Aplicação de CRUD

Utilizando as tecnologias solicitadas pela empresa DBM Contact Center

Desenvolvida por Lucas Cachel, Desenvolvedor

🗁 Estrutura do Projeto

O projeto está organizado em camadas e pastas para manter a separação de responsabilidades e facilitar a manutenção. A seguir, a estrutura de diretórios e arquivos:

MeuProjetoBackend/

│── bin/ # Arquivos compilados da aplicação

│── Controllers/ # Controladores da API

│── Data/ # Configuração do banco de dados e contexto do Entity Framework

│── MeuProjetoBackend.Tests/ # Testes unitários do projeto

│── Migrations/ # Arquivos de migração do banco de dados

│── Models/ # Modelos de domínio da aplicação

│── obj/ # Arquivos temporários gerados pela compilação

│── Properties/ # Configurações do projeto

│── Repositories/ # Implementação dos repositórios para acesso ao banco

│── Services/ # Lógica de negócios e serviços da aplicação

│── Tests/ # Outras implementações de testes (separadas da pasta principal de testes)

│── Validators/ # Validações de dados com FluentValidation

│── Views/ # (Se aplicável) Arquivos relacionados a visualizações

```
├── .dockerignore             # Arquivo para ignorar arquivos no build do Docker
├── .gitignore                # Arquivo para ignorar arquivos no Git
├── appsettings.Development.json  # Configuração de ambiente de desenvolvimento
├── appsettings.json          # Configuração geral da aplicação
├── Dockerfile                # Definição da imagem Docker da aplicação
├── MeuProjetoBackend.csproj  # Arquivo de configuração do projeto .NET
├── MeuProjetoBackend.http    # Arquivo de requisições HTTP para testar a API
├── MeuProjetoBackend.sln     # Solução do projeto no Visual Studio
├── Program.cs                # Arquivo principal que inicia a aplicação
├── README.txt                # Arquivo de documentação
```

📌 Descrição das Camadas e Responsabilidades

O projeto segue uma estrutura em camadas para garantir a separação de responsabilidades e facilitar a manutenção e escalabilidade.

- **Controllers:** Responsáveis por receber as requisições HTTP, validar os dados de entrada e chamar os serviços apropriados. Eles atuam como intermediários entre a interface do usuário e a lógica de negócios.
- **Models:** Contém as classes que representam as entidades do domínio da aplicação, definindo a estrutura dos dados que serão armazenados no banco.
- **Repositories:** Implementa o acesso ao banco de dados utilizando o **Entity Framework Core**. Essa camada encapsula as operações

CRUD e evita o acoplamento direto da lógica de negócios com a persistência de dados.
- **Services:** Contém a lógica de negócios da aplicação. Os controllers chamam os serviços para processar as requisições antes de acessar os repositórios.
- **Validators:** Utiliza o **FluentValidation** para validar os dados recebidos pela API antes do processamento. Isso ajuda a evitar inconsistências no banco de dados.
- **Migrations:** Contém os scripts gerados pelo **FluentMigrator** para versionamento do banco de dados. Isso permite evoluir o esquema do banco de forma controlada.
- **Tests:** Implementa testes unitários com **xUnit**, garantindo que os componentes funcionem corretamente e prevenindo regressões.
- **Data:** Define o contexto do **Entity Framework Core**, permitindo a comunicação com o banco de dados.

⚙️ Explicação sobre a Escolha de Tecnologias e Padrões de Projeto

📌 ASP.NET Core 8.0

O **.NET 8.0** foi escolhido por ser a versão **LTS (Long-Term Support)** mais recente, garantindo suporte prolongado e estabilidade para aplicações em produção. Embora o **.NET 9.0** já esteja disponível, ele não possui suporte LTS, o que pode trazer riscos para aplicações empresariais que precisam de confiabilidade a longo prazo.

📌 C#

A linguagem C# foi escolhida por ser **fortemente tipada, moderna e orientada a objetos**, permitindo o desenvolvimento seguro e escalável. Além disso, a integração com o **.NET Core** proporciona alta performance e compatibilidade com diversas plataformas.

📌 Entity Framework Core

O **Entity Framework Core** foi utilizado para a **abstração do acesso ao banco de dados**, reduzindo a necessidade de escrever queries SQL manuais e acelerando o desenvolvimento.

📌 FluentMigrator

O **FluentMigrator** foi escolhido para gerenciar as **migrações do banco de dados**, garantindo versionamento controlado e facilitando mudanças na estrutura de dados.

📌 FluentValidation

O **FluentValidation** permite definir regras de validação claras e reutilizáveis, garantindo que os dados sejam verificados antes de serem processados.

📌 xUnit

O **xUnit** foi utilizado para os **testes unitários**, pois é uma das ferramentas mais populares para testes no .NET, suportando **injeção de dependência e execução paralela**.

📌 Docker

O uso do **Docker** permite a criação de containers para a aplicação, garantindo um ambiente de execução padronizado, independente do sistema operacional.

🚧 Desafios Encontrados Durante o Desenvolvimento e Como Foram Solucionados

1. **Configuração do ambiente e dependências**
    a. Durante a instalação e configuração do **.NET Core**, **Entity Framework** e **FluentMigrator**, algumas dependências precisaram ser ajustadas. A solução foi revisar o arquivo

`csproj` e garantir que todas as versões das bibliotecas fossem compatíveis.

2. **Migração do banco de dados**
    a. Ao rodar as migrações do banco de dados, alguns erros ocorreram devido à estrutura das entidades. A solução foi revisar os mapeamentos e ajustar as **chaves primárias e relacionamentos** no **FluentMigrator**.
3. **Testes unitários falhando**
    a. Alguns testes estavam falhando devido à falta de mock dos serviços. A solução foi utilizar **Moq** para simular dependências nos testes unitários, garantindo que cada componente fosse testado isoladamente.

🧪 Plano de Testes

A aplicação implementa testes unitários utilizando **xUnit** para garantir a confiabilidade do código. Os principais cenários cobertos são:

1. **Testes de Controllers**
    a. Verifica se os endpoints retornam os códigos HTTP esperados (`200 OK`, `400 Bad Request`, `404 Not Found`).
2. **Testes de Services**
    a. Garante que a lógica de negócios está funcionando corretamente, validando regras de negócio específicas.
3. **Testes de Repositories**
    a. Simula consultas ao banco de dados utilizando **InMemoryDatabase** para evitar dependência externa.
4. **Testes de Validação**
    a. Valida se os dados inseridos seguem as regras definidas no **FluentValidation**.

Com esses testes, é possível garantir que a aplicação funcione corretamente e evitar regressões em futuras alterações.