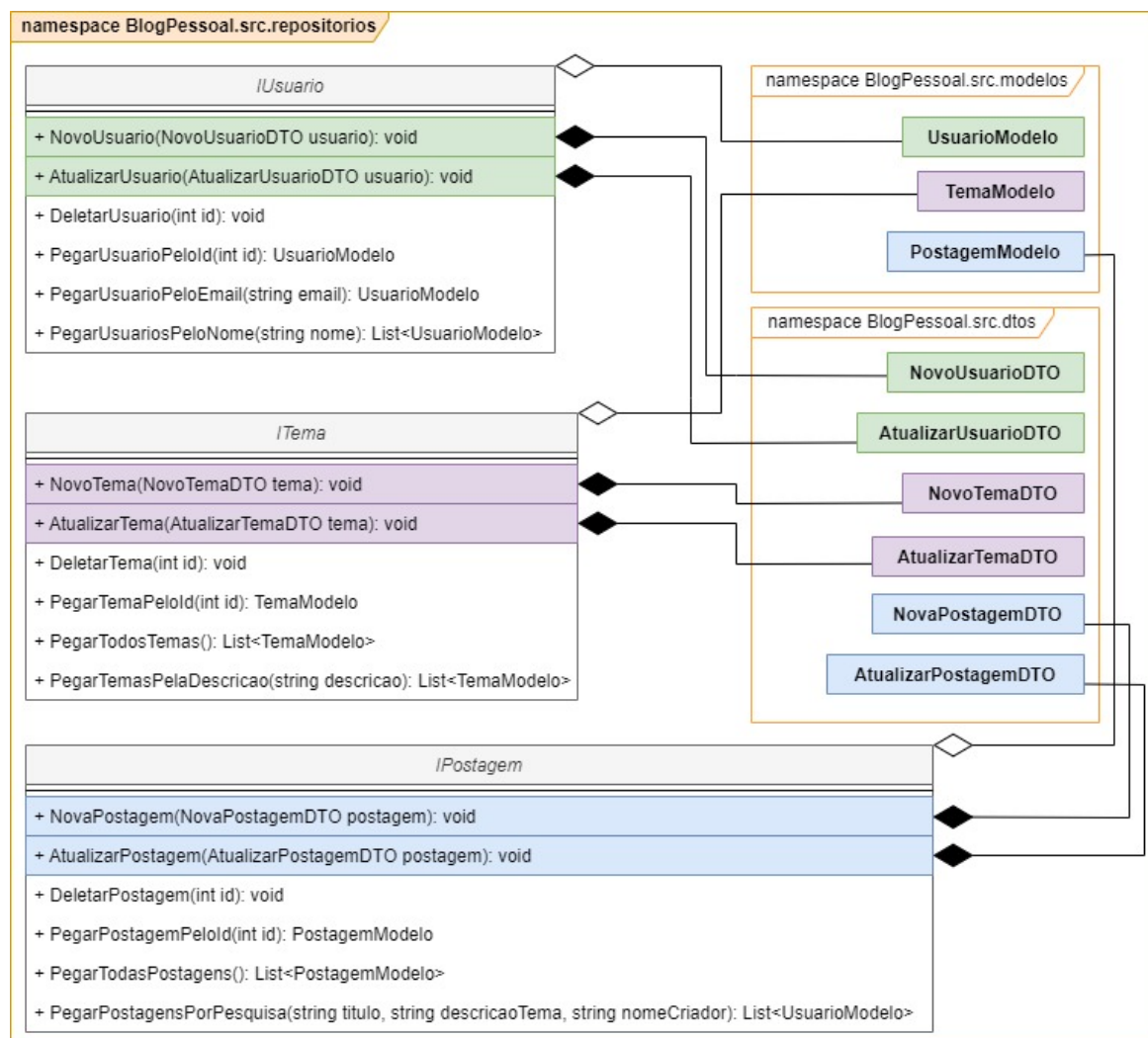


# ASPNET - 3 - Blog Pessoal - Implementar interfaces

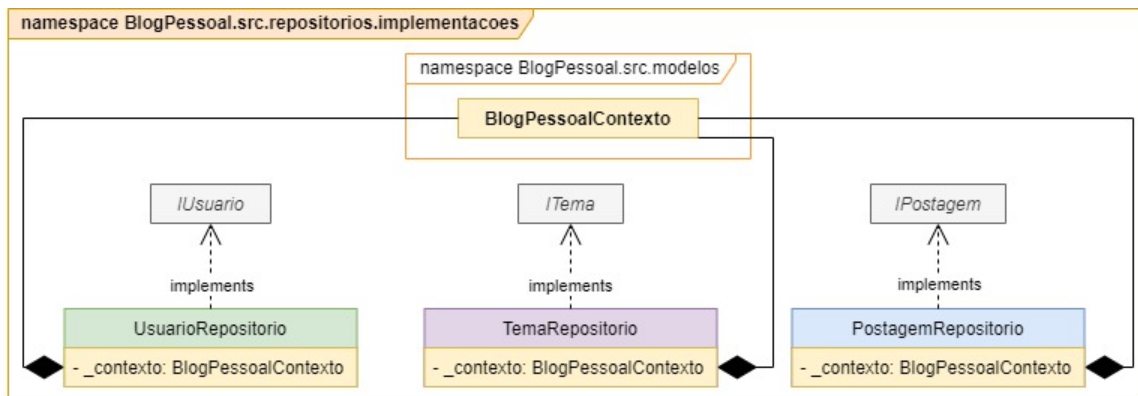
1. Definição da camada de implementações repositório
2. Implementando métodos UsuarioRepositorio
3. Implementando métodos TemaRepositorio
4. Implementando métodos PostagemRepositorio
5. Adicionando escopo de interfaces repositórios
6. Teste unitário

## 1. Definição da camada repositório

Esta camada repositório em sua pasta de implementações é onde implementamos o comportamento proposto pelas interfaces **IUsuario**, **ITema** e **IPostagem**. Abaixo segue o diagrama de classes do esquema.



Com a assinatura escrita pelas interfaces é possível observar o comportamento que o método deve tomar, é muito importante acompanhar a semântica da palavra que define o método para saber o que de fato devera ser montado. Dentro do `namespace BlogPessoal.src.repositorios.implementacoes` é necessário implementar (*implements*), as interfaces para assim ser utilizada pelo sistema.



## 2. Implementando métodos UsuarioRepositorio

A classe `UsuarioRepositorio` deve ser implementada em um arquivo chamado `UsuarioRepositorio.cs` dentro da pasta de `(src/repositorios/implementacoes)`. A estrutura da classe deve estar da seguinte maneira:

```
using System.Collections.Generic;
using System.Linq;
using BlogPessoal.src.data;
using BlogPessoal.src.dtos;
using BlogPessoal.src.modelos;

namespace BlogPessoal.src.repositorios.implementacoes
{
    public class UsuarioRepositorio : IUsuario
    {
        #region Atributos

        private readonly BlogPessoalContexto _contexto;

        #endregion Atributos

        #region Construtores

        public UsuarioRepositorio(BlogPessoalContexto contexto)
        {
            _contexto = contexto;
        }

        #endregion Construtores

        #region Métodos

        #endregion Métodos
    }
}
```

Em C# é possível definir `#region` onde podemos delimitar aonde irá cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificá-la com `#endregion`. Na `#region Atributos` foi definida a variável `_contexto` do tipo `BlogPessoalContexto`, que carrega o contexto de dados para ser utilizado pela classe. Esse contexto apenas é possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve à injeção de dependências do asp.net.

