Nesta segunda parte, daremos continuidade à construção da nossa *API REST em .NET com Oracle XE e Stored Procedures*.

A camada Application e sua função na organização da Regra de Negócio

A Camada **Application** é responsável por orquestrar a lógica da aplicação, fazendo a ponte entre os controladores e a infraestrutura. É aqui que centralizamos as interfaces e sua implementações concretas, como a `UserService`, que aplica regras básicas e delega persistência ao `IUserRepository`, sem depender diretamente do banco de dados.

Seguimos os princípios da Clean Architecture, mantendo a separação de responsabilidade e favorecendo testabilidade e desacoplamento. Embora a **Application** esteja organizada como uma pasta dentro do projeto principal, ela poderia facilmente ser movida para uma **class library** separada, o que é comum em aplicações maiores.

Vale ressaltar que, embora tenhamos optado por uma estrutura mais enxuta e direta, sem o uso de padrões como **Command Handlers**, **Use Cases** explícitos ou **Value Objects**, esse modelo é comum em muitos projetos reais e funciona bem em contextos onde a complexidade do domínio é moderada. A escolha visa manter o foco nos fundamentos, com uma organização clara, de fácil leitura e alinhada aos princípios modernos da arquitetura em camadas.

Implementando a Classe UserService

Com a estrutura da camada **Application** definida, seguimos para a implementação da classe `UserService`, que é a implementação concreta da interface `IUserService`, e tem como responsabilidade orquestrar as operações relacionadas a usuários. Ela atua como intermediária entre o controlador e o repositório, garantindo que a lógica de negócio permaneça desacoplada da infraestrutura.

Ao receber um `CreateUserDto`, a `UserService` utiliza a biblioteca **Flavio.Santos.NetCore.ObjectMapping** para convertê-lo em um `UserDto`, e adiciona um `Id` gerado com `Guid.NewGuid()`. Esse processo mantém o código limpo e evita duplicação. Em seguida, a instância convertida é repassada para o repositório via `IUserRepository`, que realiza a persistência no banco.

Os métodos `GetAllAsync()`, `UpdateAsync()` e `DeleteAsync()` seguem o mesmo padrão: aplicam regras simples, quando necessário, e delegam as ações ao repositório. Essa abordagem reforça a

separação de responsabilidade e contribui para uma arquitetura modular, mantendo a aplicação mais organizada, testável e alinhada aos princípios da **Clean Architecture**.

```
using OracleCrud.Sp.Api.Application.Interfaces;
using OracleCrud.Sp.Api.Domain.Dtos;
using FDS.NetCore.ObjectMapping.Extensions;

namespace OracleCrud.Sp.Api.Application.Services;

public class UserService : IUserService
{
    private readonly IUserRepository _repository;

    public UserService(IUserRepository repository)
    {
        _repository = repository;
    }

    public async Task<IEnumerable<UserDto>> GetAllAsync()
    {
        return await _repository.GetAllAsync();
    }

    public async Task<string> InsertAsync(CreateUserDto user)
    {
        var userDto = user
            .MapTo<UserDto>()
            .Apply(t => t.Id = Guid.NewGuid().ToString());

        return await _repository.InsertAsync(userDto);
    }

    public async Task<string> UpdateAsync(UserDto user)
    {
        return await _repository.UpdateAsync(user);
    }

    public async Task<string> DeleteAsync(string id)
    {
        return await _repository.DeleteAsync(id);
    }
}
```

Instalando e utilizando a biblioteca `Flavio.Santos.NetCore.ObjectMapping`

Para facilitar o mapeamento entre objetos com estrutura semelhante, utilizamos a biblioteca `Flavio.Santos.NetCore.ObjectMapping`. Ela oferece uma abordagem fluente, simples e sem configuração para converter objetos entre tipos diferentes com propriedades compatíveis, como no caso de `CreateUserDto` para `UserDto`.

