Documentação do Projeto - Sistema de Gestão de Clientes

Descrição do Projeto

Este projeto é uma aplicação web desenvolvida com **Java Spring Boot** utilizando a IDE **VSCode**. O sistema permite o gerenciamento de informações sobre clientes, incluindo o cadastramento, consulta, alteração, exclusão, listagem e pesquisa. Ele também realiza a integração com a **API ViaCEP** para consultar e preencher automaticamente os dados de endereço de um cliente a partir do **CEP** informado.

A aplicação é uma **API RESTful**, que expõe endpoints para interagir com os dados de clientes e endereços. Utiliza **DTOs** (Data Transfer Objects) para separar as camadas de comunicação entre o banco de dados e o front-end, além de um **Mapper** para converter entre as entidades e os DTOs.

Objetivos do Sistema

- **Cadastro de clientes**: Registrar informações de clientes como nome, data de cadastro e endereço.
- **Consultas de clientes**: Permitir a busca de clientes cadastrados.
- **Alteração e exclusão de clientes**: Modificar ou remover clientes do sistema.
- **Pesquisa de clientes**: Permitir buscas detalhadas com base em parâmetros como nome ou outros atributos.
- **Preenchimento automático de endereço**: Quando o CEP é informado, o sistema consulta a API ViaCEP para preencher os dados de endereço automaticamente.

Arquitetura do Sistema

A aplicação segue uma arquitetura baseada no padrão **MVC (Model-View-Controller)**, com a comunicação feita via HTTP para uma API RESTful.

- **Model**: Representa a estrutura de dados do sistema, incluindo as entidades `Cliente`, `Contato` e `Endereco`.
- **Controller**: Define os endpoints da API que expõem as funcionalidades para o usuário.
- **Service**: Contém a lógica de negócios da aplicação.
- **Repository**: Interage diretamente com o banco de dados para armazenar e recuperar os dados.
- **DTO (Data Transfer Object)**: Utilizado para transferir dados entre as camadas de forma mais eficiente.
- **Mapper**: Mapeia as entidades para os DTOs e vice-versa.

Requisitos Técnicos

- **Java 17** ou superior.
- **Spring Boot**: Framework utilizado para o desenvolvimento da aplicação.
- **Banco de Dados Relacional MySQL**: Utilizado para armazenar os dados.
- **JPA/Hibernate** para persistência de dados.
- **ModelMapper** ou **MapStruct** para conversão entre entidades e DTOs.
- **API RESTful** com endpoints para gerenciar as operações sobre os clientes e endereços.
- **API ViaCEP** para consulta de endereço via CEP.

Estrutura do Sistema

Modelos de Dados

Cliente

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nome;
    private String dataCadastro;

    @OneToOne(cascade = CascadeType.ALL)
    private Endereco endereco;

    @OneToMany(cascade = CascadeType.ALL)
    private List<Contato> contatos;

    // Getters e Setters
}
```

Contato

```
@Entity
public class Contato {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String tipo;
    private String texto;

    @ManyToOne
    @JoinColumn(name = "cliente_id", nullable = false)
    private Cliente cliente;
```

```
    // Getters e Setters
}
```

Endereço

```
@Entity
public class Endereco {
    @Id
    private String cep;
    private String logradouro;
    private String cidade;
    private String numero;
    private String complemento;

    // Getters e Setters
}
```

DTOs

- **ClienteDTO**: Representa os dados de cliente que são enviados e recebidos nas requisições.

```
public class ClienteDTO {
    private Integer id;
    private String nome;
    private String dataCadastro;
    private EnderecoDTO endereco;

    // Getters e Setters
}
```

- **EnderecoDTO**: Representa os dados de endereço para transferência.

```
public class EnderecoDTO {
    private String cep;
    private String logradouro;
    private String cidade;
    private String numero;
    private String complemento;

    // Getters e Setters
}
```

- **ContatoDTO**: Representa os dados de contato de um cliente.

```
public class ContatoDTO {
    private Integer id;
    private String tipo;
    private String texto;

    // Getters e Setters
}
```

Mapper

Utilização de **ModelMapper** ou **MapStruct** para mapear as entidades para DTOs e vice-versa, facilitando a transferência de dados entre as camadas de forma eficiente.

