de um serviço remoto, originalmente desenvolvido com Java RMI, utilizando tecnologias modernas de Web Services (WS) ou Application Programming Interface (API). O projeto consiste em um sistema de gerenciamento de Recursos Humanos (RH) para um hospital, permitindo operações de CRUD (Create, Read, Update, Delete) para diferentes tipos de funcionários.

Seguindo os requisitos da disciplina, a nova arquitetura abandona o acoplamento da tecnologia RMI em favor de uma API RESTful, que se comunica com clientes através do protocolo HTTP. Adicionalmente, foram desenvolvidos dois clientes em linguagens distintas da utilizada no servidor (Python e JavaScript), demonstrando a interoperabilidade e a flexibilidade da solução baseada em API.

2. Implementação Clientes-Servidor

A solução foi projetada seguindo o padrão cliente-servidor, com uma clara separação de responsabilidades entre o back-end (servidor) e os front-ends (clientes).

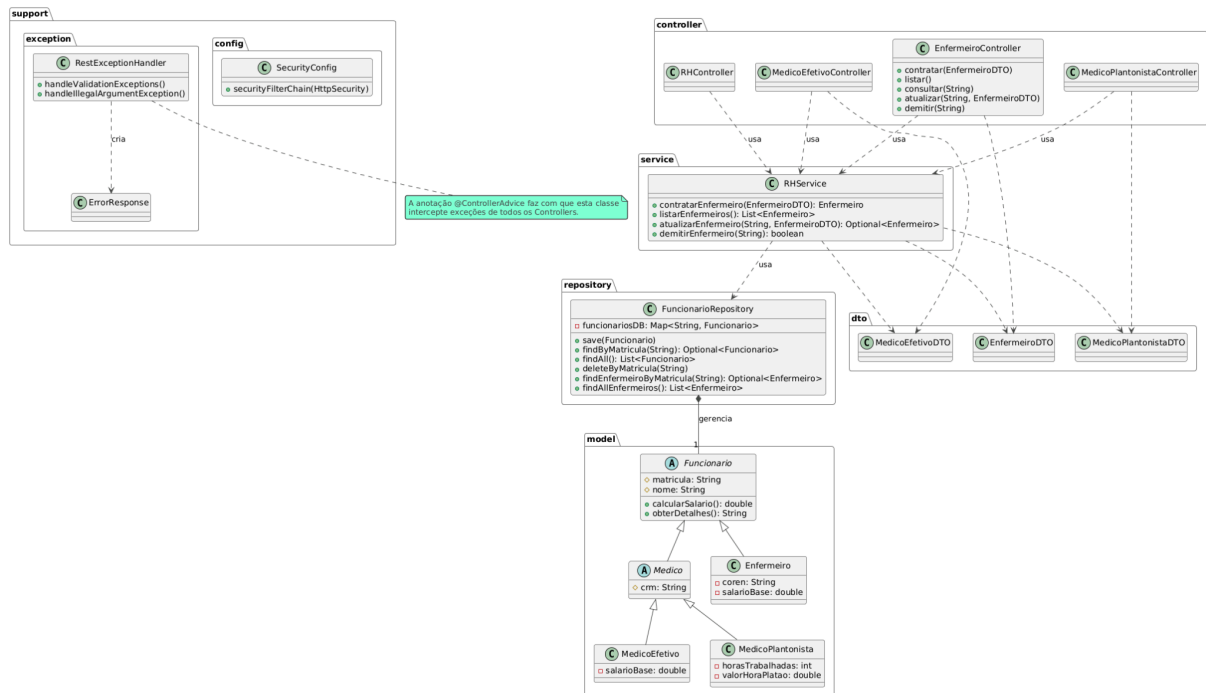
- **Servidor (Back-end):** Desenvolvido em Java com o framework Spring Boot. Ele expõe uma API RESTful que serve como ponto central para toda a lógica de negócio e persistência de dados. A comunicação é feita via HTTP, e os dados são trafegados no formato JSON.
- **Clientes (Front-end):**
 1. **Cliente de Terminal:** Uma aplicação de console desenvolvida em Python, que permite ao usuário interagir com todas as funcionalidades da API através de um menu de texto.
 2. **Cliente Web:** Uma Single-Page Application (SPA) desenvolvida com HTML, CSS e JavaScript, utilizando o framework Bootstrap para uma interface de usuário que roda diretamente no navegador.

3. Arquitetura do Servidor (API REST)

O servidor foi estruturado seguindo os princípios de design da arquitetura em camadas, para garantir a separação de responsabilidades, manutenibilidade e testabilidade do código.

