




CochitAPI - Sistema de Gestão e Controle de Serviços

Uma API REST desenvolvida em Java com Spring Boot para gestão e controle de serviços, funcionários e clientes com persistência em banco de dados.

Link do repositório: <https://github.com/Adriel-Cochito/cochitoapi>

-  Aluno: Adriel Henrique Borges Cochito
-  Desenvolvimento de aplicações Java com Spring Boot [25E3_2]
-  MIT Engenharia de Software (JAVA)

Sobre o Projeto

Este projeto faz parte da disciplina "Desenvolvimento Avançado com Spring e Microserviços" da Pós-graduação MIT em Engenharia de Software. A aplicação implementa um sistema completo de CRUD (Create, Read, Update, Delete) para gestão de entidades de negócio, seguindo as melhores práticas de desenvolvimento com Spring Framework.

Status do Projeto: ☒ **CONCLUÍDO** - Todas as 4 Features implementadas com sucesso!

Tecnologias Utilizadas

- **Java 17**
- **Spring Boot 3.5.4**
- **Spring Web**
- **Spring Data JPA**
- **Spring Boot Validation**
- **H2 Database**
- **Maven**
- **RESTful API**
- **Bean Validation**
- **Global Exception Handling**

Estrutura do Projeto

```
src/main/java/br/edu/infnet/cochitoapi/
├── controller/           # Camada de controle (REST Controllers)
├── model/
│   ├── domain/          # Entidades de domínio
│   │   └── exceptions/   # Exceções customizadas e handlers
│   ├── repository/       # Interfaces de repositório JPA
│   └── service/          # Camada de serviço (regras de negócio)
├── loader/              # Classes para carga inicial de dados
└── CochitoapiApplication.java
```

Arquitetura

O projeto segue o padrão MVC (Model-View-Controller) com separação clara de responsabilidades:

- **Controller:** Responsável por receber requisições HTTP e retornar respostas
- **Service:** Contém a lógica de negócio e validações
- **Repository:** Camada de acesso a dados com Spring Data JPA
- **Model:** Define as entidades de domínio e suas relações

Modelo de Domínio

```
Pessoa (Classe Abstrata - @MappedSuperclass)
├── Funcionario (@Entity)
└── Cliente (@Entity)

Endereco (@Entity - Classe de Associação)
Servico (@Entity - Entidade Independente)
```

Entidades

Pessoa (Abstrata - @MappedSuperclass)

- **id:** Integer (PK, auto-increment)
- **nome:** String (validado: 3-50 caracteres)
- **email:** String (validado: formato email válido)
- **cpf:** String (validado: formato XXX.XXX.XXX-XX)
- **telefone:** String (validado: formato (XX) XXXXX-XXXX)

Funcionario (@Entity extends Pessoa)

- **matricula:** int (obrigatório, mínimo: 1)
- **salario:** double (mínimo: 0)
- **ativo:** boolean
- **endereco:** Endereco (@ManyToOne, cascade=ALL)

Cliente (@Entity extends Pessoa)

- **fidelidade:** String (validado: 3-20 caracteres)

Servico (@Entity)

- **id:** Integer (PK, auto-increment)
- **titulo:** String (validado: 3-100 caracteres)
- **preco:** double (mínimo: 0)
- **descricao:** String (validado: 10-500 caracteres)

Endereco (@Entity)

- **id:** Integer (PK, auto-increment)
- **cep:** String (validado: formato XXXXX-XXX)
- **logradouro:** String (validado: 3-100 caracteres)

- **complemento**: String
- **unidade**: String
- **bairro**: String (validado: 3-50 caracteres)
- **localidade**: String (validado: 3-50 caracteres)
- **uf**: String (validado: 2 caracteres)
- **estado**: String (validado: 3-50 caracteres)

🔧 Funcionalidades Implementadas

☑ Feature 1: Configuração Essencial (100% Concluída)

- ☑ Configuração inicial do projeto Spring Boot
- ☑ Modelagem de entidade principal (Funcionario)
- ☑ Implementação de operações CRUD básicas em memória
- ☑ API REST simples com carregamento inicial de dados
- ☑ Criação da primeira classe Controller
- ☑ Implementação do primeiro Loader
- ☑ Integração com Spring Boot e Maven