```
@Mapper(componentModel = "spring")
public interface ClienteMapper {
    ClienteDTO toClienteDTO(Cliente cliente);
    Cliente toCliente(ClienteDTO clienteDTO);
}
```

Endpoints da API

Cadastro de Cliente

- **Método**: `POST`
- **Endpoint**: `/clientes`
- **Descrição**: Cadastra um novo cliente no sistema, preenchendo os dados do cliente e o endereço a partir do CEP informado.

Corpo da Requisição:

```
{
  "nome": "João da Silva",
  "dataCadastro": "2025-01-17",
  "cep": "01001-000"
}
```

-

Resposta:

```
{
  "id": 1,
  "nome": "João da Silva",
  "dataCadastro": "2025-01-17",
  "endereco": {
    "cep": "01001-000",
    "logradouro": "Praça da Sé",
```

```
    "cidade": "São Paulo",
    "numero": "100",
    "complemento": "Próximo à Catedral"
  }
}
```


-

Consulta de Cliente

- **Método**: `GET`
- **Endpoint**: `/clientes/{id}`
- **Descrição**: Consulta os dados de um cliente pelo ID.

Resposta:


```
{
  "id": 1,
  "nome": "João da Silva",
  "dataCadastro": "2025-01-17",
  "endereco": {
    "cep": "01001-000",
    "logradouro": "Praça da Sé",
    "cidade": "São Paulo",
    "numero": "100",
    "complemento": "Próximo à Catedral"
  }
}
```


-

Alteração de Cliente

- **Método**: `PUT`
- **Endpoint**: `/clientes/{id}`
- **Descrição**: Altera os dados de um cliente existente.

Corpo da Requisição:


```
{
  "nome": "João da Silva Modificado",
  "dataCadastro": "2025-01-18",
  "cep": "01001-000"
}
```


-

Exclusão de Cliente

- **Método**: `DELETE`

- **Endpoint**: `/clientes/{id}`
- **Descrição**: Exclui um cliente do sistema.

Listagem de Clientes

- **Método**: `GET`
- **Endpoint**: `/clientes`
- **Descrição**: Retorna uma lista de todos os clientes cadastrados.

Resposta:


```
[
 {
  "id": 1,
  "nome": "João da Silva",
  "dataCadastro": "2025-01-17",
  "endereco": {
   "cep": "01001-000",
   "logradouro": "Praça da Sé",
   "cidade": "São Paulo",
   "numero": "100",
   "complemento": "Próximo à Catedral"
  }
 }
]
```


-

Pesquisa de Cliente

- **Método**: `GET`
- **Endpoint**: `/clientes/search?nome={nome}`
- **Descrição**: Realiza uma busca de clientes com base no nome.

Resposta:


```
[
 {
  "id": 1,
  "nome": "João da Silva",
  "dataCadastro": "2025-01-17",
  "endereco": {
   "cep": "01001-000",
   "logradouro": "Praça da Sé",
   "cidade": "São Paulo",
   "numero": "100",
   "complemento": "Próximo à Catedral"
  }
 }
```

]

-

Problemas Conhecidos e Soluções

Problema 1: Erro ao conectar com o banco de dados

Descrição: Durante a implantação, pode ocorrer um erro de conexão com o banco de dados se as credenciais não estiverem configuradas corretamente.

Solução: Verifique se as credenciais do banco de dados estão corretas no arquivo `application.properties` ou `application.yml`. Exemplo de configuração para MySQL:

```
spring.datasource.url=jdbc:mysql://localhost:3306/nome_do_banco
spring.datasource.username=usuario
spring.datasource.password=senha
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Problema 2: API ViaCEP não retorna dados corretos

Descrição: O sistema pode não preencher os dados de endereço corretamente ao consultar a API ViaCEP.

Solução: Verifique se o CEP fornecido está correto. Caso a API ViaCEP não esteja retornando os dados corretamente, tente utilizar outro serviço de CEP, como o **WS ViaCEP** ou **CEP Aberto**, e faça ajustes no código de integração.

Problema 3: Dependências do Maven não resolvidas

Descrição: Ao executar o projeto, pode ocorrer um erro informando que algumas dependências do Maven não foram resolvidas.

Solução: Execute o comando abaixo para forçar a resolução das dependências:

```
mvn clean install
```