Diagrama de Classes

O diagrama abaixo ilustra a relação entre as principais classes do servidor:



3.1. Camada de Modelo (model)

Esta camada representa as entidades centrais do sistema. A classe abstrata *Funcionario* serve como base para os tipos específicos *Enfermeiro*, *MedicoEfetivo* e *MedicoPlantonista*. A anotação *@JsonTypeInfo* foi utilizada na classe base para facilitar a serialização e desserialização polimórfica dos dados em JSON.

3.2. Camada de Repositório (repository)

Esta camada abstrai a lógica de persistência dos dados. Foram implementadas duas estratégias de persistência distintas para demonstrar a flexibilidade da arquitetura:

- **FuncionarioRepositoryCsv:** Responsável por persistir os dados da entidade *Enfermeiro* em um arquivo *enfermeiros_data.csv*. Ele carrega os dados para um cache em memória na inicialização e reescreve o arquivo a cada operação de alteração.
- **FuncionarioRepository:** Responsável por gerenciar as entidades de *Medico* (*MedicoEfetivo*, *MedicoPlantonista*) em um Map em memória, servindo como uma base de dados volátil para estas entidades.

3.3. Camada de Serviço (*service*)

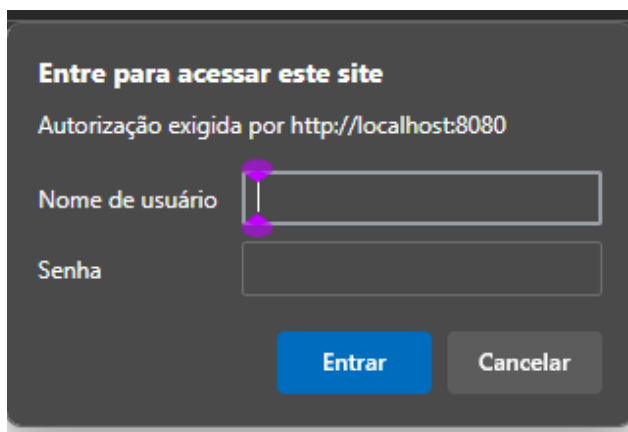
O *RHService* atua como o cérebro da aplicação, orquestrando a lógica de negócio. Ele recebe as solicitações dos controllers, aplica as regras de negócio (como a verificação de matrícula duplicada) e coordena as chamadas aos repositórios corretos, decidindo se os dados de um funcionário devem ser persistidos em memória ou em CSV.

3.4. Camada de Controle (*controller*)

Esta camada é a porta de entrada da API. Foram criados controllers específicos para cada entidade (*EnfermeiroController*, *MedicoEfetivoController*, etc.), expondo endpoints RESTful para todas as operações CRUD (*@GetMapping*, *@PostMapping*, *@PutMapping*, *@DeleteMapping*). Os DTOs (Data Transfer Objects) são utilizados para validar os dados de entrada.

3.5. Segurança e Tratamento de Exceções

- **Segurança:** A API foi protegida com Spring Security, exigindo Autenticação Básica (Basic Auth) para todas as requisições. A configuração, centralizada em *SecurityConfig.java*, desabilita a proteção CSRF (inadequada para APIs stateless) e define um usuário em memória para acesso.



- **Exceções:** Foi implementado um tratador de exceções global (*RestExceptionHandler*) com a anotação *@ControllerAdvice*. Ele captura exceções de negócio como matrícula duplicada e de validação, retornando respostas de erro padronizadas em JSON com os códigos de status HTTP apropriados (400, 404, 500.)

4. Implementação dos Clientes

4.1. Cliente 1: Python (Aplicação de Terminal)

Foi desenvolvido um cliente de console em Python, estruturado em três arquivos para uma melhor organização.

- **api_client.py** - A Camada de Comunicação: Este arquivo contém a classe `ApiClient`, que encapsula toda a lógica de comunicação com a API REST. Sua única responsabilidade é montar e enviar requisições HTTP, utilizando a biblioteca `requests`. Um método genérico `request(method, endpoint, data)` foi implementado para evitar a repetição de código, lidando de forma centralizada com a construção da URL, a adição das credenciais de Autenticação Básica e o tratamento de erros de conexão.
- **app_logic.py** - A Camada de Lógica e Interface: Este arquivo é o cérebro da aplicação cliente. Ele contém as funções que geram a interface do usuário (como os menus de texto), coletam os dados de entrada (`coletar_dados_funcionario`) e processam as respostas recebidas do `ApiClient` (`processar_resposta`). A função `processar_resposta` interpreta os códigos de status HTTP para fornecer feedback ao usuário, seja em caso de sucesso (200...) ou de erro (400...), exibindo as mensagens de validação enviadas pelo servidor.
- **main.py** - É o inicializador da aplicação. Ele configura as constantes (URL base e credenciais), instancia o `ApiClient` e executa o loop do menu principal, delegando as ações do usuário para as funções apropriadas no `app_logic.py`.