☑ Feature 2: Expansão do Modelo de Domínio (100% Concluída)

- ☑ **Estrutura do modelo de domínio expandido**
 - ☑ Classe Mãe: Pessoa (abstrata) com 4+ atributos
 - ☑ Classe Filha 1: Funcionario (extends Pessoa) com atributos específicos
 - ☑ Classe Filha 2: Cliente (extends Pessoa) com atributos específicos
 - ☑ Classe de Associação: Endereco (ManyToOne com Funcionario)
- ☑ **Tratamento de exceções customizadas**
 - ☑ RecursoInvalidoException para regras de negócio
 - ☑ RecursoNaoEncontradoException para recursos inexistentes
 - ☑ GlobalExceptionHandler para tratamento centralizado
- ☑ **Interface CrudService<T,ID> atualizada**
 - ☑ Contrato completo: incluir, alterar, buscarPorId, listarTodos, excluir
- ☑ **Gerenciamento de dados iniciais (Loaders)**
 - ☑ FuncionarioLoader: carrega funcionários e endereços
 - ☑ ClienteLoader: carrega clientes
 - ☑ ServicoLoader: carrega serviços
- ☑ **Camada de serviço completa**
 - ☑ FuncionarioService: CRUD + inativar()
 - ☑ ClienteService: CRUD + atualizarFidelidade()
 - ☑ ServicoService: CRUD completo
- ☑ **Camada de controle (API REST)**
 - ☑ FuncionarioController: GET, POST, PUT, PATCH, DELETE
 - ☑ ClienteController: GET, POST, PUT, PATCH, DELETE
 - ☑ ServicoController: GET, POST, PUT, DELETE
- ☑ **Testes com Postman**
 - ☑ Coleções preparadas para todos os endpoints
 - ☑ RequestBody e PathVariable implementados

- ☒ Validação de todos os verbos HTTP

☒ Feature 3: Persistência com Banco de Dados (100% Concluída)

- ☒ **Dependências essenciais (pom.xml)**
 - ☒ Spring Boot Starter Data JPA
 - ☒ H2 Database
 - ☒ Spring Boot Validation
- ☒ **Configuração do banco de dados (application.properties)**
 - ☒ Configuração H2 completa (jdbc:h2:~/databaseCochito)
 - ☒ Console H2 habilitado (/h2-console)
 - ☒ Configuração JPA/Hibernate (ddl-auto=create, show-sql=true)
- ☒ **Mapeamento das entidades com JPA**
 - ☒ @Entity em Funcionario, Cliente e Servico
 - ☒ @MappedSuperclass em Pessoa (estratégia de herança)
 - ☒ @Id e @GeneratedValue para chaves primárias
 - ☒ Relacionamento @ManyToOne entre Funcionario e Endereco
 - ☒ Cascade ALL para persistência automática de endereços
- ☒ **Bean Validation implementado**
 - ☒ @NotNull, @NotBlank, @Email em Pessoa
 - ☒ @Size para validação de tamanho de strings
 - ☒ @Pattern para validação de CPF e telefone
 - ☒ @Min para validação de salário e preço mínimos
 - ☒ @Valid para validação em cascata
- ☒ **Criação de repositórios com Spring Data JPA**
 - ☒ FuncionarioRepository extends JpaRepository<Funcionario, Integer>
 - ☒ ClienteRepository extends JpaRepository<Cliente, Integer>
 - ☒ ServicoRepository extends JpaRepository<Servico, Integer>
- ☒ **Atualização da camada de serviço**
 - ☒ FuncionarioService migrado para JpaRepository
 - ☒ ClienteService migrado para JpaRepository
 - ☒ ServicoService migrado para JpaRepository
 - ☒ Remoção completa de Map/ConcurrentHashMap
- ☒ **Refinamento da API REST com ResponseEntity**
 - ☒ Todos os Controllers: Status HTTP apropriados (201, 200, 204, 400, 404)
 - ☒ Tratamento adequado de códigos de resposta
- ☒ **Tratamento de exceções refinado**
 - ☒ GlobalExceptionHandler com ResponseEntity
 - ☒ Tratamento de MethodArgumentNotValidException
 - ☒ ErrorResponse e ValidationErrorResponse estruturados
 - ☒ Timestamps e URIs de erro incluídos

