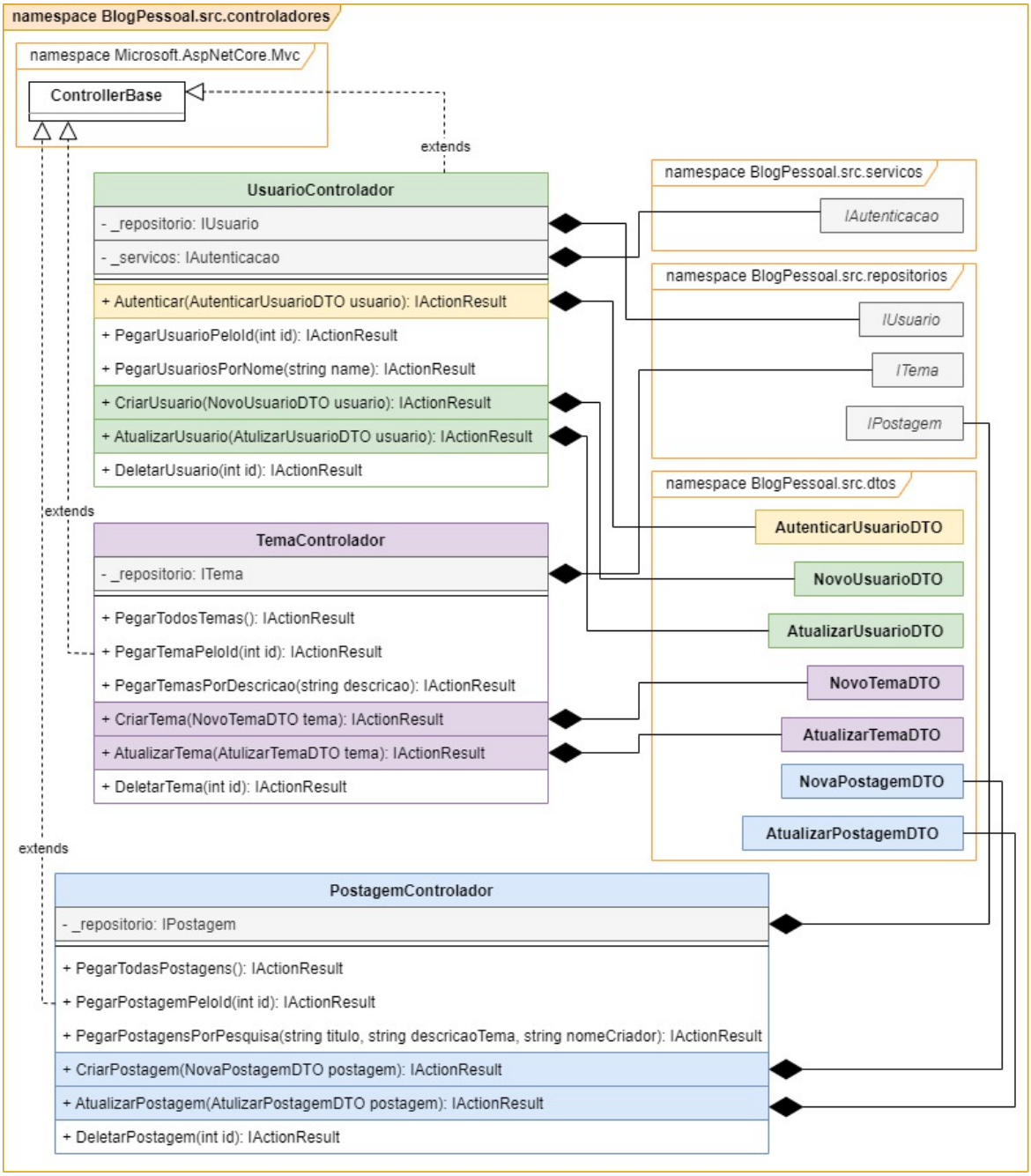
# ASPNET - 4 - Blog Pessoal - Criar controladores

1. Definição da camada controladora
2. Implementando métodos UsuarioControlador
3. Implementando métodos TemaControlador
4. Implementando métodos PostagemControlador
5. Adicionando escopo de controladores
6. Testes de integração com Postman

## 1. Definição da camada controladora

Em sua e definição na arquitetura MVC, um controlador é responsável por manipular os eventos que acontecem entre o cliente e o servidor. O mesmo é responsável de administrar solicitações de requisições e respostas. Em sua implementação é prescindível estender a classe `ControllerBase`, da biblioteca `using Microsoft.AspNetCore.Mvc;`. Abaixo segue o exemplo do diagrama de classe que utilizaremos para escrever as classes `UsuarioControlador`, `TemaControlador` e `PostagemControlador`. Para desenvolve esse processo é necessario que esteja definida e implementada pelo repositório as interfaces `IUsuario`, `ITema` e `IPostagem`.




Neste diagrama é possível notar que a classe controladora conhece vários elementos do nosso sistema, como `dtos` e `repositorios`. Isso se deve pelo fato de que a mesma ira intermediar a transação de informações, então deve esperar dados de entrada do usuário com `dtos`, e com isso pegar recursos no banco através de nosso repositório.

## 2. Implementando métodos UsuarioControlador

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

```
using BlogPessoal.src.dtos;
using BlogPessoal.src.repositorios;
using Microsoft.AspNetCore.Mvc;

namespace BlogPessoal.src.controladores
{
    [ApiController]
    [Route("api/Usuarios")]
```

```
    [Produces("application/json")]
    public class UsuarioControlador : ControllerBase
    {
        #region Atributos

        private readonly IUsuario _repositorio;

        #endregion


        #region Construtores

        public UsuarioControlador(IUsuario repositorio)
        {
            _repositorio = repositorio;
        }

        #endregion


        #region Métodos

        #endregion
    }
}
```

É possível notar que a classe é datada com algumas notações:

| Notação | Definição |
| --- | --- |
| `[ApiController]` | Informa para o sistema que a classe notada é um controlador |
| `[Route("api/Usuarios")]` | Define um *endpoint* de entrada para o controlador; |
| `[Produces("application/json")]` | Define o tipo de saída que o controlador irá fornecer para o cliente que solicitar seus recursos. |

Em C# é possível definir `#region` onde podemos delimitar aonde ira cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificar de fecha-lá com `#endregion`. Na `#region Atributos` foi definido a variável `_repositorio` do tipo `IUsuario`, que carrega o repositório com acesso aos dados para ser utilizado pelos classe. Esse repositório é apenas possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve a injeção de dependências do asp.net . Para implementação dos métodos é necessário implementar na `#region Métodos`.