## Método PegarUsuarioPeloEmail (#region Métodos):

Este método tem a responsabilidade de buscar um único registro de usuário no banco de dados utilizando o E-mail passado como parâmetro. Sua implementação se dá na classe `UsuarioRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public UsuarioModelo PegarUsuarioPeloEmail(string email)
{
    return _contexto.Usuarios.FirstOrDefault(u => u.Email == email);
}
```

O trecho de código acima utiliza o contexto de usuário para pesquisar o primeiro elemento incidente na pesquisa com o E-mail igual ao passado como parâmetro da função.

## Método PegarUsuarioPeloNome (#region Métodos):

Este método tem a responsabilidade de buscar uma lista de usuários no banco de dados utilizando o nome passado como parâmetro. Sua implementação se dá na classe `UsuarioRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public List<UsuarioModelo> PegarUsuarioPeloNome(string nome)
{
    return _contexto.Usuarios
        .Where(u => u.Nome.Contains(nome))
        .ToList();
}
```

O trecho de código acima utiliza o contexto de usuário para pesquisar uma lista que é definida pelo parâmetro Nome. Nesta pesquisa é possível ver uma cláusula `where` em conjunto com `Contains` que representa o `Like` no SQL.

## Método PegarUsuarioPeloid (#region Métodos):

Este método tem a responsabilidade de buscar um único registro de usuário no banco de dados utilizando o Id passado como parâmetro. Sua implementação se dá na classe `UsuarioRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public UsuarioModelo PegarUsuarioPeloid(int id)
{
    return _contexto.Usuarios.FirstOrDefault(u => u.Id == id);
}
```

O trecho de código acima utiliza o contexto de usuário para pesquisar o primeiro elemento incidente na pesquisa com o Id igual ao passado como parâmetro da função.

## Método NovoUsuario (#region Métodos):

Este método tem a responsabilidade de criar um usuário no banco de dados e sua implementação se dá na classe `UsuarioRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public void NovoUsuario(NovoUsuarioDTO usuario)
{
    _contexto.Usuarios.Add(new UsuarioModelo
    {
        Email = usuario.Email,
        Nome = usuario.Nome,
        Senha = usuario.Senha,
        Foto = usuario.Foto
    });
    _contexto.SaveChanges();
}
```

O trecho de código chama o contexto da aplicação e cria um construtor de `UsuarioModelo`, todos os parâmetros da dto `NovoUsuarioDTO`, são passados para o modelo, e ao final o contexto salva as mudanças no banco.

## Método AtualizarUsuario (#region Métodos):

Este método tem a responsabilidade de atualizar um usuário existente no banco de dados e sua implementação se dá na classe `UsuarioRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public void AtualizarUsuario(AtualizarUsuarioDTO usuario)
{
    var usuarioExistente = PegarUsuarioPeloId(usuario.Id);
    usuarioExistente.Nome = usuario.Nome;
    usuarioExistente.Senha = usuario.Senha;
    usuarioExistente.Foto = usuario.Foto;
    _contexto.Usuarios.Update(usuarioExistente);
    _contexto.SaveChanges();
}
```

O método `PegarUsuarioPeloId` recebe um parâmetro de `usuario.Id` passado pelo dto `AtualizarUsuarioDTO`, o retorno passado pela método é um modelo de usuário existente no banco de dados. Com esse objeto existente em mãos todos os parâmetros da dto `AtualizarUsuarioDTO`, são passados para o modelo. Para enviar a ação é necessário chamar o contexto de usuários e acionar o método para atualizar, isso é visto no trecho `_contexto.Usuarios.Update(usuarioExistente);`, passando como parâmetro o modelo alterado.

## Método DeletarUsuario (#region Métodos):

Este método tem a responsabilidade de deletar um usuário existente no banco de dados através do Id passado como parâmetro e sua implementação se dá na classe `UsuarioRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public void DeletarUsuario(int id)
{
    _contexto.Usuarios.Remove(PegarUsuarioPeloId(id));
    _contexto.SaveChanges();
}
```

O método `PegarUsuarioPeloId` recebe um parâmetro de `usuario.Id` passado como parâmetro, o retorno passado pelo método é um modelo de usuário existente no banco de dados. Com esse objeto existente em mãos é passado para o contexto de usuário em seu método `Remove`, ao final para finalizar a transação é chamado o contexto novamente e finalizado as mudanças.

