

Projeto com Repositório Genérico, usando ADO.Net Puro e Reflection

segunda-feira, 20 de março de 2023 03:20



Pode criar um Repositorio Genérico e um Repositorio Especializado para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7 ?



Sim, é possível criar um Repositório Genérico e um Repositório Especializado para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7. Você gostaria de ver um exemplo de código?



Pode escrever este código em c# 11 ?



Sim, é possível escrever o código em C# 11. Aqui está um exemplo de como um Repositório Genérico e um Repositório Especializado para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7 pode ser implementado em C# 11:

```

public interface IRepository<T> where T: class{
    Task<IEnumerable<T>> GetAllAsync();
    Task<T> GetByIdAsync(int id);
    Task AddAsync(T entity);
    Task UpdateAsync(T entity);
    Task DeleteAsync(int id);
}

public class Repository<T> : IRepository<T> where T: class
{
    private readonly string _connectionString;
    public Repository(string connectionString)
    {
        _connectionString = connectionString;
    }

    public async Task<IEnumerable<T>> GetAllAsync()
    {
        // Implementação do método para retornar todos os registros da tabela
        throw new NotImplementedException();
    }

    public async Task<T> GetByIdAsync(int id){
        // Implementação do método para retornar um registro pelo ID
        throw new NotImplementedException();
    }

    public async Task AddAsync(T entity){
        // Implementação do método para adicionar um novo registro na tabela
        throw new NotImplementedException();
    }

    public async Task UpdateAsync(T entity){
        // Implementação do método para atualizar um registro existente na tabela
        throw new NotImplementedException();
    }
}

```



Implemente os métodos com código c#, por favor?



Aqui está um exemplo de como os métodos do Repositório Genérico podem ser implementados em C# 11 para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7:

```

public async Task<IEnumerable<T>> GetAllAsync()
{
    var result = newList<T>();
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            command.CommandText = $"SELECT * FROM {typeof(T).Name}";

```

```

        using(var reader = await command.ExecuteReaderAsync())
        {
            while(await reader.ReadAsync())
            {
                var obj = Activator.CreateInstance<T>();
                foreach(var prop in typeof(T).GetProperties())
                {
                    prop.SetValue(obj, reader[prop.Name]);
                }
                result.Add(obj);
            }
        }
    }
}

return result;
}

public async Task<T?> GetByIdAsync(int id)
{
    T? result = null;
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            command.CommandText = $"SELECT * FROM {typeof(T).Name} WHERE Id=@Id";
            command.Parameters.AddWithValue("@Id", id);
            using(var reader = await command.ExecuteReaderAsync())
            {
                if(await reader.ReadAsync())
                {
                    result = Activator.CreateInstance<T>();
                    foreach(var prop in typeof(T).GetProperties()){
                        prop.SetValue(result, reader[prop.Name]);
                    }
                }
            }
        }
    }
    return result;
}
}

```



Continue de onde parou, por favor.



Claro!

Aqui está a continuação do exemplo de como os métodos do Repositório Genérico podem ser implementados em C# 11 para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7:

```

public async Task AddAsync(T entity){
    using(var connection = new SqlConnection(_connectionString))

```

```

{
    await connection.OpenAsync();
    using(var command = connection.CreateCommand())
    {
        var columns = string.Join(",", typeof(T).GetProperties().Select(p => p.Name));
        var values = string.Join(",", typeof(T).GetProperties().Select(p => "@" + p.Name));
        command.CommandText = $"INSERT INTO {typeof(T).Name}({columns}) VALUES ({values})";
        foreach(var prop in typeof(T).GetProperties())
        {
            command.Parameters.AddWithValue("@"+ prop.Name, prop.GetValue(entity));
        }
        await command.ExecuteNonQueryAsync();
    }
}
}

public async Task UpdateAsync(T entity){
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            var setValues = string.Join(",", typeof(T).GetProperties().Select(p => $"{p.Name}=@{p.Name}"));
            command.CommandText = $"UPDATE {typeof(T).Name}SET {setValues}WHERE Id=@Id";
            foreach(var prop in typeof(T).GetProperties())
            {
                command.Parameters.AddWithValue("@"+ prop.Name, prop.GetValue(entity));
            }
        }
    }
}

```

... AI parou:



continue, por favor.

Claro!