## Método PegarUsuarioPeloId (#region Métodos):

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

```
[HttpGet("id/{idUsuario}")]
public IActionResult PegarUsuarioPeloId([FromRoute] int idUsuario)
{
    var usuario = _repositorio.PegarUsuarioPeloId(idUsuario);

    if (usuario == null) return NotFound();

    return Ok(usuario);
}
```

O trecho de código acima utiliza o repositório de usuário para pesquisar o primeiro elemento incidente na pesquisa com o id igual ao passado como parâmetro da função. Também é analisado se o retorno de `usuario` é `null`, caso confirmado retorna um *status* `NotFound`, caso contrario retorna um *status* `Ok`, com `usuario` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpGet("id/{idUsuario}")]` | Notação responsavel por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `id/` passando como parâmetro de rota um `idUsuario`; |
| `[FromRoute]` | Responsável por pegar o `idUsuario` passado pela rota e jogar dentro do método. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(usuario)` | Responsável de passar status 200 com um objeto usuário na transferência; |
| `NotFound()` | Responsável de passar status 404 indicando que rota com usuário não foi encontrada no servidor. |

## Método PegarUsuariosPeloNome (#region Métodos):

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

```
[HttpGet]
public IActionResult PegarUsuariosPeloNome([FromQuery] string nomeUsuario)
{
    var usuarios = _repositorio.PegarUsuariosPeloNome(nomeUsuario);

    if (usuarios.Count < 1) return NoContent();

    return Ok(usuarios);
}
```

O trecho de código acima utiliza o repositório de usuário para pesquisar lista de usuários que possuam o nome definido pelo parâmetro `nomeUsuario`. Também é analisado se o retorno de `usuarios` é menor que 1, caso confirmado retorna um *status* `NoContent`, caso contrario retorna um *status* `Ok`, com `usuarios` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
|---|---|
| `[HttpGet]` | Notação responsavel por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `/`; |
| `[FromQuery]` | Responsável por pegar o `nomeUsuario` passado por parâmetro da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
|---|---|
| `Ok(usuarios)` | Responsável de passar status 200 com lista de usuário na transferência; |
| `NoContent()` | Responsável de passar status 204 indicando que a solicitação foi atendida porem não possui nada na transferência. |

## Método PegarUsuarioPeloEmail (#region Métodos):

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

```
[HttpGet("email/{emailUsuario}")]
public IActionResult PegarUsuarioPeloEmail([FromRoute] string emailUsuario)
{
    var usuario = _repositorio.PegarUsuarioPeloEmail(emailUsuario);

    if (usuario == null) return NotFound();

    return Ok(usuario);
}
```

O trecho de código acima utiliza o repositório de usuário para pesquisar o primeiro elemento incidente na pesquisa com o e-mail igual ao passado como parâmetro da função. Também é analisado se o retorno de `usuario` é `null`, caso confirmado retorna um *status* `NotFound`, caso contrario retorna um *status* `Ok`, com `usuario` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpGet("email/{emailUsuario}")]` | Notação responsavel por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `email/` passando como parâmetro de rota um `emailUsuario`; |
| `[FromRoute]` | Responsável por pegar o `emailUsuario` passado pela rota e jogar dentro do método. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(usuario)` | Responsável de passar status 200 com um objeto usuário na transferência; |
| `NotFound()` | Responsável de passar status 404 indicando que rota com usuário não foi encontrada no servidor. |

## Método NovoUsuario (#region Métodos):

Este método tem a responsabilidade de criar um novo usuário no banco recebendo como parâmetro de função um objeto `NovoUsuarioDTO`. Sua implementação se da na classe `UsuarioControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpPost]
public IActionResult NovoUsuario([FromBody] NovoUsuarioDTO usuario)
{
    if(!ModelState.IsValid) return BadRequest();

    _repositorio.NovoUsuario(usuario);
    return Created($"api/Usuarios/{usuario.Email}", usuario);
}
```

O trecho de código acima utiliza o repositório de usuário para salvar um novo usuário definido dentro de uma `dto` passada como parâmetro. Também é analisado se a `dto` contem algum erro, caso confirmado retorna um *status* `BadRequest`, caso contrario retorna um *status* `Created`, com rota de acesso e `usuario` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpPost]` | Notação responsável por informar que o método será acessado por uma requisição do tipo POST para o *endpoint* `/`; |
| `[FromBody]` | Responsável por pegar o `NovoUsuarioDTO` passado pelo corpo da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Created("",usuario)` | Responsável de passar status 201 com usuário na transferência; |
| `BadRequest()` | Responsável de passar status 400 indicando que a solicitação não foi atendida por erro na requisição do cliente. |

## Método AtualizarUsuario (#region Métodos):

Este método tem a responsabilidade de atualizar um usuário já existente no banco recebendo como parâmetro de função um objeto `AtualizarUsuarioDTO`. Sua implementação se da na classe `UsuarioControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpPut]
public IActionResult AtualizarUsuario([FromBody] AtualizarUsuarioDTO usuario)
{
    if(!ModelState.IsValid) return BadRequest();

    _repositorio.AtualizarUsuario(usuario);
    return Ok(usuario);
}
```

O trecho de código acima utiliza o repositório de usuário para atualizar um usuário definido por uma `dto` passada como parâmetro. Também é analisado se a `dto` contem algum erro, caso confirmado retorna um *status* `BadRequest`, caso contrario retorna um *status* `Ok`, com `usuario` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpPut]` | Notação responsável por informar que o método será acessado por uma requisição do tipo PUT para o *endpoint* `/`; |
| `[FromBody]` | Responsável por pegar o `AtualizarUsuarioDTO` passado pelo corpo da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(usuario)` | Responsável de passar status 200 com usuário na transferência; |
| `BadRequest()` | Responsável de passar status 400 indicando que a solicitação não foi atendida por erro na requisição do cliente. |

### Método DeletarUsuario (#region Métodos):

Este método tem a responsabilidade de deletar um único registro de usuário do banco de dados utilizando o id de usuário passado como parâmetro. Sua implementação se da na classe `UsuarioControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpDelete("deletar/{idUsuario}")]
public IActionResult DeletarUsuario([FromRoute] int idUsuario)
{
    _repositorio.DeletarUsuario(idUsuario);
    return NoContent();
}
```

O trecho de código acima utiliza o repositório de usuário para pesquisar o primeiro elemento incidente na pesquisa com o id igual ao passado como parâmetro da função. Retorna um *status* `NoContent`.

#### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpDelete("deletar/{idUsuario}")]` | Notação responsável por informar que o método será acessado por uma requisição do tipo DELETE para o *endpoint* `deletar/` passando como parâmetro de rota um `idUsuario`; |
| `[FromRoute]` | Responsável por pegar o `idUsuario` passado pela rota e jogar dentro do método. |

#### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `NoContent()` | Responsável de passar status 204 informando que transação foi bem sucedida porem sem retorno de transferência; |

## 3. Implementando métodos TemaControlador

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