A instalação pode ser feita diretamente via terminal do projeto:

```
dotnet add package Flavio.Santos.NetCore.ObjectMapping
```

Ou se preferir, pelo gerenciador de pacotes do **Visual Studio Manage Nuget Packages**.

Exemplo prático de uso da biblioteca de mapeamento

Para compreender melhor como a biblioteca **Flavio.Santos.NetCore.ObjectMapping** facilita a transformação de objetos, vejamos o seguinte cenário comum: ao receber dados de um cliente via API, é comum utilizar um **DTO** de entrada (`CreateUserDto`). Esse objeto, então, precisa ser transformado em um **DTO** interno ou persistido, como o `UserDto`.

Estrutura dos objetos:

```
// Objeto recebido via requisição HTTP
public class CreateUserDto
{
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}

// Objeto utilizado internamente ou para persistência
public class UserDto
{
```

```
    public string Id { get; set; } = default!;
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}
```

Transformação usando MapTo() e Apply():

```
CreateUserDto user = new CreateUserDto
{
    Name = "João",
    Email = "joao@email.com"
};

var userDto = user
    .MapTo<UserDto>()
    .Apply(u => u.Id = Guid.NewGuid().ToString());
```

Neste exemplo:

- O método `MapTo<UserDto>()` copia automaticamente as propriedades com nomes e tipos compatíveis (`Name` e `Email`).
- O método `Apply(…)` adiciona o valor `Id`, permitindo estender o objeto de maneira fluente, sem criar código repetitivo.

Esse tipo de abordagem é especialmente útil em serviços de aplicação e camadas intermediárias, reduzindo o acoplamento e aumentando a clareza do código.

A camada Infrastructure e seu Papel na Arquitetura

A camada **Infrastructure** é responsável por implementar os detalhes técnicos da aplicação, como o acesso ao banco de dados, chamadas externas, serviços de terceiros ou qualquer outra dependência que envolva infraestrutura. No nosso caso, ela abriga a implementação concreta da interface `IUserRepository`, utilizando a biblioteca **Oracle.ManagedDataAccess.Client** para executar a stored procedures no banco de dados **Oracle XE**. Essa separação permite que o restante da aplicação (como os serviços e controladores) dependa apenas de contratos (interfaces), mantendo o sistema mais flexível, testável e aderente ao princípio da inversão de dependência. Em projetos maiores, essa camada pode ser estruturada como uma **class library** separada, reforçando o desacoplamento entre regras de

negócio e detalhes de infraestrutura.

Implementando a Classe UserRepository

A classe `UserRepository` é responsável por concretizar o contrato definido pela interface `IUserRepository`, realizando o acesso direto ao banco de dados **Oracle**. Esta implementação reside na camada **Infrastructure**, que tem como função lidar com aspectos externos à aplicação, como persistência de dados, acesso a serviços de terceiros ou recursos do sistema.

Neste projeto, optamos por utilizar a biblioteca **Oracle.ManagedDataAccess.Client** em vez de **ORMs** como **Entity Framework** ou **Dapper**. Essa abordagem proporciona maior controle sobre a execução de **comandos SQL** e chamadas a **Stored Procedures**, sendo particularmente adequada quando se deseja os recursos específicos do **Oracle** ou seguir diretrizes de legado da empresa.

A classe encapsula operações como `GetAllAsync()`, `InsertAsync()`, `UpdateAsync()` e `DeleteAsync()`, comunicando-se com o banco por meio de **views** e **procedures** previamente definidas. O uso de comandos parametrizados garante segurança contra **SQL Injection**, além de promover clareza e previsibilidade no comportamento do repositório.

