### Documentação do Projeto: ControleCadastro API

Link do Repositório:

https://github.com/carlossl95/ControleCadastro/tree/f13f490d87278b3dda5f0eae3d1f1fb25f9abdf2/ThomasGreg

# 1. Visão Geral

Este projeto tem como objetivo implementar uma solução para o cadastro de clientes e endereços, utilizando uma arquitetura baseada em camadas no framework **.NET Core 6.0**. O sistema expõe uma API RESTful com endpoints para realizar operações de criação, atualização, visualização e remoção de clientes e endereços. Além disso, inclui controle de autenticação e autorização, validando e gerenciando tokens.

A aplicação também inclui uma interface **front-end** desenvolvida com **ASP.NET Core**, utilizando **Razor Pages** na estrutura **MVC** para permitir que os usuários interajam com o sistema de maneira intuitiva.

# 2. Arquitetura da Solução

## Camadas do Projeto

A aplicação segue a arquitetura em camadas, separando as responsabilidades em diferentes namespaces, facilitando a manutenção, escalabilidade e testes. A arquitetura é dividida nas seguintes camadas:

- **API (ControleCadastro.API)**
  - Responsável pela exposição dos endpoints e interação com o usuário.
  - Controladores para gerenciamento de clientes, endereços e autenticação.
- **Application (ControleCadastro.Application)**
  - Contém os DTOs (Data Transfer Objects), serviços de negócios e interfaces de serviços.
  - Camada responsável pela lógica de aplicação.
- **Domain (ControleCadastro.Domain)**

    - Define as entidades de domínio (Cliente, Endereco, etc.) e as regras de negócios.
    - Contém também as interfaces de repositório e serviços específicos (ex. Token de autenticação).
- **Infra (ControleCadastro.Infra)**
    - **Ioc (ControleCadastro.Infra.Ioc):** Gerenciamento de dependências e injeção de dependências.
    - **Data (ControleCadastro.Infra.Data):** Implementação dos repositórios, mapeamento de entidades para o banco de dados e a configuração do contexto de banco de dados.

### Fluxo de Dados

1. **Solicitação do Cliente**: O cliente faz uma requisição HTTP para um endpoint da API (por exemplo, `/api/cliente`).
2. **Controller**: O controlador correspondente recebe a solicitação e chama os serviços necessários na camada `Application`.
3. **Serviço**: O serviço da camada `Application` lida com a lógica de negócios, chama os repositórios para acessar ou modificar o banco de dados.
4. **Repositório**: O repositório é responsável por realizar operações no banco de dados, utilizando SQL e Stored Procedures.
5. **Resposta**: A resposta é retornada ao controlador, que envia a resposta ao cliente.

## 3. Estrutura dos Diretórios

- **ControleCadastro.API**
    - **Controllers**: Controladores que expõem os endpoints da API.
        - `AuthorizationController`: Gerencia os clientes "fronts" que podem acessar a aplicação.
        - `ClienteController`: Gerencia as operações relacionadas aos clientes.
        - `EnderecoController`: Gerencia as operações relacionadas aos endereços dos clientes.
        - `TokenController`: Realiza a autenticação de clientes "front" e autentica usuários do front-end, gerando tokens de autenticação.
    - **Models**: Modelos de dados para representar informações em requisições e respostas (ex. `UserToken`).

    - **appsettings.json**: Contém as configurações da aplicação (conexão de banco de dados, strings de configuração).
    - **Program.cs**: Configura a inicialização da aplicação e as dependências da API.
- **ControleCadastro.Application**
    - **DTOs**: Data Transfer Objects para comunicação entre as camadas.
        - `ClienteDTO`: Modelos para operações de cliente (inclusão, atualização, visualização).
        - `EnderecoDTO`: Modelos para operações de endereço.
    - **Interfaces**: Definem contratos de serviços que serão implementados.
        - `IAutorizationService`: Interface para o serviço de autorização.
        - `IClienteService`: Interface para o serviço de cliente.
        - `IEnderecoService`: Interface para o serviço de endereço.
    - **Mappings**: Perfis de mapeamento entre entidades e DTOs.
    - **Services**: Implementações dos serviços definidos nas interfaces.
        - `AutorizationService`: Lógica de autenticação e geração de tokens.
        - `ClienteService`: Lógica para criação, atualização e visualização de clientes.
        - `EnderecoService`: Lógica para gerenciar endereços de clientes.
- **ControleCadastro.Domain**
    - **Entities**: Entidades que representam os modelos de dados do domínio.
        - `Autorization`, `Cliente`, `Endereco`, `Login`, `Retorno`: Representam as entidades de domínio.
    - **Interfaces**: Interfaces de repositórios para persistência de dados.
        - `IEnderecoRepository`, `IClienteRepository`, `IAutorizationRepository`: Interfaces para comunicação com o banco de dados.
    - **Token**: Geração e validação de tokens.
        - `ITokenGenerateService`: Interface para serviço de geração de tokens.
    - **Utilities**: Utilitários para facilitar processos do domínio.
        - `RetornoHelper`: Utilitário para formatação e retorno de resultados.
    - **Validation**: Validações e exceções do domínio.
        - `DomainExceptionValidation`: Lógica para validação de regras de negócio.
- **ControleCadastro.Infra**
    - **Ioc**: Configuração de injeção de dependências.