```
using BlogPessoal.src.dtos;
using BlogPessoal.src.repositorios;
using Microsoft.AspNetCore.Mvc;

namespace BlogPessoal.src.controladores
{
    [ApiController]
```

```
    [Route("api/Temas")]
    [Produces("application/json")]
    public class TemaControlador : ControllerBase
    {
        #region Atributos

        private readonly ITema _repositorio;

        #endregion


        #region Construtores

        public TemaControlador(ITema repositorio)
        {
            _repositorio = repositorio;
        }

        #endregion


        #region Métodos

        #endregion
    }
}
```

É possível notar que a classe é datada com algumas notações:

| Notação | Definição |
| --- | --- |
| `[ApiController]` | Informa para o sistema que a classe notada é um controlador |
| `[Route("api/Temas")]` | Define um *endpoint* de entrada para o controlador; |
| `[Produces("application/json")]` | Define o tipo de saída que o controlador irá fornecer para o cliente que solicitar seus recursos. |

Em C# é possível definir `#region` onde podemos delimitar aonde ira cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificar de fecha-lá com `#endregion`. Na `#region Atributos` foi definido a variável `_repositorio` do tipo `ITema`, que carrega o repositório com acesso aos dados para ser utilizado pelos classe. Esse repositório é apenas possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve a injeção de dependências do asp.net . Para implementação dos métodos é necessário implementar na `#region Métodos`.

## Método PegarTodosTemas (#region Métodos):

Este método tem a responsabilidade de buscar todos registros de tema no banco de dados. Sua implementação se da na classe `TemaControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpGet]
public IActionResult PegarTodosTemas()
{
    var lista = _repositorio.PegarTodosTemas();

    if (lista.Count < 1) return NoContent();

    return Ok(lista);
}
```

O trecho de código acima utiliza o repositório de tema para pesquisar uma lista de temas. Também é analisado se o retorno de `lista` se é seu tamanho é menor que 1, caso confirmado retorna um *status* `NoContent`, caso contrario retorna um *status* `Ok`, com `lista` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpGet]` | Notação responsável por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `/`; |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(lista)` | Responsável de passar status 200 com um lista de temas na transferência; |
| `NoContent()` | Responsável de passar status 204 indicando que não aconteceu erro más que não possui nenhum elemento. |

## Método PegarTemaPeloId (#region Métodos):

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

```
[HttpGet("id/{idTema}")]
public IActionResult PegarTemaPeloId([FromRoute] int idTema)
{
    var tema = _repositorio.PegarTemaPeloId(idTema);

    if (tema == null) return NotFound();

    return Ok(tema);
}
```

O trecho de código acima utiliza o repositório de tema para pesquisar o primeiro elemento incidente na pesquisa com o id igual ao passado como parâmetro da função. Também é analisado se o retorno de `tema` é `null`, caso confirmado retorna um *status* `NotFound`, caso contrario retorna um *status* `Ok`, com `tema` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
|---|---|
| `[HttpGet("id/{idTema}")]` | Notação responsável por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `id/` passando como parâmetro de rota um `idTema`; |
| `[FromRoute]` | Responsável por pegar o `idTema` passado pela rota e jogar dentro do método. |

### Retorno: IActionResult

| Retorno | Definição |
|---|---|
| `Ok(tema)` | Responsável de passar status 200 com um objeto tema na transferência; |
| `NotFound()` | Responsável de passar status 404 indicando que rota com tema não foi encontrada no servidor. |

## Método PegarTemasPelaDescricao (#region Métodos):

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

```
[HttpGet]
public IActionResult PegarTemasPelaDescricao([FromQuery] string descricaoTema)
{
    var temas = _repositorio.PegarTemasPelaDescricao(descricaoTema);

    if (temas.Count < 1) return NoContent();

    return Ok(temas);
}
```

O trecho de código acima utiliza o repositório de tema para pesquisar lista de temas que possuam a descrição pelo parâmetro `descricao`. Também é analisado se o retorno de `temas` é menor que 1, caso confirmado retorna um *status* `NoContent`, caso contrario retorna um *status* `Ok`, com `temas` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpGet]` | Notação responsavel por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `/`; |
| `[FromQuery]` | Responsável por pegar o `descricao` passado por parâmetro da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(temas)` | Responsável de passar status 200 com lista de temas na transferência; |
| `NoContent()` | Responsável de passar status 204 indicando que a solicitação foi atendida porem não possui nada na transferência. |

## Método NovoTema (#region Métodos):

Este método tem a responsabilidade de criar um novo tema no banco recebendo como parâmetro de função um objeto `NovoTemaDTO`. Sua implementação se da na classe `TemaControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpPost]
public IActionResult NovoTema([FromBody] NovoTemaDTO tema)
{
    if(!ModelState.IsValid) return BadRequest();

    _repositorio.NovoTema(tema);
    return Created($"api/Temas/id/{tema.Id}", tema);
}
```

O trecho de código acima utiliza o repositório de tema para salvar um novo tema definido dentro de uma `dto` passada como parâmetro. Também é analisado se a `dto` contem algum erro, caso confirmado retorna um *status* `BadRequest`, caso contrario retorna um *status* `Created`, com rota de acesso e `tema` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpPost]` | Notação responsável por informar que o método será acessado por uma requisição do tipo POST para o *endpoint* `/`; |
| `[FromBody]` | Responsável por pegar o `NovoTemaDTO` passado pelo corpo da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Created("",tema)` | Responsável de passar status 201 com tema na transferência; |
| `BadRequest()` | Responsável de passar status 400 indicando que a solicitação não foi atendida por erro na requisição do cliente. |

## Método AtualizarTema (#region Métodos):

Este método tem a responsabilidade de atualizar um tema já existente no banco recebendo como parâmetro de função um objeto `AtualizarTemaDTO`. Sua implementação se da na classe `TemaControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpPut]
public IActionResult AtualizarTema([FromBody] AtualizarTemaDTO tema)
{
    if(!ModelState.IsValid) return BadRequest();

    _repositorio.AtualizarTema(tema);
    return Ok(tema);
}
```

O trecho de código acima utiliza o repositório de tema para atualizar um tema definido por uma `dto` passada como parâmetro. Também é analisado se a `dto` contem algum erro, caso confirmado retorna um *status* `BadRequest`, caso contrario retorna um *status* `Ok`, com `tema` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpPut]` | Notação responsável por informar que o método será acessado por uma requisição do tipo PUT para o *endpoint* `/`; |
| `[FromBody]` | Responsável por pegar o `AtualizarTemaDTO` passado pelo corpo da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(tema)` | Responsável de passar status 200 com tema na transferência; |
| `BadRequest()` | Responsável de passar status 400 indicando que a solicitação não foi atendida por erro na requisição do cliente. |