## 3. Implementando métodos TemaRepositorio

A classe `TemaRepositorio` deve ser implementada em um arquivo chamado `TemaRepositorio.cs` dentro da pasta de `(src/repositorios/implementacoes)`. A estrutura da classe deve estar da seguinte maneira:

```
using System.Collections.Generic;
using System.Linq;
using BlogPessoal.src.data;
using BlogPessoal.src.dtos;
using BlogPessoal.src.modelos;

namespace BlogPessoal.src.repositorios.implementacoes
{
    public class TemaRepositorio : ITema
    {
        #region Atributos

        private readonly BlogPessoalContexto _contexto;

        #endregion Atributos

        #region Construtores

        public TemaRepositorio(BlogPessoalContexto contexto)
        {
            _contexto = contexto;
        }

        #endregion Construtores

        #region Métodos

        #endregion Métodos
    }
}
```

```
}  
}
```

Em C# é possível definir `#region` onde podemos delimitar aonde irá cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificar de fechá-la com `#endregion`. Na `#region Atributos` foi definida a variável `_contexto` do tipo `BlogPessoalContexto`, que carrega o contexto de dados para ser utilizado pela classe. Esse contexto apenas é possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve a injeção de dependências do asp.net.

## Método PegarTodosTemas (#region Métodos):

Este método tem a responsabilidade de buscar uma lista de temas no banco de dados utilizando. Sua implementação se dá na classe `TemaRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public List<TemaModelo> PegarTodosTemas()  
{  
    return _contexto.Temas.ToList();  
}
```

O trecho de código acima utiliza o contexto de tema para pesquisar uma lista de temas

## Método PegarTemaPelaDescricao (#region Métodos):

Este método tem a responsabilidade de buscar uma lista de temas no banco de dados utilizando a descrição passada como parâmetro. Sua implementação se dá na classe `TemaRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public List<TemaModelo> PegarTemaPelaDescricao(string descricao)  
{  
    return _contexto.Temas  
        .Where(u => u.Descricao.Contains(descricao))  
        .ToList();  
}
```

O trecho de código acima utiliza o contexto de tema para pesquisar uma lista que é definida pelo parâmetro `descricao`. Nesta pesquisa é possível ver uma cláusula `where` em conjunto com `contains` que representa o `Like` no SQL.

## Método PegarTemaPeloid (#region Métodos):

Este método tem a responsabilidade de buscar um único registro de tema no banco de dados utilizando o Id passado como parâmetro. Sua implementação se dá na classe `TemaRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public TemaModelo PegarTemaPeloid(int id)  
{  
    return _contexto.Temas.FirstOrDefault(u => u.Id == id);  
}
```

O trecho de código acima utiliza o contexto de tema para pesquisar o primeiro elemento incidente na pesquisa com o Id igual ao passado como parâmetro da função.

## Método NovoTema (#region Métodos):

Este método tem a responsabilidade de criar um tema no banco de dados e sua implementação se da na classe `TemaRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public void NovoTema(NovoTemaDTO tema)
{
    _contexto.Temas.Add(new TemaModelo
    {
        Descricao = tema.Descricao
    });
    _contexto.SaveChanges();
}
```

O trecho de código chama o contexto da aplicação e cria um construtor de `TemaModelo`, todos os parâmetros da dto `NovoTemaDTO`, são passados para o modelo. e ao final o contexto salva as mudanças no banco.