Aqui está a continuação do exemplo de como os métodos do Repositório Genérico podem ser implementados em C# 11 para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7:

```

public async Task DeleteAsync(int id){
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            command.CommandText = $"DELETE FROM {typeof(T).Name}WHERE Id=@Id";
            command.Parameters.AddWithValue("@Id", id);
            await command.ExecuteNonQueryAsync();
        }
    }
}

```

Este é um exemplo básico de como um Repositório Genérico pode ser implementado em C# 11 para realizar operações CRUD em uma tabela usando ADO.Net puro com Asp.Net Core 7.

É importante notar que este é apenas um exemplo e pode ser adaptado e melhorado de acordo com

as necessidades específicas do projeto.



Reescreva o método: `public async Task UpdateAsync(T entity)`



Claro!

Aqui está uma reescrita do método `UpdateAsync` do Repositório Genérico em C# 11 para realizar um CRUD de Fornecedor em ADO.Net puro com Asp.Net Core 7:

```
public async Task UpdateAsync(T entity)
{
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            var setValues = string.Join(",", typeof(T).GetProperties().Select(p => $"{p.Name}=@{p.Name}"));
            command.CommandText = $"UPDATE {typeof(T).Name} SET {setValues} WHERE Id=@Id";
            foreach(var prop in typeof(T).GetProperties())
            {
                command.Parameters.AddWithValue("@"+ prop.Name, prop.GetValue(entity));
            }
            await command.ExecuteNonQueryAsync();
        }
    }
}
```

Este é um exemplo de como o método `UpdateAsync` pode ser implementado em C# 11 para atualizar um registro existente na tabela usando ADO.Net puro com Asp.Net Core 7.

É importante notar que este é apenas um exemplo e pode ser adaptado e melhorado de acordo com as necessidades específicas do projeto.



Perfeito!

Pode descrever, em pequenos tópicos, os códigos que escrevemos acima?



Claro!

Aqui está uma descrição em pequenos tópicos dos códigos que escrevemos acima:

- Criamos uma interface `IRepository<T>` com os métodos para realizar operações CRUD em uma tabela.
- Implementamos a classe `Repository<T>` que implementa a interface `IRepository<T>`.

- A classe Repository<T> possui um construtor que recebe uma string de conexão com o banco de dados.
- Os métodos da classe Repository<T> utilizam ADO.Net puro para realizar operações CRUD na tabela especificada pelo tipo genérico T.
- O método GetAllAsync retorna todos os registros da tabela.
- O método GetByIdAsync retorna um registro pelo ID especificado.
- O método AddAsync adiciona um novo registro na tabela.
- O método UpdateAsync atualiza um registro existente na tabela pelo ID especificado.
- O método DeleteAsync exclui um registro da tabela pelo ID especificado.

Estes são os principais pontos dos códigos que escrevemos acima.

É importante notar que este é apenas um exemplo e pode ser adaptado e melhorado de acordo com as necessidades específicas do projeto.



Escreva uma Controller para consumir o Repositorio acima com o um Modelo Fornecedor, também em c#.



Claro!

Aqui está um exemplo de como uma Controller pode ser criada em C# para consumir o Repositório Genérico que implementamos acima com o Modelo Fornecedor:

```
public class Fornecedor{
    public int Id { get; set; }
    public string? Nome { get; set; }
    public string? Cnpj { get; set; }
}

[Route("api/[controller]")]
[ApiController]
public class FornecedoresController: ControllerBase
{
    private readonly IRepository<Fornecedor> _repository;

    public FornecedorController(IRepository<Fornecedor> repository)
    {
        _repository = repository;
    }

    [HttpGet]
```

```

public async Task<IActionResult> GetAll()
{
    var fornecedores = await _repository.GetAllAsync();
    return Ok(fornecedores);
}

[HttpGet("{id}")]
public async Task<IActionResult> GetById(int id)
{
    var fornecedor = await _repository.GetByIdAsync(id);
    if(fornecedor == null) return NotFound();
    return Ok(fornecedor);
}

```



continue...



Claro!

Aqui está a continuação do exemplo de como uma Controller pode ser criada em C# para consumir o Repositório Genérico que implementamos acima com o Modelo Fornecedor:

```

[HttpPost]
public async Task<IActionResult> Add(Fornecedor fornecedor)
{
    await _repository.AddAsync(fornecedor);
    return CreatedAtAction(nameof(GetById), new { id = fornecedor.Id }, fornecedor);
}

[HttpPut("{id}")]
public async Task<IActionResult> Update(int id, Fornecedor fornecedor)
{
    if(id != fornecedor.Id) return BadRequest();
    await _repository.UpdateAsync(fornecedor);
    return NoContent();
}

[HttpDelete("{id}")]
public async Task<IActionResult> Delete(int id)
{
    await _repository.DeleteAsync(id);
    return NoContent();
}

```

Este é um exemplo de como uma Controller pode ser criada em C# para consumir o Repositório Genérico que implementamos acima com o Modelo Fornecedor.