        - `DependencyInjection`: Contém as configurações de injeção de dependências.
        - `DependencyInjectSwagger`: Configuração do Swagger para documentação da API.
    - **Data**: Implementação de repositórios e contexto de dados.
        - **Contexto**: `ApplicationDbContext`: Representa o contexto de acesso ao banco de dados.
        - **EntityConfiguration**: Mapeamento das entidades para tabelas do banco de dados.
            - `EnderecoConfiguration`, `ClientConfiguration`, `AutorizationConfiguration`.
        - **IdentityTokenGenerateService**: Serviço responsável pela geração de tokens JWT.
        - **ProcedureService**: Serviços que utilizam Stored Procedures para operações no banco de dados.
        - **Repositories**: Implementações dos repositórios.
            - `AutorizationRepository`, `ClienteRepository`, `EnderecoRepository`.
- **ControleCadastro.Front (ASP.NET Core MVC com Razor)**
    - **Views**: Páginas Razor que compõem a interface do usuário.
        - **Login**: Tela de login onde os usuários podem se autenticar.
        - **Cadastro**: Tela de cadastro de novos usuários e clientes.
        - **Home**: Tela inicial após o login, onde o usuário pode acessar suas informações.
        - **Endereço**: Lista de endereços associados ao cliente com opções de criação, edição e exclusão.
        - **Perfil**: Tela onde o usuário pode adicionar ou alterar a foto de perfil e atualizar sua senha.
    - **Controllers**: Controladores que gerenciam as interações com o front-end.
        - `AccountController`: Gerencia a autenticação e registro de novos usuários.
        - `ClienteController`: Gerencia a interação com os dados dos clientes no front-end.
        - `EnderecoController`: Gerencia a visualização, criação, edição e exclusão de endereços.
        - `PerfilController`: Gerencia as funcionalidades do perfil do usuário, como alterar a senha e adicionar uma foto de perfil.

# 4. Funcionalidades do Front-End

A parte do **front-end** foi implementada utilizando **ASP.NET Core** com **Razor Pages** na estrutura **MVC**, proporcionando uma interface de usuário interativa e dinâmica. As principais funcionalidades do front-end incluem:

## 1. Tela de Login

- A tela de login permite que os usuários autentiquem-se utilizando suas credenciais.
- Após a autenticação, o usuário é redirecionado para a tela inicial (**Home**), onde pode acessar suas informações e funcionalidades.

## 2. Tela de Cadastro

- A tela de cadastro permite que um novo usuário ou cliente seja registrado no sistema.
- Durante o cadastro, o usuário insere informações como nome, e-mail e senha.

## 3. Tela Home

- A tela inicial apresenta as informações gerais do usuário autenticado e fornece links para acessar outras funcionalidades, como a lista de endereços e o perfil do usuário.

## 4. Gerenciamento de Endereços

- A tela de lista de endereços exibe os endereços associados ao cliente. Nessa tela, o usuário pode:
- **Excluir** um endereço diretamente da lista.
- **Criar** um novo endereço, o que direciona o usuário para uma tela separada de cadastro de endereço.
- **Editar** um endereço existente, o que redireciona o usuário para uma tela de edição onde ele pode atualizar as informações do endereço selecionado.
- Dessa forma, as operações de **criar** e **editar** são tratadas em telas específicas para garantir uma experiência de usuário mais organizada e intuitiva.

## 5. Tela de Perfil

- A tela de perfil permite ao usuário:
    - **Alterar a senha**: O usuário pode modificar sua senha.

- **Adicionar ou alterar a foto de perfil**: O usuário pode enviar uma imagem para ser utilizada como sua foto de perfil.

# 5. Regras de Negócio

## Criação de Client no Banco de Dados

Para que um cliente "front" possa se comunicar com a API, é necessário criar um **Client** diretamente no banco de dados, utilizando um script de `INSERT` na tabela `Autorization`. A tabela `Autorization` é responsável por armazenar informações sobre os clientes "front" (aplicações que farão requisições à API) e seus privilégios de acesso.