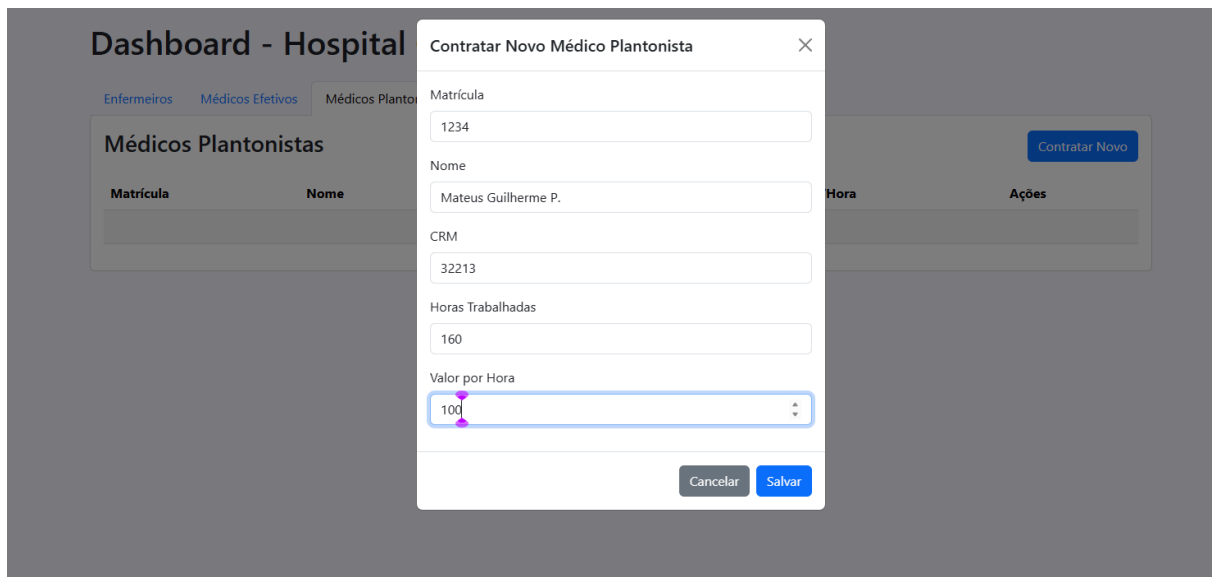
```
--- CLIENTE PYTHON - API DE RH ---
1. Gerenciar Enfermeiros
2. Gerenciar Médicos Efetivos
3. Gerenciar Médicos Plantonistas
4. Listar TODOS os funcionários
0. Sair
Escolha uma opção:
```

4.2. Cliente 2: JavaScript (Single-Page Application Web)

O segundo cliente é uma aplicação web rica e interativa, contida em um único arquivo `index.html`.

- Utiliza HTML5 para a estrutura e o framework Bootstrap 5 (via CDN) para criar uma interface responsiva e profissional com abas, tabelas e modais.

- O JavaScript (com async/await e fetch) é usado para todas as interações com a API. Ao carregar, a página solicita as credenciais do usuário, que são codificadas em Base64 e enviadas no cabeçalho Authorization de cada requisição, conforme o padrão Basic Auth. A UI é atualizada dinamicamente sem a necessidade de recarregar a página, consumindo os endpoints CRUD da API para gerenciar todos os tipos de funcionários.



5. Descrição das Rotas da API (Endpoints)

A API foi projetada para ser RESTful, utilizando os métodos HTTP para representar as operações de CRUD. Os endpoints são divididos por recurso para garantir clareza e organização. A autenticação é necessária para todos os endpoints.

5.1. Endpoints Gerais de RH

Estes endpoints são gerenciados pelo **RHController** e lidam com operações que se aplicam a todos os funcionários.