☒ Feature 4: Robustez, Validação Avançada e Relacionamentos Complexos (100% Concluída)

- ☒ **Bean Validation Avançado**
 - ☒ Validações sofisticadas implementadas (@Min, @Max, @Pattern, @Email, @Size)
 - ☒ Validações em todas as entidades (Pessoa, Funcionario, Cliente, Servico, Endereco)

- ☒ Feedback estruturado ao cliente via `GlobalExceptionHandler`
- ☒ `ValidationErrorResponse` com detalhes dos campos que falharam
- ☒ **Tratamento Global de Exceções Robusto**
 - ☒ `@ControllerAdvice` e `@ExceptionHandler` implementados
 - ☒ Mapeamento completo de exceções:
 - ☒ `IllegalArgumentException` → 404 `NOT_FOUND`
 - ☒ `RecursoNaoEncontradoException` → 404 `RESOURCE_NOT_FOUND`
 - ☒ `RecursoInvalidoException` → 400 `INVALID_DATA`
 - ☒ `MethodArgumentNotValidException` → 400 `VALIDATION_ERROR`
 - ☒ Estrutura de erro padronizada (JSON com timestamp, status, error, message, path)
 - ☒ Classes de resposta especializadas: `ErrorResponse` e `ValidationErrorResponse`
- ☒ **Implementação de Relacionamento One-to-Many**
 - ☒ Relacionamento `@ManyToOne` entre `Funcionario` e `Endereco` implementado
 - ☒ Cascade ALL para operações em cascata
 - ☒ Validação `@Valid` para objetos relacionados
- ☒ **População de Dados via Loaders**
 - ☒ Arquivos texto dedicados: `funcionario.txt`, `cliente.txt`, `servico.txt`
 - ☒ Loaders específicos: `FuncionarioLoader`, `ClienteLoader`, `ServicoLoader`
 - ☒ Associação dinâmica entre `Funcionario` e `Endereco`
 - ☒ Ordem correta de execução dos loaders
- ☒ **Uso Completo de Repositórios JPA**
 - ☒ Spring Data JPA em todas as entidades
 - ☒ Métodos de consulta automáticos (`findById`, `findAll`, `save`, `delete`)
 - ☒ Demonstração de funcionalidades JPA em serviços e controladores

Banco de Dados

Configuração H2

```
# application.properties
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/databaseCochito
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Console H2 disponível em: <http://localhost:8080/h2-console>

Estrutura das Tabelas

PESSOA (Superclasse - `@MappedSuperclass`)

- Atributos herdados pelas tabelas filhas

FUNCIONARIO

```
CREATE TABLE funcionario (  
  id INTEGER AUTO_INCREMENT PRIMARY KEY,  
  nome VARCHAR(50) NOT NULL,  
  email VARCHAR(255) NOT NULL,  
  cpf VARCHAR(14) NOT NULL,  
  telefone VARCHAR(15) NOT NULL,  
  matricula INTEGER NOT NULL,  
  salario DOUBLE,  
  ativo BOOLEAN,  
  endereco_id INTEGER,  
  FOREIGN KEY (endereco_id) REFERENCES endereco(id)  
);
```

CLIENTE

```
CREATE TABLE cliente (  
  id INTEGER AUTO_INCREMENT PRIMARY KEY,  
  nome VARCHAR(50) NOT NULL,  
  email VARCHAR(255) NOT NULL,  
  cpf VARCHAR(14) NOT NULL,  
  telefone VARCHAR(15) NOT NULL,  
  fidelidade VARCHAR(20) NOT NULL  
);
```

SERVICO

```
CREATE TABLE servico (  
  id INTEGER AUTO_INCREMENT PRIMARY KEY,  
  titulo VARCHAR(100) NOT NULL,  
  preco DOUBLE NOT NULL,  
  descricao VARCHAR(500) NOT NULL  
);
```

ENDereco

```
CREATE TABLE endereco (  
  id INTEGER AUTO_INCREMENT PRIMARY KEY,  
  cep VARCHAR(9),  
  logradouro VARCHAR(255),  
  complemento VARCHAR(255),  
  unidade VARCHAR(50),  
  bairro VARCHAR(100),  
  localidade VARCHAR(100),  
  uf VARCHAR(2),
```