## Método AtualizarTema (#region Métodos):

Este método tem a responsabilidade de atualizar um tema existente no banco de dados e sua implementação se da na classe `TemaRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public void AtualizarTema(AtualizarTemaDTO tema)
{
    var temaExistente = PegarTemaPeloId(tema.Id);
    temaExistente.Descricao = tema.Descricao;
    _contexto.Temas.Update(temaExistente);
    _contexto.SaveChanges();
}
```

O método `PegarTemaPeloId` recebe um parâmetro de `tema.Id` passado pelo dto `AtualizarTemaDTO`, o retorno passado pela método é um modelo de tema existente no banco de dados. Com esse objeto existente em mãos todos os parâmetros da dto `AtualizarTemaDTO`, são passados para o modelo. Para enviar a ação é necessário chamar o contexto de tema e acionar o método para atualizar, isso é visto no trecho `_contexto.Temas.Update(temaExistente);`, passando como parâmetro o modelo alterado.

## Método DeletarTema (#region Métodos):

Este método tem a responsabilidade de deletar um tema existente no banco de dados através do Id passado como parâmetro e sua implementação se da na classe `TemaRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```

public void DeletarTema(int id)
{
    _contexto.Temas.Remove(PegarTemaPeId(id));
    _contexto.SaveChanges();
}

```

O método `PegarTemaPeId` recebe um parâmetro de `tema.Id` passado como parâmetro, o retorno passado pela método é um modelo de tema existente no banco de dados. Com esse objeto existente em mãos é passado para o contexto de tema em seu método `Remove`, ao final para finalizar a transação é chamado o contexto novamente e finalizado as mudanças.

## 4. Implementando métodos PostagemRepositorio

A classe `PostagemRepositorio` deve ser implementada em um arquivo chamado *PostagemRepositorio.cs* dentro da pasta de (`src/repositorios/implementacoes`). A estrutura da classe deve estar da seguinte maneira:

```

using System.Collections.Generic;
using System.Linq;
using BlogPessoal.src.data;
using BlogPessoal.src.dtos;
using BlogPessoal.src.modelos;
using Microsoft.EntityFrameworkCore;

namespace BlogPessoal.src.repositorios.implementacoes
{
    public class PostagemRepositorio : IPostagem
    {
        #region Atributos

        private readonly BlogPessoalContexto _contexto;

        #endregion Atributos

        #region Construtores

        public PostagemRepositorio(BlogPessoalContexto contexto)
        {
            _contexto = contexto;
        }

        #endregion Construtores

        #region Métodos

        #endregion Métodos
    }
}

```

Em C# é possível definir `#region` onde podemos delimitar aonde irá cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificar de fechá-la com `#endregion`. Na `#region Atributos` foi definido a variável `_contexto` do tipo `BlogPessoalContexto`, que carrega o contexto de dados para ser utilizado pela classe. Esse



contexto apenas é possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve a injeção de dependências do asp.net.

## Método PegarTodasPostagens (#region Métodos):

Este método tem a responsabilidade de buscar uma lista de postagens no banco de dados utilizando. Sua implementação se da na classe `PostagemRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public List<PostagemModelo> PegarTodasPostagens()
{
    return _contexto.Postagens.ToList();
}
```

O trecho de código acima utiliza o contexto de postagem para pesquisar uma lista de postagens.

## Método PegarPostagemPeloid (#region Métodos):

Este método tem a responsabilidade de buscar um único registro de postagem no banco de dados utilizando o Id passado como parâmetro. Sua implementação se da na classe `PostagemRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public PostagemModelo PegarPostagemPeloid(int id)
{
    return _contexto.Postagens.FirstOrDefault(u => u.Id == id);
}
```

O trecho de código acima utiliza o contexto de postagem para pesquisar o primeiro elemento incidente na pesquisa com o Id igual ao passado como parâmetro da função.

## Método PegarPostagensPorPesquisa (#region Métodos):

Este método tem a responsabilidade de buscar uma lista de postagens no banco de dados utilizando trêes parâmetros possíveis na pesquisa. Sua implementação se da na classe `PostagemRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public List<PostagemModelo> PegarPostagensPorPesquisa(
    string titulo,
    string descricaoTema,
    string nomeCriador)
{
    switch (titulo, descricaoTema, nomeCriador)
    {
        case (null, null, null):
            return PegarTodasPostagens();

        case (null, null, _):
            return _contexto.Postagens
                .Include(p => p.Tema)
                .Include(p => p.Criador)
                .where(p => p.Criador.Nome.Contains(nomeCriador))
    }
}
```