- **GET /api/rh/funcionarios**
 - Descrição: Retorna uma lista de todos os funcionários cadastrados no sistema, independentemente do tipo (Enfermeiros, Médicos Efetivos e Plantonistas).
 - Resposta de Sucesso (200 OK): Um array JSON contendo os objetos de todos os funcionários.
- **GET /api/rh/funcionarios/{matricula}**
 - Descrição: Consulta e retorna os dados de um funcionário específico, identificado pela sua matrícula.
 - Resposta de Sucesso (200 OK): Um objeto JSON com os dados do funcionário encontrado.
 - Resposta de Erro (404 Not Found): Se nenhum funcionário com a matrícula informada for encontrado.
- **DELETE /api/rh/funcionarios/{matricula}**
 - Descrição: Demite (remove) um funcionário específico do sistema.
 - Resposta de Sucesso (204 No Content): Retorna uma resposta vazia, indicando que o recurso foi removido com sucesso.
 - Resposta de Erro (404 Not Found): Se o funcionário a ser demitido não for encontrado.

5.2. Endpoints de Enfermeiros


Gerenciados pelo **EnfermeiroController**, estes endpoints lidam exclusivamente com a entidade **Enfermeiro**. Os dados são persistidos em um arquivo CSV.

- **POST /api/enfermeiros**
 - Descrição: Contrata (cria) um novo enfermeiro.
 - Corpo da Requisição: Um objeto JSON (**EnfermeiroDTO**) contendo **matricula**, **nome**, **coren** e **salarioBase**.
 - Resposta de Sucesso (201 Created): Retorna o objeto JSON completo do enfermeiro recém-criado.
 - Resposta de Erro (400 Bad Request): Se os dados enviados forem inválidos (ex: campos em branco) ou se a matrícula já existir.
- **GET /api/enfermeiros**
 - Descrição: Retorna uma lista de todos os enfermeiros cadastrados.
 - Resposta de Sucesso (200 OK): Um array JSON com os objetos de todos os enfermeiros.
- **GET /api/enfermeiros/{matricula}**
 - Descrição: Consulta um enfermeiro específico.
 - Resposta de Sucesso (200 OK): O objeto JSON do enfermeiro.

- Resposta de Erro (404 Not Found): Se a matrícula não for encontrada.
- **PUT /api/enfermeiros/{matricula}**
 - Descrição: Atualiza os dados de um enfermeiro existente.
 - Corpo da Requisição: Um objeto JSON (**EnfermeiroDTO**) com os novos dados. A matrícula na URL é usada para encontrar o recurso, e a matrícula no corpo pode ser omitida.
 - Resposta de Sucesso (200 OK): Retorna o objeto JSON completo do enfermeiro com os dados atualizados.
 - Resposta de Erro (404 Not Found): Se o enfermeiro a ser atualizado não for encontrado.
- **DELETE /api/enfermeiros/{matricula}**
 - Descrição: Demite um enfermeiro específico.
 - Resposta de Sucesso (204 No Content): Resposta vazia.
 - Resposta de Erro (404 Not Found): Se o enfermeiro a ser demitido não for encontrado.

5.3. Endpoints de Médicos Efetivos e Plantonistas

Os endpoints para Médicos Efetivos (**/api/medicos-efetivos**) e Médicos Plantonistas (**/api/medicos-plantonistas**) seguem a mesma estrutura de CRUD do **EnfermeiroController**, com a diferença de que seus dados são persistidos em memória e os DTOs correspondentes são utilizados nas operações de **POST** e **PUT**.

 Swagger
by SMARTBEAN

v3/api-docs

Explore

OpenAPI definition v0 **OAS 3.0**
v3/api-docs

Servers
http://localhost:8080 - Generated server url

medico-plantonista-controller

GET

/api/medicos-plantonistas/{matricula}

PUT

/api/medicos-plantonistas/{matricula}

DELETE

/api/medicos-plantonistas/{matricula}

GET

/api/medicos-plantonistas

POST

/api/medicos-plantonistas

medico-efetivo-controller

GET

/api/medicos-efetivos/{matricula}

PUT

/api/medicos-efetivos/{matricula}

DELETE

/api/medicos-efetivos/{matricula}

GET

/api/medicos-efetivos

POST

/api/medicos-efetivos

enfermeiro-controller

GET

/api/enfermeiros/{matricula}

PUT

/api/enfermeiros/{matricula}

DELETE

/api/enfermeiros/{matricula}

GET

/api/enfermeiros

POST

/api/enfermeiros

rh-controller

GET

/api/rh/hospital

GET

/api/rh/funcionarios

GET

/api/rh/funcionarios/{matricula}

DELETE

/api/rh/funcionarios/{matricula}

6. Conclusão

O projeto cumpriu todos os objetivos propostos, migrando uma arquitetura legada baseada em Java RMI para uma API RESTful com Java e o framework Spring Boot. A nova arquitetura do servidor é modular, organizada em camadas de Controle, Serviço e Repositório, e inclui duas estratégias de persistência de dados: uma em memória e outra baseada em arquivo CSV.