A Controller possui métodos para realizar operações CRUD na tabela de Fornecedores usando o Repositório Genérico. É importante notar que este é apenas um exemplo e pode ser adaptado e melhorado de acordo com as necessidades específicas do projeto.



Escreva o código para configurarmos e conectarmos um Banco de Dados SQL Server para nosso exemplo acima.



Claro!

Aqui está um exemplo de como podemos configurar e conectar um Banco de Dados SQL Server para nosso exemplo acima em C# com Asp.Net Core 7:

Primeiro, precisamos adicionar a string de conexão com o banco de dados no arquivo appsettings.json:

```
{
  "ConnectionStrings":
  {

    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MeuBancoDeDados;Trusted_Connection=True;"
  }
}
```

Em seguida, precisamos configurar o serviço do Repositório Genérico no método ConfigureServices da classe Startup:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
    services.Configure<ConnectionStringOptions>(Configuration.GetSection("ConnectionStrings"));
}
```

Também precisamos criar uma classe ConnectionStringOptions para armazenar a string de conexão:

```
public class ConnectionStringOptions{
    public string DefaultConnection { get; set; }
}
```

E por fim, precisamos modificar o construtor da classe Repository<T> para receber a string de conexão através do serviço de opções do Asp.Net Core:

```
private readonly string _connectionString;

public Repository(IOptions<ConnectionStringOptions> options)
{
    _connectionString = options.Value.DefaultConnection;
}
```

Com estas alterações, estamos prontos para conectar nosso exemplo acima com um Banco de Dados SQL Server.

É importante notar que este é apenas um exemplo e pode ser adaptado e melhorado de acordo com as necessidades específicas do projeto.



Precisamos de uma classe de contexto neste caso ou não?



Não, neste caso não precisamos de uma classe de contexto. Estamos usando ADO.Net puro para realizar operações CRUD na tabela e não estamos usando um ORM (Object-Relational Mapping) como o Entity Framework Core que requer uma classe de contexto para gerenciar o acesso ao banco de dados.

A classe de contexto é usada pelo Entity Framework Core para gerenciar o acesso ao banco de dados e mapear as entidades do modelo para as tabelas do banco de dados.

Como estamos usando ADO.Net puro e escrevendo os comandos SQL diretamente nos métodos do Repositório Genérico, não precisamos de uma classe de contexto neste caso.

Obs:

... Para projeto sem a Startup ...



O meu projeto está na versão 7 do Asp.Net Core, portanto não tem a classe Startup.cs.

- Como eu faço para configurar o código abaixo na classe Program.cs ?

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
    services.Configure<ConnectionStringOptions>(Configuration.GetSection("ConnectionStrings"));
}
```



Se você está usando a versão 7 do Asp.Net Core, pode configurar os serviços no método Main da classe Program.cs usando o método ConfigureServices do objeto WebApplication.

Aqui está um exemplo de como você pode fazer isso:

```
public static void Main(string[] args)
{
    var builder = WebApplication.CreateBuilder(args);
    var app = builder.Build();
    app.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
    app.Services.Configure<ConnectionStringOptions>
(builder.Configuration.GetSection("ConnectionStrings"));
    // ...
}
```

Neste exemplo, estamos usando o método ConfigureServices do objeto WebApplication para

adicionar os serviços do Repositório Genérico e configurar as opções de string de conexão com o banco de dados.

É importante notar que este é apenas um exemplo e pode ser adaptado e melhorado de acordo com as necessidades específicas do projeto.