```

        .ToList();

    case (null, _, null):
        return _contexto.Postagens
            .Include(p => p.Tema)
            .Include(p => p.Criador)
            .Where(p => p.Tema.Descricao.Contains(descricaoTema))
            .ToList();

    case (_, null, null):
        return _contexto.Postagens
            .Include(p => p.Tema)
            .Include(p => p.Criador)
            .Where(p => p.Titulo.Contains(titulo))
            .ToList();

    case (_, _, null):
        return _contexto.Postagens
            .Include(p => p.Tema)
            .Include(p => p.Criador)
            .Where(p =>
                p.Titulo.Contains(titulo) &
                p.Tema.Descricao.Contains(descricaoTema))
            .ToList();

    case (null, _, _):
        return _contexto.Postagens
            .Include(p => p.Tema)
            .Include(p => p.Criador)
            .Where(p =>
                p.Tema.Descricao.Contains(descricaoTema) &
                p.Criador.Nome.Contains(nomeCriador))
            .ToList();

    case (_, null, _):
        return _contexto.Postagens
            .Include(p => p.Tema)
            .Include(p => p.Criador)
            .Where(p =>
                p.Titulo.Contains(titulo) &
                p.Criador.Nome.Contains(nomeCriador))
            .ToList();

    case (_, _, _):
        return _contexto.Postagens
            .Include(p => p.Tema)
            .Include(p => p.Criador)
            .Where(p =>
                p.Titulo.Contains(titulo) |
                p.Tema.Descricao.Contains(descricaoTema) |
                p.Criador.Nome.Contains(nomeCriador))
            .ToList();
    }
}

```

O trecho definido acima é uma pesquisa que simula 3 filtros ao mesmo tempo, um para título, outro para descrição de tema, e outro para nome do criador. Esse tipo de filtro é fundamental para modelos de negócio como E-commerce, onde se vê necessário a utilização de refinamento de pesquisa. Notasse que este código está desenvolvido dentro de um `switch`, com diferentes casos de pesquisa. O contexto de postagem é chamado e o mesmo inclui o tema e o criador, com isso ao realizar uma postagem é possível ver seu criador e seu tema, é como se estivéssemos fazendo um INNER JOIN com SQL.

## Método NovaPostagem (#region Métodos):

Este método tem a responsabilidade de criar uma postagem no banco de dados e sua implementação se dá na classe `PostagemRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```
public void NovaPostagem (NovaPostagemDTO postagem)
{
    _contexto.Postagens.Add(new PostagemModelo
    {
        Titulo = postagem.Titulo,
        Descricao = postagem.Descricao,
        Foto = postagem.Foto,
        Criador = _contexto.Usuarios.FirstOrDefault(
            u => u.Email == postagem.EmailCriador),
        Tema = _contexto.Temas.FirstOrDefault(
            t => t.Descricao == postagem.DescricaoTema)
    });
    _contexto.SaveChanges();
}
```

O trecho de código chama o contexto da aplicação e cria um construtor de `PostagemModelo`, todos os parâmetros da dto `NovaPostagemDTO`, são passados para o modelo e ao final o contexto salva as mudanças no banco. Para definir o Criador é necessário pesquisar por e-mail utilizando o contexto de usuário, a pesquisa irá retornar um objeto `UsuarioModel` para definir o Criador. O mesmo ocorre com ao definir o tema da postagem, uma pesquisa é feita pela descrição, o retorno devolve um objeto completo de tema que define postagem.