### *Exemplo de Script para Criação de um Client:*

```
sql
Copiar
INSERT INTO [ThomasGreg].[dbo].[Autorization]
    ([ClientId], [ClientSecret], [IsAdmin])
VALUES
    ('client123', 'secret123', 0);
```

- **ClientId**: Identificador único do cliente.
- **ClientSecret**: Senha ou segredo utilizado para autenticação do cliente.
- **IsAdmin**: Define se o cliente tem privilégios de administrador (`1` para admin, `0` para não admin).

## Acesso com Base no Role de Admin

- Um cliente com o campo `IsAdmin` igual a `1` (administrador) tem acesso a todos os dados de clientes e endereços na API.
- Já um cliente sem privilégios de admin (`IsAdmin = 0`) tem acesso restrito **somente aos clientes e endereços vinculados a ele**. Ou seja, um cliente "front" não administrador não pode acessar dados de outros clientes.

Essa arquitetura garante que diferentes "frontends" (clientes) possam utilizar a mesma aplicação sem risco de vazamento de dados entre eles. Cada cliente tem um escopo de dados exclusivo, o que melhora a segurança da aplicação, permitindo que múltiplos "frontends" possam operar de forma isolada.

# 6. Segurança: Criptografia de Senhas com Argon2

A segurança das senhas dos usuários foi implementada utilizando o algoritmo **Argon2**, que é considerado um dos algoritmos mais seguros e eficientes para hashing de senhas.

## Motivos para escolher Argon2:

1. **Segurança**: Argon2 é amplamente considerado um dos algoritmos mais seguros para hashing de senhas. Ele foi vencedor do Password Hashing Competition (PHC) e é resistente a ataques de força bruta e ataques de hardware acelerados, como aqueles realizados com GPUs.
2. **Configuração de Custo (Memory-Hard)**: O Argon2 é altamente configurável, permitindo que seja ajustado o uso de memória, tempo de execução e paralelismo. Isso torna mais difícil para os atacantes executarem ataques de força bruta em paralelo, aumentando a segurança.
3. **Resistência a Ataques de Preimage**: Argon2 foi projetado para proteger contra ataques de "preimage" e ataques de colisão, oferecendo maior segurança do que os algoritmos de hash mais antigos, como SHA-1 e MD5.
4. **Defesa Contra GPUs e ASICs**: O algoritmo Argon2 foi projetado especificamente para ser "memory-hard", ou seja, é intencionalmente mais lento e exige mais memória do que outros algoritmos como bcrypt e PBKDF2, o que dificulta ataques utilizando hardware especializado (como GPUs ou ASICs).

## Como funciona a Criptografia de Senhas com Argon2:

O serviço de autenticação utiliza o algoritmo Argon2 para gerar o hash seguro das senhas fornecidas pelos usuários durante o registro. Esse hash é armazenado no banco de dados, garantindo que a senha original nunca seja armazenada em texto claro. Durante o processo de login, a senha fornecida pelo usuário é comparada com o hash armazenado no banco de dados.

# 7. Guias para Desenvolvedores

## Configuração do Ambiente

1. **Instalação do Banco de Dados**

    a. Utilize o SQL Server 2016 ou superior.
    b. **Importante**: A aplicação está configurada para criar automaticamente as tabelas e executar as stored procedures ao iniciar a API. Não é necessário rodar manualmente migrações ou scripts SQL. Basta garantir que o banco de dados já esteja criado e a API será responsável pela criação das tabelas e execução das procedures automaticamente durante o processo de inicialização.
2. **Dependências**
    a. Execute o comando `dotnet restore` para restaurar as dependências do projeto.
    b. As dependências estão listadas no arquivo `ControleCadastro.API.csproj`.
3. **Configuração do Swagger**
    a. O Swagger é utilizado para documentar e testar a API.
    b. Após iniciar a aplicação, acesse o Swagger em `http://localhost:{porta}/swagger` para testar os endpoints.

### Padrões de Codificação

- Utilize o padrão de nomenclatura PascalCase para classes, métodos e propriedades.
- Adote a convenção de nomenclatura camelCase para variáveis locais e parâmetros de métodos.
- Mantenha a separação clara entre as responsabilidades nas camadas (Controller, Service, Repository).

## 8. Conclusão

Esta arquitetura foi projetada para garantir que a aplicação seja escalável, segura e de alta performance, atendendo aos requisitos de cadastro de clientes e endereços. A estrutura em camadas e o uso de boas práticas de desenvolvimento garantem que a solução seja facilmente mantida e estendida no futuro. Além disso, a aplicação já está funcional e pronta para uso imediato, com a criação automática das tabelas e procedures ao inicializar a API. A utilização do Argon2 para criptografia de senhas oferece uma camada extra de segurança, garantindo a proteção dos dados sensíveis dos usuários. O front-end em **ASP.NET Core MVC** com **Razor Pages** proporciona uma interface amigável para os usuários interagirem com o sistema.