### Método DeletarTema (#region Métodos):

Este método tem a responsabilidade de deletar um único registro de tema do banco de dados utilizando o id de tema passado como parâmetro. Sua implementação se da na classe `TemaControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpDelete("deletar/{idTema}")]
public IActionResult DeletarTema([FromRoute] int idTema)
{
    _repositorio.DeletarTema(idTema);
    return NoContent();
}
```

O trecho de código acima utiliza o repositório de tema para pesquisar o primeiro elemento incidente na pesquisa com o id igual ao passado como parâmetro da função. Retorna um *status* `NoContent`.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpDelete("deletar/{idTema}")]` | Notação responsável por informar que o método será acessado por uma requisição do tipo DELETE para o *endpoint* `deletar/` passando como parâmetro de rota um `idTema`; |
| `[FromRoute]` | Responsável por pegar o `idTema` passado pela rota e jogar dentro do método. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `NoContent()` | Responsável de passar status 204 informando que transação foi bem sucedida porem sem retorno de transferência; |

## 4. Implementando métodos PostagemControlador

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

```
using BlogPessoal.src.dtos;
using BlogPessoal.src.repositorios;
using Microsoft.AspNetCore.Mvc;

namespace BlogPessoal.src.controladores
{
    [ApiController]
    [Route("api/Postagens")]
```

```
    [Produces("application/json")]
    public class PostagemControlador : ControllerBase
    {
        #region Atributos

        private readonly IPostagem _repositorio;

        #endregion


        #region Construtores

        public PostagemControlador(IPostagem repositorio)
        {
            _repositorio = repositorio;
        }

        #endregion


        #region Métodos

        #endregion
    }
}
```

É possível notar que a classe é datada com algumas notações:

| Notação | Definição |
| --- | --- |
| `[ApiController]` | Informa para o sistema que a classe notada é um controlador |
| `[Route("api/Postagens")]` | Define um *endpoint* de entrada para o controlador; |
| `[Produces("application/json")]` | Define o tipo de saída que o controlador irá fornecer para o cliente que solicitar seus recursos. |

Em C# é possível definir `#region` onde podemos delimitar aonde ira cada trecho de código digitado. Sempre quando temos uma `#region` aberta, devemos certificar de fecha-lá com `#endregion`. Na `#region Atributos` foi definido a variável `_repositorio` do tipo `IPostagem`, que carrega o repositório com acesso aos dados para ser utilizado pelos classe. Esse repositório é apenas possível ser utilizado quando inicializamos o mesmo através de um construtor, isso se deve a injeção de dependências do asp.net . Para implementação dos métodos é necessário implementar na `#region Métodos`.

## Método PegarPostagemPeloId (#region Métodos):

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

```
[HttpGet("id/{idPostagem}")]
public IActionResult PegarPostagemPeloId([FromRoute] int idPostagem)
{
    var postagem = _repositorio.PegarPostagemPeloId(idPostagem);

    if (postagem == null) return NotFound();

    return Ok(postagem);
}
```

O trecho de código acima utiliza o repositório de postagem para pesquisar o primeiro elemento incidente na pesquisa com o id igual ao passado como parâmetro da função. Também é analisado se o retorno de `postagem` é `null`, caso confirmado retorna um *status* `NotFound`, caso contrario retorna um *status* `Ok`, com `postagem` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
|---|---|
| `[HttpGet("id/{idTPostagem}")]` | Notação responsável por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `id/` passando como parâmetro de rota um `idPostagem`; |
| `[FromRoute]` | Responsável por pegar o `idTema` passado pela rota e jogar dentro do método. |

### Retorno: IActionResult

| Retorno | Definição |
|---|---|
| `Ok(postagem)` | Responsável de passar status 200 com um objeto postagem na transferência; |
| `NotFound()` | Responsável de passar status 404 indicando que rota com postagem não foi encontrada no servidor. |

## Método PegarTodasPostagens (#region Métodos):

Este método tem a responsabilidade de buscar todos registros de postagens no banco de dados. Sua implementação se da na classe `PostagemControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpGet]
public IActionResult PegarTodasPostagens()
{
    var lista = _repositorio.PegarTodasPostagens ();

    if (lista.Count < 1) return NoContent();

    return Ok(lista);
}
```

O trecho de código acima utiliza o repositório de postagem para pesquisar uma lista de postagens. Também é analisado se o retorno de `lista` se seu tamanho é menor que 1, caso confirmado retorna um *status* `NoContent`, caso contrario retorna um *status* `Ok`, com `lista` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpGet]` | Notação responsável por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `/`; |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(lista)` | Responsável de passar status 200 com um lista de postagens na transferência; |
| `NoContent()` | Responsável de passar status 204 indicando que não aconteceu erro más que não possui nenhum elemento. |

## Método PegarPostagensPorPesquisa (#region Métodos):

Este método tem a responsabilidade de buscar uma lista de postagens no banco de dados utilizando titulo, descrição do tema ou nome do criador passados como parâmetro. Sua implementação se da na classe `PostagemControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpGet]
public IActionResult PegarPostagensPorPesquisa(
    [FromQuery] string titulo,
    [FromQuery] string descricaoTema,
    [FromQuery] string nomeCriador)
{
    var postagens = _repositorio.PegarPostagensPorPesquisa(titulo,
descricaoTema, nomeCriador);

    if (postagens.Count < 1) return NoContent();

    return Ok(postagens);
}
```

O trecho de código acima utiliza o repositório de postagem para pesquisar lista de postagens que possuam as características fornecidas dos parâmetros `titulo, descricaoTema e nomeCriador`. Também é analisado se o retorno de `postagens` é menor que 1, caso confirmado retorna um *status* `NoContent`, caso contrario retorna um *status* `Ok`, com `postagens` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpGet]` | Notação responsavel por informar que o método será acessado por uma requisição do tipo GET para o *endpoint* `/`; |
| `[FromQuery]` | Responsável por pegar o `titulo, descricaoTema e nomeCriador` passado por parâmetro da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(postagens)` | Responsável de passar status 200 com lista de postagens na transferência; |
| `NoContent()` | Responsável de passar status 204 indicando que a solicitação foi atendida porem não possui nada na transferência. |

## Método NovaPostagem (#region Métodos):

Este método tem a responsabilidade de criar uma nova postagem no banco recebendo como parâmetro de função um objeto `NovaPostagemDTO`. Sua implementação se da na classe `PostagemControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpPost]
public IActionResult NovaPostagem([FromBody] NovaPostagemDTO postagem)
{
    if(!ModelState.IsValid) return BadRequest();

    _repositorio.NovaPostagem(postagem);
    return Created($"api/Postagens/id/{postagem.Id}", postagem);
}
```