## Método AtualizarPostagem (#region Métodos):

Este método tem a responsabilidade de atualizar uma postagem existente no banco de dados e sua implementação se dá na classe `PostagemRepositorio` dentro do namespace `BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```

public void AtualizarPostagem(AtualizarPostagemDTO postagem)
{
    var postagemExistente = PegarPostagemPeloId(postagem.Id);
    postagemExistente.Titulo = postagem.Titulo;
    postagemExistente.Descricao = postagem.Descricao;
    postagemExistente.Foto = postagem.Foto;
    postagemExistente.Tema = _contexto.Temas.FirstOrDefault(
        t => t.Descricao == postagem.DescricaoTema);
    _contexto.Postagens.Update(postagemExistente);
    _contexto.SaveChanges();
}

```

O método `PegarPostagemPeloId` recebe um parâmetro de `postagem.Id` passado pelo dto `AtualizarPostagemDTO`, o retorno passado pela método é um modelo de postagem existente no banco de dados. Com esse objeto existente em mãos todos os parâmetros da dto `AtualizarPostagemDTO`, são passados para o modelo. Para enviar a ação é necessário chamar o contexto de postagem e acionar o método para atualizar, isso é visto no trecho `_contexto.Postagens.Update(postagemExistente);`, passando como parâmetro o modelo alterado. Vale ressaltar também que o atributo tema é definido pela pesquisa utilizando o contexto de tema para encontrar o objeto inteiro.

## Método DeletarPostagem (#region Métodos):

Este método tem a responsabilidade de deletar uma postagem existente no banco de dados através do Id passado como parâmetro e sua implementação se dá na classe `PostagemRepositorio` dentro do `namespace BlogPessoal.src.repositorios.implementacoes`. Segue o trecho de código que define o método:

```

public void DeletarPostagem(int id)
{
    _contexto.Postagens.Remove(PegarPostagemPeloId(id));
    _contexto.SaveChanges();
}

```

O método `PegarPostagemPeloId` recebe um parâmetro de `postagem.Id` passado como parâmetro, o retorno passado pela método é um modelo de postagem existente no banco de dados. Com esse objeto existente em mãos é passado para o contexto de postagem em seu método `Remove`, ao final para finalizar a transação é chamado o contexto novamente e finalizado as mudanças.

## 5. Adicionando escopo de interfaces repositórios

Após implementar as *interfaces* é necessário criar o escopo para futuras injeções de dependência. Para essa ação é necessário acessar o documento `Startup.cs` e adicionar o trecho de código abaixo:

```

// Repositorios
services.AddScoped<IUsuario, UsuarioRepositorio>();
services.AddScoped<ITema, TemaRepositorio>();
services.AddScoped<IPostagem, PostagemRepositorio>();

```

Seu código irá ficar assim no método `ConfigureServices` :

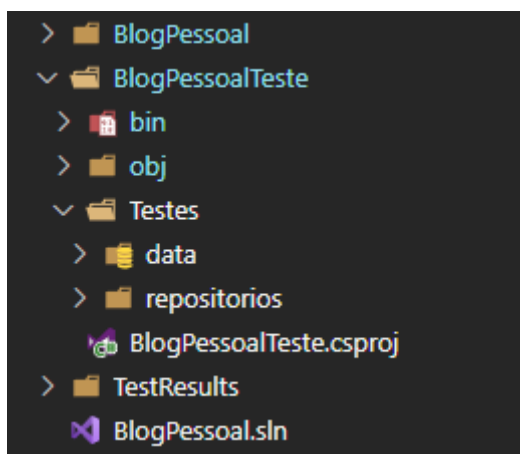
```
public void ConfigureServices(IServiceCollection services)
{
    // Contexto
    IConfigurationRoot config = new ConfigurationBuilder()
        .SetBasePath(AppDomain.CurrentDomain.BaseDirectory)
        .AddJsonFile("appsettings.json")
        .Build();

    services.AddDbContext<AppBlogContext>(
        opt => opt.
            UseSqlServer(config.GetConnectionString("DefaultConnection")));

    // Repositorios
    services.AddScoped<IUsuario, UsuarioRepositorio>();
    services.AddScoped<ITema, TemaRepositorio>();
    services.AddScoped<IPostagem, PostagemRepositorio>();
}
```

## 6. Testes unitário para Repositório

Para o teste unitário criar garantir que o diretório de testes esteja da seguinte maneira:



### Teste unitário UsuarioRepositorio:

Dentro da pasta repositórios criar teste abaixo para validar se o repositório de usuário está funcionando corretamente