```
estado VARCHAR(50)
);
```

🔗 Endpoints da API

Funcionários

- `GET /api/funcionarios` - Lista todos os funcionários
- `GET /api/funcionarios/{id}` - Busca funcionário por ID
- `POST /api/funcionarios` - Cria novo funcionário (201 CREATED)
- `PUT /api/funcionarios/{id}` - Altera funcionário completo (200 OK)
- `PATCH /api/funcionarios/{id}/inativar` - Inativa funcionário (200 OK)
- `DELETE /api/funcionarios/{id}` - Remove funcionário (204 NO CONTENT)

Clientes

- `GET /api/clientes` - Lista todos os clientes
- `GET /api/clientes/{id}` - Busca cliente por ID
- `POST /api/clientes` - Cria novo cliente (201 CREATED)
- `PUT /api/clientes/{id}` - Altera cliente completo (200 OK)
- `PATCH /api/clientes/{id}/fidelidade` - Atualiza nível de fidelidade (200 OK)
- `DELETE /api/clientes/{id}` - Remove cliente (204 NO CONTENT)

Serviços

- `GET /api/servicos` - Lista todos os serviços
- `GET /api/servicos/{id}` - Busca serviço por ID
- `POST /api/servicos` - Cria novo serviço (201 CREATED)
- `PUT /api/servicos/{id}` - Altera serviço completo (200 OK)
- `DELETE /api/servicos/{id}` - Remove serviço (204 NO CONTENT)

📖 Como Executar

Pré-requisitos

- Java 17 ou superior
- Maven 3.6+

Passos para execução

1. Clone o repositório

```
git clone [URL_DO_REPOSITORIO]
cd cochitoapi
```

2. Compile o projeto

```
mvn clean install
```

3. Execute a aplicação

```
mvn spring-boot:run
```

4. Acesse a API

```
http://localhost:8080/api
```

5. Acesse o Console H2 (Para visualizar o banco)

```
http://localhost:8080/h2-console
```

Arquivos de Dados

O projeto utiliza arquivos texto para carga inicial dos dados:

- **funcionario.txt** - Dados dos funcionários e endereços
- **cliente.txt** - Dados dos clientes
- **servico.txt** - Dados dos serviços

Formato dos arquivos:

funcionario.txt:

```
Nome;Email;CPF;Telefone;Matricula;Salario;EhAtivo;CEP;Logradouro;Complemento;Unidade;Bairro;Localidade;UF;Estado
```

cliente.txt:

```
Nome;CPF;Email;Telefone;Fidelidade
```

servico.txt:

```
Titulo;Preco;Descricao
```



Testando a API

Recomenda-se o uso do **Postman** para testar os endpoints da API.

Exemplo de teste POST (Funcionário com Validação):

POST `http://localhost:8080/api/funcionarios`

Content-Type: `application/json`

```
{
  "nome": "João Silva",
  "cpf": "123.456.789-00",
  "email": "joao@email.com",
  "telefone": "(11) 99999-9999",
  "matricula": 12345,
  "salario": 5000.00,
  "ativo": true,
  "endereco": {
    "cep": "01234-567",
    "logradouro": "Rua das Flores",
    "bairro": "Centro",
    "localidade": "São Paulo",
    "uf": "SP",
    "estado": "São Paulo"
  }
}
```

Exemplo de resposta de erro de validação:

```
{
  "success": false,
  "status": 400,
  "code": "VALIDATION_ERROR",
  "message": "Dados inválidos fornecidos",
  "path": "/api/funcionarios",
  "timestamp": "2025-01-XX...",
  "validationErrors": [
    {
      "field": "cpf",
      "rejectedValue": "123456789",
      "message": "CPF deve estar no formato XXX.XXX.XXX-XX"
    }
  ]
}
```


Padrões e Boas Práticas Implementadas

- **Arquitetura em Camadas:** Controller, Service, Repository bem definidas
- **Injeção de Dependência:** Uso de injeção por construtor
- **Tratamento de Exceções:** GlobalExceptionHandler centralizado

- **Interface Genérica:** `CrudService<T, ID>` para padronização
- **Bean Validation:** Validações declarativas com annotations
- **JPA/Hibernate:** Mapeamento objeto-relacional automático
- **Response Entity:** Controle granular de respostas HTTP com códigos apropriados
- **Estratégia de Herança:** `@MappedSuperclass` para Pessoa
- **Relacionamentos JPA:** `@ManyToOne` com cascade configurado
- **Transacional:** `@Transactional` para operações que modificam dados
- **Validação em Cascata:** `@Valid` para objetos relacionados

Status Final das Features

Feature	Status	Entregáveis	Progresso
Feature 1	<input checked="" type="checkbox"/> Concluída	Configuração base + CRUD simples	100%
Feature 2	<input checked="" type="checkbox"/> Concluída	Modelo expandido + CRUD completo	100%
Feature 3	<input checked="" type="checkbox"/> Concluída	Persistência JPA + API refinada	100%
Feature 4	<input checked="" type="checkbox"/> Concluída	Validação avançada + Tratamento global	100%

 Projeto 100% Implementado ☒

Todas as funcionalidades foram entregues com sucesso:

☒ Arquitetura Completa

- **Persistência Real:** H2 Database com JPA/Hibernate
- **Validação Robusta:** Bean Validation em todas as entidades
- **Tratamento de Erros:** `GlobalExceptionHandler` com respostas estruturadas
- **API RESTful:** Endpoints completos com códigos HTTP apropriados

☒ Modelo de Domínio Robusto

- **Herança:** Pessoa como `@MappedSuperclass`
- **Relacionamentos:** `@ManyToOne` entre `Funcionario` e `Endereco`
- **Entidades Completas:** `Funcionario`, `Cliente`, `Servico`, `Endereco`
- **Validações:** Todas as regras de negócio implementadas

☒ Funcionalidades Avançadas

- **CRUD Completo:** Create, Read, Update, Delete para todas as entidades
- **Operações Especiais:** `inativar()`, `atualizarFidelidade()`
- **Carga de Dados:** Loaders automáticos a partir de arquivos texto
- **Console H2:** Interface para visualização dos dados

Conceitos Aplicados - Aprendizado Consolidado

Este projeto demonstra o domínio completo dos seguintes conceitos:

Fundamentos Spring Boot

- Configuração de projeto com Spring Initializr
- Injeção de Dependência e Inversão de Controle
- Componentes (@Component, @Service, @Repository, @RestController)
- ApplicationRunner para inicialização de dados

Arquitetura e Design Patterns

- Padrão MVC (Model-View-Controller)
- Separação de responsabilidades em camadas
- Interface CrudService genérica para padronização
- Tratamento centralizado de exceções

Orientação a Objetos

- Herança: classe abstrata Pessoa
- Polimorfismo: método abstrato obterTipo()
- Encapsulamento: getters/setters
- Associação: relacionamento entre classes

Persistência e Banco de Dados

- Spring Data JPA e Hibernate
- Mapeamento objeto-relacional com annotations
- Estratégias de herança (@MappedSuperclass)
- Relacionamentos (@ManyToOne, cascade)
- H2 Database em memória

Validação e Tratamento de Erros

- Bean Validation (@NotNull, @NotBlank, @Size, @Pattern, @Email, @Min)
- Exceções customizadas (RecursoInvalidoException, RecursoNaoEncontradoException)
- GlobalExceptionHandler com @ControllerAdvice
- ResponseEntity com códigos HTTP apropriados

API RESTful

- Verbos HTTP (GET, POST, PUT, PATCH, DELETE)
- @PathVariable e @RequestBody
- Códigos de status HTTP (200, 201, 204, 400, 404)
- Estrutura de respostas JSON padronizada



Conclusão do Projeto

Este projeto foi desenvolvido com sucesso seguindo metodologia ágil com entregas incrementais por features. **Todas as 4 features foram implementadas completamente**, demonstrando o domínio dos conceitos de desenvolvimento avançado com Spring Boot.

Principais conquistas:

- ☒ Arquitetura sólida e bem estruturada
- ☒ Persistência real com banco de dados
- ☒ Validações robustas e tratamento de erros
- ☒ API RESTful completa e funcional
- ☒ Aplicação das melhores práticas de desenvolvimento

Projeto desenvolvido como parte do curso de Pós-graduação MIT em Engenharia de Software - Instituto INFNET.

Status Final: 🏆 **PROJETO 100% CONCLUÍDO** - Todas as Features Implementadas com Sucesso!