O trecho de código acima utiliza o repositório de postagem para salvar uma nova postagem definida dentro de uma `dto` passada como parâmetro. Também é analisado se a `dto` contem algum erro, caso confirmado retorna um *status* `BadRequest`, caso contrario retorna um *status* `Created`, com rota de acesso e `tema` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpPost]` | Notação responsável por informar que o método será acessado por uma requisição do tipo POST para o *endpoint* `/`; |
| `[FromBody]` | Responsável por pegar o `NovaPostagemDTO` passado pelo corpo da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Created("",postagem)` | Responsável de passar status 201 com postagem na transferência; |
| `BadRequest()` | Responsável de passar status 400 indicando que a solicitação não foi atendida por erro na requisição do cliente. |

## Método AtualizarPostagem (#region Métodos):

Este método tem a responsabilidade de atualizar uma postagem já existente no banco recebendo como parâmetro de função um objeto `AtualizarPostagemDTO`. Sua implementação se da na classe `PostagemControlador` dentro do `namespace BlogPessoal.src.controladores`. Segue o trecho de código que define o método:

```
[HttpPut]
public IActionResult AtualizarPostagem([FromBody] AtualizarPostagemDTO postagem)
{
    if(!ModelState.IsValid) return BadRequest();

    _repositorio.AtualizarPostagem(postagem);
    return Ok(postagem);
}
```

O trecho de código acima utiliza o repositório de postagem para atualizar uma postagem definida por uma `dto` passada como parâmetro. Também é analisado se a `dto` contem algum erro, caso confirmado retorna um *status* `BadRequest`, caso contrario retorna um *status* `Ok`, com `postagem` sendo passado como parâmetro do método.

### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpPut]` | Notação responsável por informar que o método será acessado por uma requisição do tipo PUT para o *endpoint* `/`; |
| `[FromBody]` | Responsável por pegar o `AtualizarPostagemDTO` passado pelo corpo da requisição feita pelo cliente. |

### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `Ok(postagem)` | Responsável de passar status 200 com postagem na transferência; |
| `BadRequest()` | Responsável de passar status 400 indicando que a solicitação não foi atendida por erro na requisição do cliente. |

### Método DeletarPostagem (#region Métodos):

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

```
[HttpDelete("deletar/{idPostagem}")]
public IActionResult DeletarPostagem([FromRoute] int idPostagem)
{
    _repositorio.DeletarPostagem(idPostagem);
    return NoContent();
}
```

O trecho de código acima utiliza o repositório de postagem para pesquisar o primeiro elemento incidente na pesquisa com o id igual ao passado como parâmetro da função. Retorna um *status* `NoContent`.

#### Notações utilizadas:

| Notação | Definição |
| --- | --- |
| `[HttpDelete("deletar/{idPostagem}")]` | Notação responsável por informar que o método será acessado por uma requisição do tipo DELETE para o *endpoint* `deletar/` passando como parâmetro de rota um `idPostagem`; |
| `[FromRoute]` | Responsável por pegar o `idPostagem` passado pela rota e jogar dentro do método. |

#### Retorno: IActionResult

| Retorno | Definição |
| --- | --- |
| `NoContent()` | Responsável de passar status 204 informando que transação foi bem sucedida porem sem retorno de transferência; |

## 5. Adicionando escopo de controladores

Após implementar os *controladores* é necessário criar o escopo para futuras injeções de dependência e utilização do sistema. Para essa ação é necessário acessar o documento *Startup.cs* e adicionar o trecho de código abaixo:

```
// Controladores
services.AddCors();
services.AddControllers();
```

Esse serviço permitira com que sua aplicação possa ser acessada de qualquer aplicação externa através de seus *endpoints*. Seu código ira 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<BlogPessoalCOntexto>(
        opt => opt.
        UseSqlServer(config.GetConnectionString("DefaultConnection")));

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

    // Controladores
    services.AddCors();
    services.AddControllers();
}
```

Também é necessário modificar o método `Configure` localizado no documento *Startup.cs*, adcionar o trecho de código abaixo:

```
// Ambiente de produção
// Rotas
app.UseRouting();

app.UseCors(c => c
            .AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader());

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

Esta configuração inicializa na aplicação suas rotas e *endpoints*, e permite também o acesso externo na API. Seu código ira ficar assim no método `Configure` :

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
BlogPessoalContexto contexto)
{
    // Ambiente de desenvolvimento
    if (env.IsDevelopment())
    {
        contexto.Database.EnsureCreated();
        app.UseDeveloperExceptionPage();
    }

    // Ambiente de produção
    // Rotas
    app.UseRouting();

    app.UseCors(c => c
```

```
            .AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader());

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Desta maneira sua aplicação estará pronta para ser testada!.

# 6. Testes de integração com Postman

## Visão geral de Testes

## Testes Funcionais

São testes que avaliam o comportamento da aplicação (funcionalidades, regras de negócio, etc) e são eles:

1. Unitário:
    - Também conhecidos como testes de unidade. Tem por objetivo testar a menor parte testável do sistema, geralmente um método;
    - Não utilizam banco, mas podem possuir *mocks*. *Mocks* são objetos "falsos" que simulam o comportamento de uma classe ou objeto real.
2. Integração:
    - Tem por objetivo testar a interação entre componentes de software, normalmente duas partes do software. É o processo de verificar se os componentes do sistema, juntos, trabalham conforme esperado;
    - Podem utilizar banco e *mocks*;
    - Costumam testar a integração entre classes, chamadas a *WebServices* (normalmente com *mocks*), entre outras coisas.
3. Sistema:
    - Tem como objetivo garantir que o sistema funciona como um todo. Se no teste de integração, o acesso ao banco, aos serviços web, etc, são testados individualmente, nos testes de sistema o sistema é testado como um todo;
    - São comumente chamados de testes de caixa-preta, pois o sistema é testado com tudo ligado: banco de dados, serviços web, batch jobs, e etc;
    - Usando como exemplo uma aplicação Web, os testes de sistema seriam responsáveis por fazer os passos que um usuário poderia fazer, então um teste de sistema poderia abrir um browser, se logar na aplicação e conferir se tudo foi executado com sucesso, da mesma forma que pode conferir como a tela da aplicação reage caso o usuário digite um dado inválido.
4. Aceitação:
    - Tem como objetivo verificar se o que foi implementado atende corretamente ao que o cliente esperava, ou seja, validar o sistema do ponto de vista do cliente. Normalmente, isso é feito através de testes de sistema.

## Testes não Funcionais

São testes que verificam atributos de um componente de sistema que não se relacionam com a funcionalidade (confiabilidade, eficiência, usabilidade, manutenabilidade e portabilidade). Se dividem em muitos tipos, mas aqui vão os principais ou mais utilizados:

1. Carga:
    - Tem como objetivo testar o sistema simulando múltiplos usuários acessando ao mesmo tempo para ver como o sistema se comporta e aguenta. Está diretamente associado aos próximos dois testes.
2. Performance:
    - Tem como objetivo testar rapidez: tempo de resposta. Pode estar sobre condições que seriam próximas às reais para verificar a performance do sistema.
3. Estresse:
    - Tem como objetivo testar os limites da aplicação. A ideia é sobrecarregar o sistema com muitos usuários para ver até onde o sistema para e como (e se) ele se recupera.