```
using System.Linq;
using BlogPessoal.src.data;
using BlogPessoal.src.dtos;
using BlogPessoal.src.repositorios;
using BlogPessoal.src.repositorios.implementacoes;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BlogPessoalTest.Testes.repositorios
{
    [TestClass]
    public class UsuarioRepositorioTeste
    {

```

```

private BlogPessoalContexto _contexto;
private IUsuario _repositorio;

[TestInitialize]
public void ConfiguracaoInicial()
{
    var opt= new DbContextOptionsBuilder<BlogPessoalContexto>()
        .UseInMemoryDatabase(databaseName: "db_blogpessoal")
        .Options;

    _contexto = new BlogPessoalContexto(opt);

    _repositorio = new UsuarioRepositorio(_contexto);
}

[TestMethod]
public void CriarQuatroUsuariosNoBancoRetornaQuatroUsuarios()
{
    //GIVEN - Dado que registro 4 usuarios no banco
    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(
            "Gustavo Boaz",
            "gustavo@email.com",
            "134652",
            "URLFOTO"));

    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(
            "Mallu Boaz",
            "mallu@email.com",
            "134652",
            "URLFOTO"));

    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(
            "Catarina Boaz",
            "catarina@email.com",
            "134652",
            "URLFOTO"));

    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(
            "Pamela Boaz",
            "pamela@email.com",
            "134652",
            "URLFOTO"));

    //WHEN - Quando pesquiso lista total
    //THEN - Então recebo 4 usuarios
    Assert.AreEqual(4, _contexto.Usuarios.Count());
}

[TestMethod]
public void PegarUsuarioPeloEmailRetornaNaoNulo()
{
    //GIVEN - Dado que registro um usuario no banco
    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(

```

```

        "Zenildo Boaz",
        "zenildo@email.com",
        "134652",
        "URLFOTO"));

//WHEN - Quando pesquiso pelo email deste usuario
var user = _repositorio.PegarUsuarioPeloEmail("zenildo@email.com");

//THEN - Então obtenho um usuario
Assert.IsNotNull(user);
}

[TestMethod]
public void PegarUsuarioPeloIdRetornaNaonuloENomeDoUsuario()
{
    //GIVEN - Dado que registro um usuario no banco
    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(
            "Neusa Boaz",
            "neusa@email.com",
            "134652",
            "URLFOTO"));

    //WHEN - Quando pesquiso pelo id 6
    var user = _repositorio.PegarUsuarioPeloId(6);

    //THEN - Então, deve me retornar um elemento não nulo
    Assert.IsNotNull(user);
    //THEN - Então, o elemento deve ser Neusa Boaz
    Assert.AreEqual("Neusa Boaz", user.Nome);
}

[TestMethod]
public void AtualizarUsuarioRetornaUsuarioAtualizado()
{
    //GIVEN - Dado que registro um usuario no banco
    _repositorio.NovoUsuario(
        new NovoUsuarioDTO(
            "Estefânia Boaz",
            "estefania@email.com",
            "134652",
            "URLFOTO"));

    //WHEN - Quando atualizamos o usuario
    var antigo =
        _repositorio.PegarUsuarioPeloEmail("estefania@email.com");
    _repositorio.AtualizarUsuario(
        new AtualizarUsuarioDTO(
            7,
            "Estefânia Moura",
            "123456",
            "URLFOTONOVA"));

    //THEN - Então, quando validamos pesquisa deve retornar nome
    Estefânia Moura
    Assert.AreEqual(
        "Estefânia Moura",
        _contexto.Usuarios.FirstOrDefault(u => u.Id == antigo.Id).Nome);
}

```

```
        //THEN - Então, quando validamos pesquisa deve retornar senha 123456
        Assert.AreEqual(
            "123456",
            _contexto.Usuarios.FirstOrDefault(u => u.Id ==
antigo.Id).Senha);
    }

}

}
```