Visão Geral do Projeto

Nome do Projeto: VestibularAPI
Objetivo: Desenvolver uma API para gerenciar as inscrições de candidatos em um sistema de vestibular, incluindo operações CRUD completas e funcionalidades específicas de consulta.

Tecnologias Utilizadas:

- **Linguagem de Programação:** C#
- **Framework:** .NET 8
- **ORM:** Entity Framework Core
- **Banco de Dados:** SQL Server
- **Padrões de Projeto:** Repository Pattern, Dependency Injection, DTO (Data Transfer Object), AutoMapper, Fluent API
- **Arquitetura:** Arquitetura em camadas (Presentation, Application, Domain, Infrastructure)

Estrutura do Projeto

O projeto segue uma arquitetura em camadas, separando as responsabilidades de cada parte do sistema:

1. **Presentation Layer** (Camada de Apresentação)
    - Contém os controladores da API que recebem as requisições HTTP e retornam as respostas. Esta camada é a única que tem um ponto de entrada (Program.cs).
2. **Application Layer** (Camada de Aplicação)
    - Contém os serviços de aplicação que implementam a lógica de negócio e orquestram as operações do sistema.
    - Implementa DTOs que são usados para transferir dados entre as camadas de apresentação e de domínio.
3. **Domain Layer** (Camada de Domínio)
    - Contém as entidades de domínio que representam os objetos de negócio, como Oferta e Inscrição.
    - Serviços de domínio que encapsulam a lógica de negócio específica.
4. **Infrastructure Layer** (Camada de Infraestrutura)
    - Contém a implementação dos repositórios que acessam o banco de dados usando Entity Framework Core.
    - Inclui as configurações do Entity Framework Core (mapeamento das entidades para o banco de dados).

Padrões de Projeto Utilizados

1. **Repository Pattern:**
    - Implementado na camada de infraestrutura, o padrão Repository abstrai o acesso ao banco de dados, encapsulando a lógica de consulta e persistência. Isso promove o desacoplamento da camada de aplicação da camada de infraestrutura, facilitando a manutenção e testes.
2. **Dependency Injection:**
    - Utilizado para gerenciar as dependências entre as camadas. A injeção de dependência é configurada no Program.cs na camada de apresentação, promovendo um design flexível e testável.
3. **DTO (Data Transfer Object):**
    - Utilizado para transferir dados entre as camadas de apresentação e aplicação. Os DTOs evitam a exposição direta das entidades de domínio na API, promovendo segurança e encapsulamento.
4. **AutoMapper:**
    - Utilizado na camada de aplicação para mapear objetos entre DTOs e entidades de domínio. O AutoMapper reduz o código de mapeamento manual, promovendo a consistência e a manutenção do código.
5. **Fluent API:**
    - Utilizado para configurar as entidades do Entity Framework Core, como definir tipos de dados, tamanhos de campos, e relações entre entidades. A configuração Fluent API é separada em arquivos específicos na camada de infraestrutura.

Implementações Específicas

1. **CRUD de Entidades:**
    - As operações CRUD (Create, Read, Update, Delete) foram implementadas para as entidades Oferta, Inscrição, e outras, conforme o diagrama fornecido.
    - Cada operação foi implementada na camada de aplicação utilizando o padrão Repository para interagir com o banco de dados.
2. **Consultas Específicas:**
    - **Inscrições por CPF:** Implementado na camada de aplicação, esse método permite a busca de todas as inscrições associadas a um determinado CPF.

- **Inscrições por Oferta:** Permite a consulta das inscrições associadas a uma determinada oferta de curso.

3. **Configuração do AutoMapper:**
    - O AutoMapper foi configurado na camada de apresentação, com perfis de mapeamento definidos na camada de aplicação. Este mapeamento converte DTOs em entidades de domínio e vice-versa.
4. **Validação com Fluent API:**
    - Exemplo: Na entidade Oferta, o campo Nome foi configurado como varchar(100) e requer um mínimo de 5 caracteres, utilizando Fluent API para garantir a integridade dos dados.

csharp

Copiar código

```
public class OfertaConfiguration : IEntityTypeConfiguration<Oferta>
{
  public void Configure(EntityTypeBuilder<Oferta> builder)
  {
    builder.Property(o => o.Nome)
      .HasColumnType("varchar(100)")
      .IsRequired()
      .HasMaxLength(100)
      .HasMinLength(5); // Hypothetical method for minimum length, otherwise use manual validation
  }
}
```

5. **AddAsync Method in Service:**
    - Implementado na classe OfertaService, o método AddAsync adiciona uma nova oferta ao banco de dados e retorna o objeto persistido, com todos os dados atualizados (como o ID gerado).

csharp

Copiar código

```
public async Task<OfertaDto> AddAsync(CreateOfertaDto createOfertaDto)
{
  var oferta = _mapper.Map<Oferta>(createOfertaDto);
  await _ofertaRepository.AddAsync(oferta);
```

```
    return _mapper.Map<OfertaDto>(oferta);
}
```

Boas Práticas Implementadas

1. **Código Limpo e Organizado:**
    - O projeto segue o princípio do *Clean Code*, com nomes de variáveis e métodos descritivos, evitando comentários desnecessários e priorizando a legibilidade do código.
2. **Separação de Responsabilidades:**
    - Cada camada é responsável por uma parte específica do sistema, promovendo a coesão e reduzindo o acoplamento entre os componentes.
3. **Uso de DTOs:**
    - Os DTOs são usados para transferir dados entre camadas, evitando a exposição direta das entidades de domínio e proporcionando uma API mais segura e flexível.
4. **Validação de Dados:**
    - A validação de dados é realizada na camada de infraestrutura usando Fluent API, garantindo que as regras de negócio sejam respeitadas antes que os dados sejam persistidos no banco.
5. **Tratamento de Exceções:**
    - O projeto inclui tratamento de exceções para capturar e gerenciar erros de forma controlada, garantindo uma melhor experiência para o usuário e facilitando a depuração.

Execução e Testes

Passos para executar o projeto:

1. **Compilar o Projeto:**
    - Navegue até a solução do projeto e compile usando dotnet build.
2. **Aplicar as Migrações:**
    - Execute dotnet ef database update na camada de apresentação para aplicar as migrações e criar o banco de dados.
3. **Executar a API:**
    - Inicie o projeto executando dotnet run na camada de apresentação. A API estará disponível na porta configurada (por padrão, https://localhost:5001).

4. **Testar as Funcionalidades:**
    - Utilize uma ferramenta como Postman ou cURL para testar as operações CRUD e as consultas específicas.

Conclusão

A **VestibularAPI** foi projetada seguindo boas práticas de arquitetura e design, utilizando tecnologias modernas e padrões de projeto reconhecidos. A API oferece uma solução robusta para o gerenciamento de inscrições em um sistema de vestibular, garantindo escalabilidade, manutenção facilitada e segurança dos dados.