## Ferramentas

- No C#, as ferramentas para testes funcionais costumam ser: **MSTest**, **xUnit** e **NUnit**, utilizado com o **Selenium** (para testes de sistema) e o uso de **Specflow** para de comportamento(BDD);
- Para testes não-funcionais, independente de linguagem, o mais conhecido é o **JMeter**, mas possuem outros como **httperf** (para testes de performance) e **locust** (para todos os tipos).

## Modelo Ideal de téste

- Os testes unitários testando praticamente todos os métodos da aplicação;
- Os testes de integração testando apenas as partes das pequenas integrações;
- Os testes de sistema (e aceitação) testando o sistema no geral.

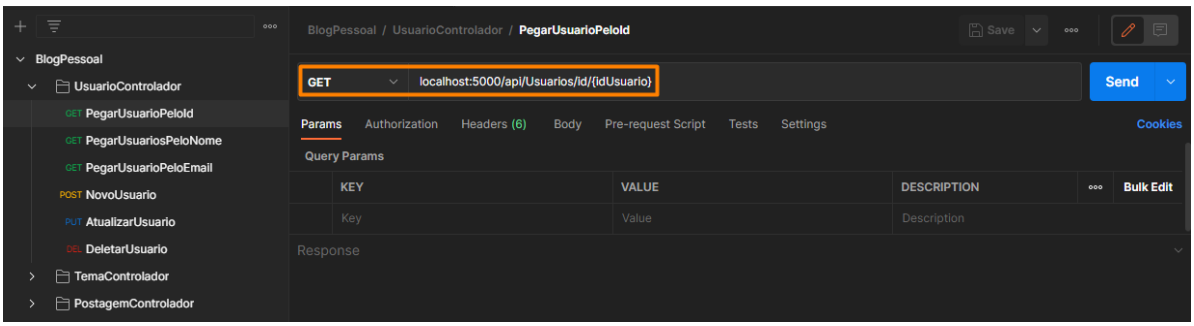
# Testando Controladores com Postman

Para testar com postman, fazer download da ferramenta no Link. Sempre que quiser testar a aplicação é necessário executar o projeto.

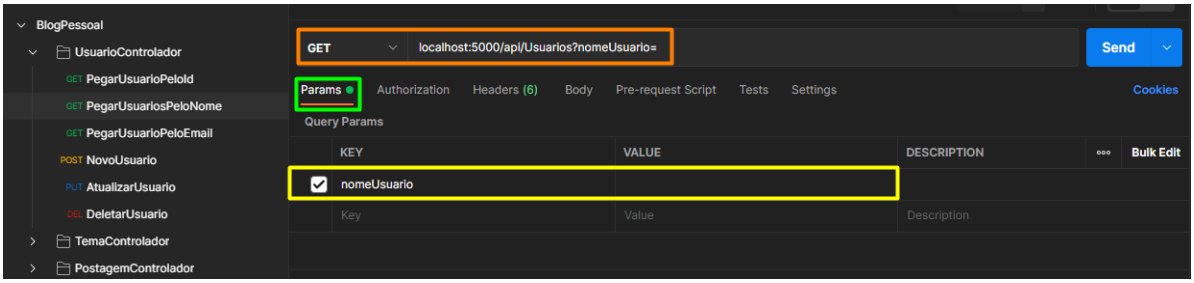
Em seguida abra o Postman e construa a estrutura para os testes de acordo com as imagens abaixo:

## Testes de UsuarioControlador

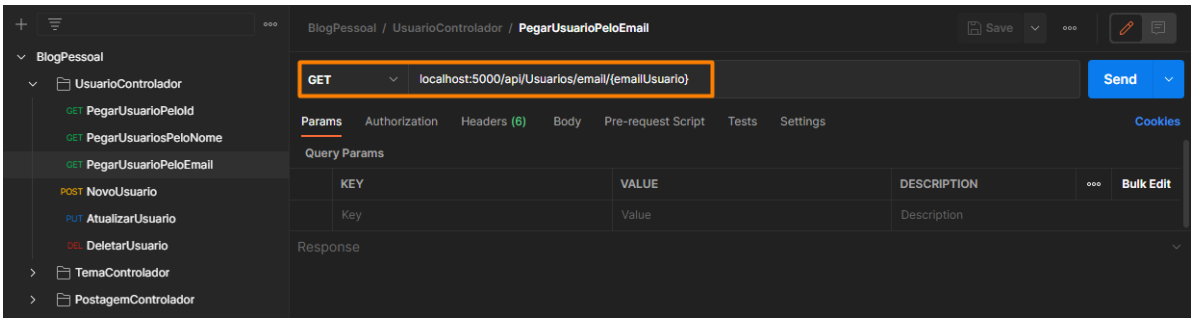
### PegarUsuarioPeloId



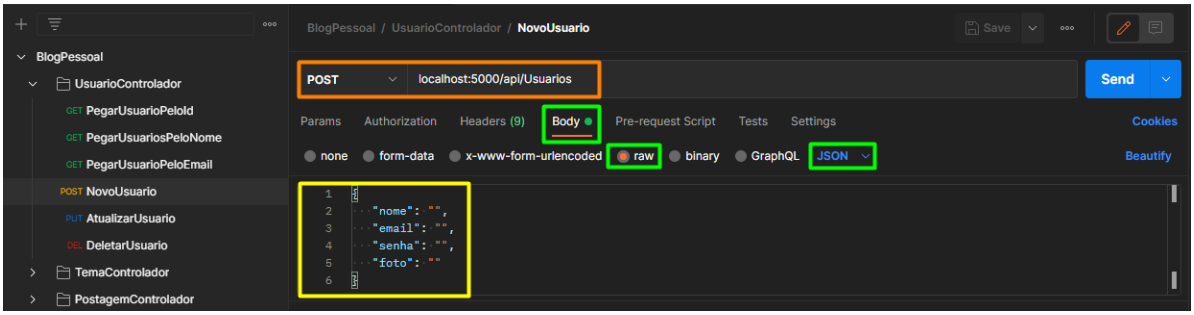

PegarUsuariosPeloNome



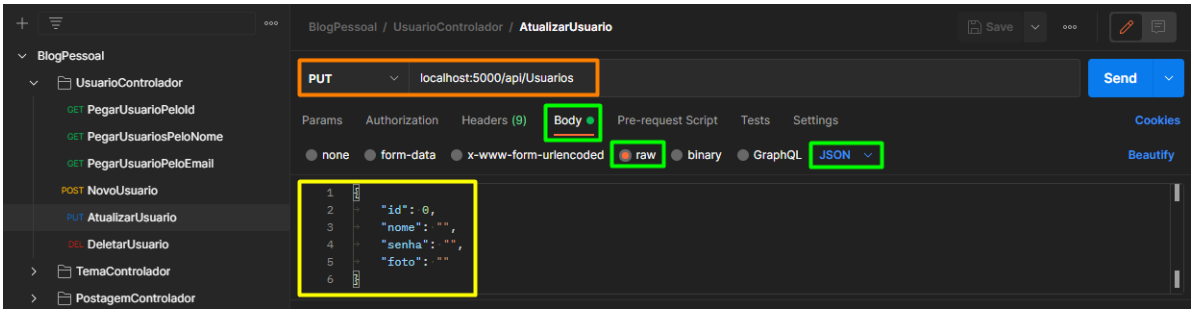

PegarUsuarioPeloEmail



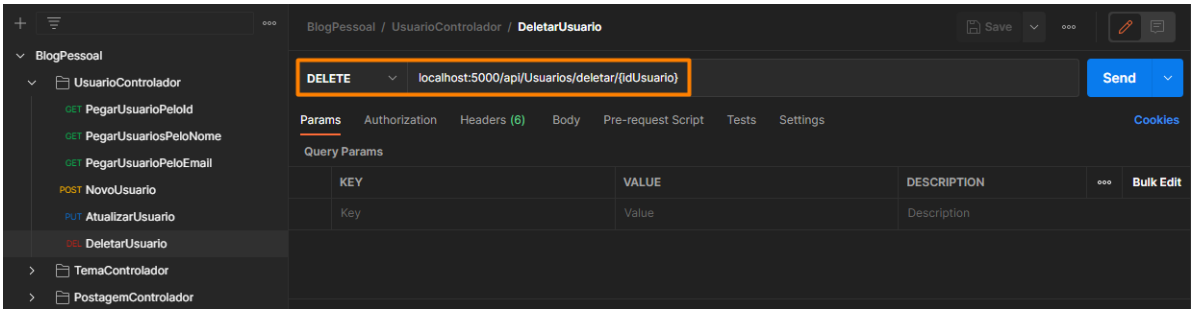

NovoUsuario




AtualizarUsuario

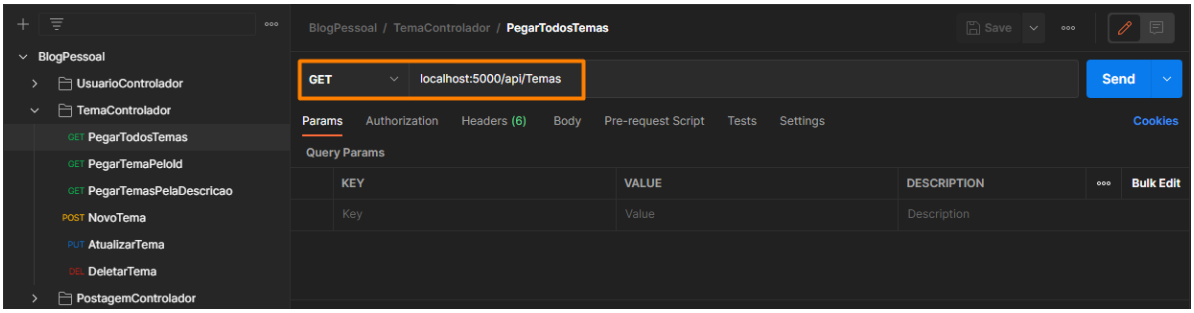



DeletarUsuario

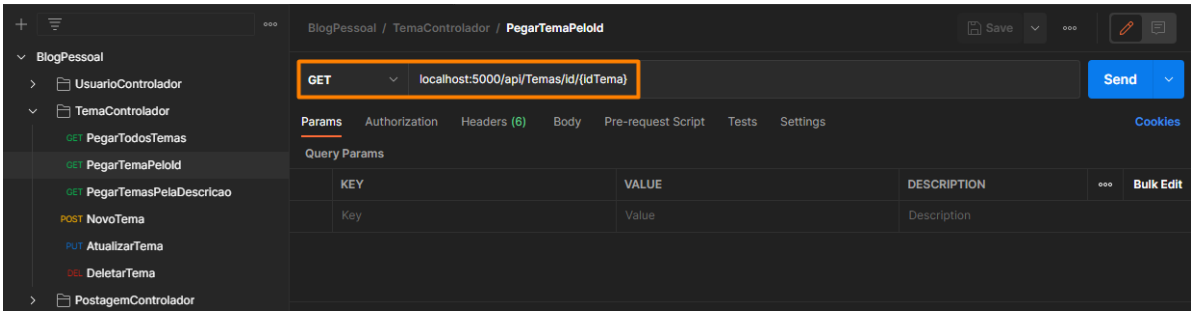



Testes de TemaControlador

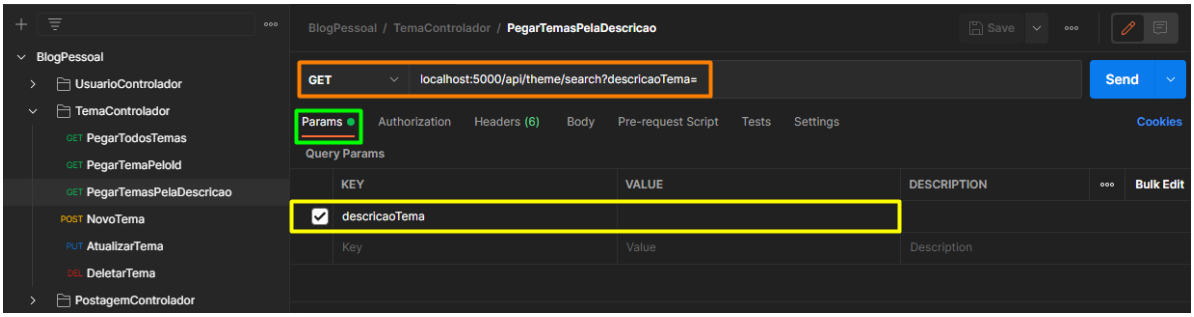
PegarTodosTemas