É importante observar que o `UserRepository` permanece completamente desacoplado da lógica de aplicação e de transformação de dados, sendo consumido exclusivamente pela `UserService`. Essa separação de responsabilidade está alinhada com os princípios da **Clean Architecture**, ainda que de forma simplificada e pragmática neste projeto.

```
using Oracle.ManagedDataAccess.Client;
using OracleCrud.Sp.Api.Application.Interfaces;
using OracleCrud.Sp.Api.Domain.Dtos;
using System.Data;

namespace OracleCrud.Sp.Api.Infrastructure.Repositories;

public class UserRepository : IUserRepository
{
    private readonly string _connectionString;

    public UserRepository(IConfiguration configuration)
```

```
    {
        _connectionString = configuration.GetConnectionString("OracleDb")!;
    }

    public async Task<IEnumerable<UserDto>> GetAllAsync()
    {
        var users = new List<UserDto>();

        using var connection = new OracleConnection(_connectionString);
        using var command = new OracleCommand("SELECT id, name, email FROM vw_users",
connection);

        await connection.OpenAsync();
        using var reader = await command.ExecuteReaderAsync();

        while (await reader.ReadAsync())
        {
            users.Add(new UserDto
            {
                Id = reader.GetString(0),
                Name = reader.GetString(1),
                Email = reader.GetString(2)
            });
        }

        return users;
    }

    public async Task<string> InsertAsync(UserDto user)
    {
        using var connection = new OracleConnection(_connectionString);
        using var command = new OracleCommand("app_user.sp_insert_user", connection)
        {
            CommandType = CommandType.StoredProcedure
        };

        command.Parameters.Add("p_id", OracleDbType.Varchar2).Value = user.Id;
        command.Parameters.Add("p_name", OracleDbType.Varchar2).Value = user.Name;
        command.Parameters.Add("p_email", OracleDbType.Varchar2).Value = user.Email;

        var resultParam = new OracleParameter("p_result", OracleDbType.Varchar2, 4000)
        {
            Direction = ParameterDirection.Output
        };
        command.Parameters.Add(resultParam);
```

```
        await connection.OpenAsync();
        await command.ExecuteNonQueryAsync();

        return resultParam.Value?.ToString() ?? "Erro ao inserir usuário.";
    }

    public async Task<string> UpdateAsync(UserDto user)
    {
        using var connection = new OracleConnection(_connectionString);
        using var command = new OracleCommand("app_user.sp_update_user", connection)
        {
            CommandType = CommandType.StoredProcedure
        };

        command.Parameters.Add("p_id", OracleDbType.Varchar2).Value = user.Id;
        command.Parameters.Add("p_name", OracleDbType.Varchar2).Value = user.Name;
        command.Parameters.Add("p_email", OracleDbType.Varchar2).Value = user.Email;

        await connection.OpenAsync();
        await command.ExecuteNonQueryAsync();

        return "Usuário atualizado com sucesso.";
    }

    public async Task<string> DeleteAsync(string id)
    {
        using var connection = new OracleConnection(_connectionString);
        using var command = new OracleCommand("app_user.sp_delete_user", connection)
        {
            CommandType = CommandType.StoredProcedure
        };

        command.Parameters.Add("p_id", OracleDbType.Varchar2).Value = id;

        await connection.OpenAsync();
        await command.ExecuteNonQueryAsync();

        return "Usuário excluído com sucesso.";
    }
}
```

Conclusão

Concluímos nesta segunda parte da série a implementação das camadas **Application** e **Infrastructure**, elementos essenciais para estruturar a lógica de negócios e o acesso a dados de forma organizada e

desacoplada. Criamos a classe `UserService`, responsável por orquestrar as operações da aplicação, e a classe `UserRepository`, que efetua a comunicação direta com o banco **Oracle** por meio de **Stored Procedures**, utilizando a biblioteca **Oracle.ManagedDataAccess.Client**.

Também introduzimos o uso da biblioteca **Flavio.Santos.NetCore.ObjectMapping**, que nos permitiu mapear objetos de forma simples e fluente, eliminando a necessidade de código repetitivo para transformação de **DTOs**.

Com essas camadas implementadas, a base do projeto está bem estruturada e pronta para receber a camada de apresentação.

Na **Parte 3**, vamos desenvolver os **Controllers**, expondo os **endpoints da API** e finalizando a aplicação com testes manuais **via HTTP**. Se você acompanhou até aqui, parabéns, estamos quase lá !!!