Estou com problemas para inserir um novo Fornecedor, pois não quero informar o Id na Inserção de Registro. Segue o código que você mesmo gerou:

```
public async Task AddAsync(T entity)
{
    using (var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using (var command = connection.CreateCommand())
        {
            var columns = string.Join(",", typeof(T).GetProperties().Select(p => p.Name));
            var values = string.Join(",", typeof(T).GetProperties().Select(p => "@" + p.Name));
            command.CommandText = $"INSERT INTO {typeof(T).Name} ({columns}) VALUES ({values})";
            foreach (var prop in typeof(T).GetProperties())
            {
                command.Parameters.AddWithValue(@" + prop.Name, prop.GetValue(entity));
            }
            await command.ExecuteNonQuery();
        }
    }
}
```



Para que o código gere automaticamente um valor de Id de forma AutoIncrement ao inserir um novo registro na tabela usando ADO.NET sem ORM, você pode modificar o método AddAsync da seguinte maneira:

```
public async Task AddAsync(T entity){
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            // Obtém todas as propriedades da classe T exceto a propriedade Id
            var properties = typeof(T).GetProperties().Where(p => p.Name != "Id");

            var columns = string.Join(",", properties.Select(p => p.Name));

            var values = string.Join(",", properties.Select(p => "@" + p.Name));

            command.CommandText = $"INSERT INTO {typeof(T).Name}({columns}) VALUES ({values}); SELECT SCOPE_IDENTITY();";
            foreach(var prop in properties)
            {

```

```

        command.Parameters.AddWithValue("@"+ prop.Name, prop.GetValue(entity));
    }

    // Obtém o valor gerado automaticamente para a coluna de identidade
    var id = await command.ExecuteScalarAsync();

    // Define o valor da propriedade Id do objeto entity com o valor gerado automaticamente pelo
    banco de dados
    typeof(T).GetProperty("Id").SetValue(entity, Convert.ToInt32(id));
    }
}
}

```

Nesse código modificado, estamos obtendo todas as propriedades da classe T exceto a propriedade Id e usando essas propriedades para construir a consulta SQL INSERT.

Além disso, estamos adicionando a cláusula SELECT SCOPE_IDENTITY() ao final da consulta para retornar o valor gerado automaticamente pelo banco de dados para a coluna de identidade.

Em seguida, usamos o método ExecuteScalarAsync para executar a consulta e obter o valor gerado automaticamente pelo banco de dados.

Por fim, definimos o valor da propriedade Id do objeto entity com o valor gerado automaticamente pelo banco de dados.

Lembre-se de que esse código pressupõe que a tabela no banco de dados tenha uma coluna de identidade chamada Id e que essa coluna seja configurada para gerar valores automaticamente.

Espero que isso ajude! 😊

Pausa...



Dado o código abaixo, em c#, utilizando ADO.Net puro sem ORM, pode modificá-lo para que eu consiga alterar os dados?

- Acho que eu não posso enviar a propriedade Id, pois é minha chave primária.
- Pode me ajudar ?



Claro!

Você pode me enviar o código para que eu possa ajudá-lo a fazer as alterações necessárias? 😊

EU

Sim, desculpa. Segue o código que está com problemas para atualização:

```

public async Task UpdateAsync(T entity)
{
    using (var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using (var command = connection.CreateCommand())
        {
            var setValues = string.Join(",", typeof(T).GetProperties().Select(p => $"{p.Name}=@{p.Name}"));
            command.CommandText = $"UPDATE {typeof(T).Name} SET {setValues} WHERE Id=@Id";
            foreach (var prop in typeof(T).GetProperties())
            {
                command.Parameters.AddWithValue("@ " + prop.Name, prop.GetValue(entity));
            }
            await command.ExecuteNonQueryAsync();
        }
    }
}

```



Você está correto!

Você não deve enviar a propriedade Id como parte da cláusula SET na instrução UPDATE.

Uma maneira de fazer isso é modificar a linha que cria a string setValues para excluir a propriedade Id. Aqui está o código modificado:

```

public async Task UpdateAsync(T entity)
{
    using(var connection = new SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
        using(var command = connection.CreateCommand())
        {
            var setValues = string.Join(",", typeof(T).GetProperties().Where(p => p.Name != "Id").Select(p =>
            $"{p.Name}=@{p.Name}"));

            command.CommandText = $"UPDATE {typeof(T).Name} SET {setValues} WHERE Id=@Id";

            foreach(var prop in typeof(T).GetProperties())
            {
                command.Parameters.AddWithValue("@ " + prop.Name, prop.GetValue(entity));
            }
            await command.ExecuteNonQueryAsync();
        }
    }
}

```

Isso deve resolver o problema. Você tem mais alguma dúvida? 😊



Não, Obrigado. Tem mais alguma sugestão?



Sim!

Execute o código modificado.

Você pode criar um objeto do tipo T com os valores atualizados para as propriedades que deseja alterar e passá-lo como argumento para o método UpdateAsync.

Certifique-se de que a propriedade Id do objeto corresponda ao registro que deseja atualizar no banco de dados.