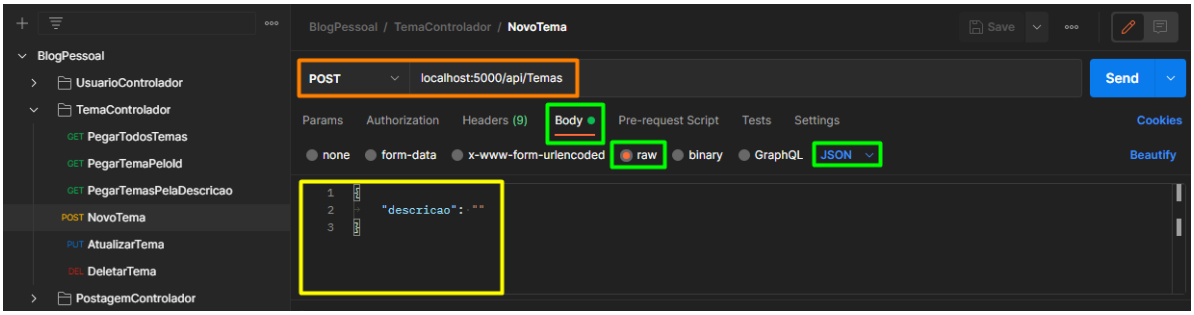

PegarTemaPeloId



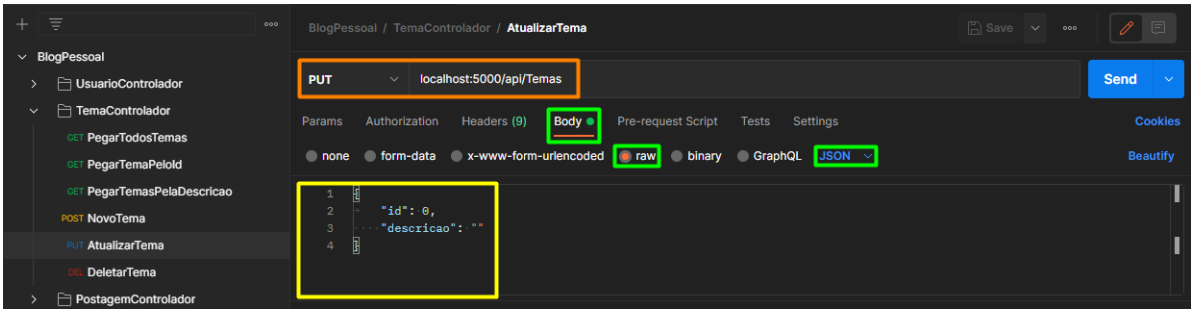

PegarTemasPelaDescricao



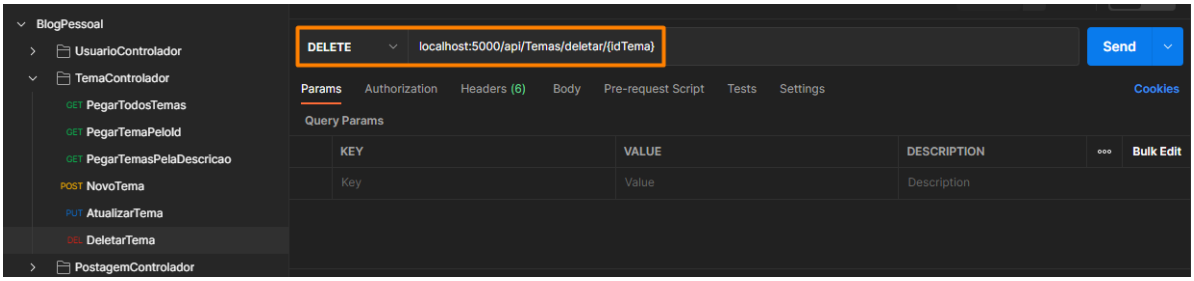

NovoTema




AtualizarTema

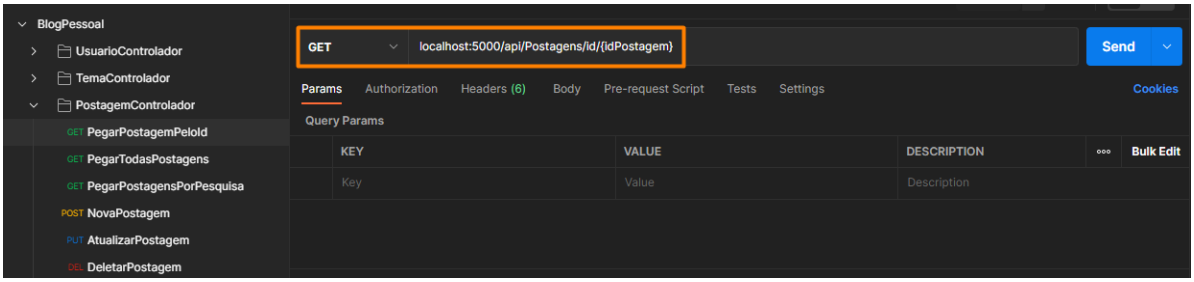



DeletarTema

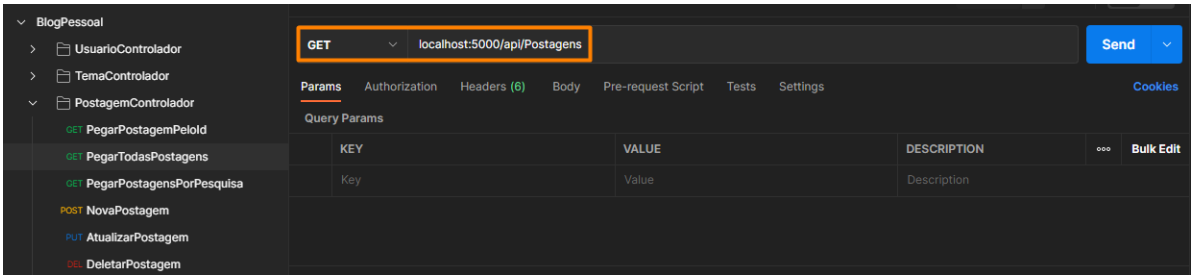



Testes de PostagemControlador

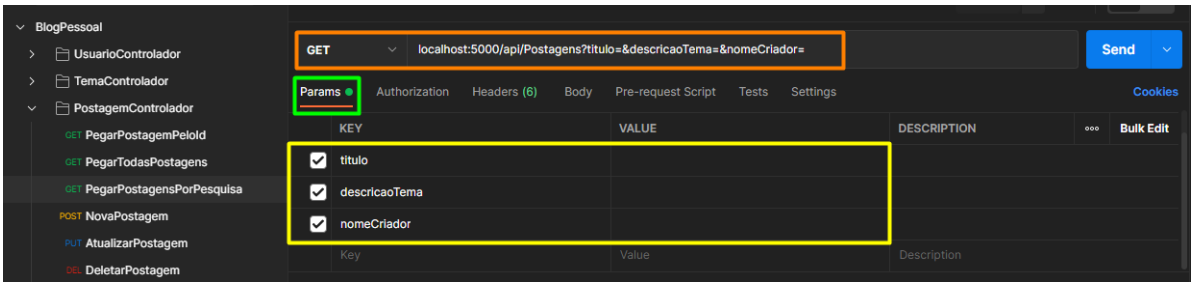
PegarPostagemPesoId




PegarTodasPostagens




PegarPostagensPorPesquisa




NovaPostagem


AtualizarPostagem


DeletarPostagem