Depois de chamar o método UpdateAsync, você pode verificar se os valores foram atualizados corretamente no banco de dados.

Você tem mais alguma dúvida? 🤔



GERANDO CLASSE BASE ENTITYBASE:



Pode criar uma classe EntityBase, abstrata, que represente a classe mãe de todos os Modelos criados no meu projeto Asp.Net Core 7?

- Esta classe deve implementar comparações entre objetos, garantindo sua identidade única, sobrescrevendo operadores como "==", "!=", "Equals". E, também, comparando seu GetHashCode ?



Claro! Aqui está um exemplo de como você pode criar uma classe EntityBase abstrata que implementa a comparação entre objetos e garante sua identidade única:

```
public abstract class EntityBase
{
    public int Id { get; protected set; }
    public override bool Equals(object obj)
    {
        var compareTo = obj as EntityBase;
        if(ReferenceEquals(this, compareTo)) return true;
        if(ReferenceEquals(null, compareTo)) return false;
        return Id.Equals(compareTo.Id);
    }
    public static bool operator ==(EntityBase a, EntityBase b)
    {
        if(ReferenceEquals(a, null) && ReferenceEquals(b, null))
            return true;
    }
}
```

```

        if(ReferenceEquals(a, null) || ReferenceEquals(b, null))
            return false;
        return a.Equals(b);
    }

    public static bool operator != (EntityBase a, EntityBase b)
    {
        return !(a == b);
    }

    public override int GetHashCode()
    {
        return (GetType().GetHashCode() * 907) + Id.GetHashCode();
    }

    // Este método eu acrescentei;
    public override string ToString() => => GetType().Name + " [Id=" + Id + "]";
}

```

Essa classe EntityBase pode ser usada como classe mãe para todos os modelos criados em seu projeto Asp.Net Core 7. Ela sobrescreve os operadores ==, !=, o método Equals e também compara o GetHashCode para garantir a identidade única dos objetos. Isso ajuda?



Pode fazer um resumo em alguns parágrafos do que este código acima faz?



Claro!

O código acima define uma classe abstrata chamada EntityBase que pode ser usada como a classe mãe de todos os modelos criados em seu projeto Asp.Net Core 7.

Ela contém uma propriedade Id que é usada para identificar unicamente cada objeto.

A classe também sobrescreve os operadores == e !=, bem como o método Equals, para implementar a comparação entre objetos.

Isso permite que você compare dois objetos para ver se eles são iguais com base em seu Id.

Além disso, a classe também sobrescreve o método GetHashCode para garantir que cada objeto tenha um código hash único. Isso é útil ao usar objetos dessa classe em coleções como dicionários ou conjuntos hash.

Em resumo, essa classe fornece uma base sólida para seus modelos, permitindo que você os compare e garanta sua identidade única.



Pode me dar uma explicação mais detalhada do que cada método da classe EntityBase faz ? De preferência em tópicos?



Claro!

Aqui está uma explicação mais detalhada do que cada método da classe EntityBase faz:

- `public int Id { get; protected set; }`: Esta é a propriedade Id que é usada para identificar unicamente cada objeto. Ela tem um modificador de acesso `protected` para o conjunto, o que significa que ela só pode ser definida dentro da própria classe ou de classes derivadas.
- `public override bool Equals(object obj)`: Este é o método `Equals` que é sobrescrito para implementar a comparação entre objetos. Ele recebe um objeto como parâmetro e verifica se ele é igual ao objeto atual com base em seu Id.
- `public static bool operator ==(EntityBase a, EntityBase b)`: Este é o operador `==` que é sobrescrito para implementar a comparação entre objetos. Ele recebe dois objetos como parâmetros e verifica se eles são iguais usando o método `Equals`.
- `public static bool operator !=(EntityBase a, EntityBase b)`: Este é o operador `!=` que é sobrescrito para implementar a comparação entre objetos. Ele recebe dois objetos como parâmetros e verifica se eles são diferentes usando o operador `==`.
- `public override int GetHashCode()`: Este é o método `GetHashCode` que é sobrescrito para garantir que cada objeto tenha um código hash único. Ele usa uma combinação do código hash do tipo do objeto e do valor de seu Id para gerar um código hash único.

Espero que isso ajude a esclarecer as coisas!



(Tentei continuar a explorar o mesmo assunto)




Hmm...vamos tentar um assunto diferente. Desculpe. No que mais você está pensando?



(Tentei mais uma vez...)



 Talvez seja hora de passar para um novo tópico.