ASP .NET Core - Usando o padrão Mediator com MediatR (CQRS)

Neste artigo vou mostrar como podemos implementar o padrão *Mediator* usando o MediatR em uma aplicação ASP .NET Core MVC na versão 3.1.

O padrão *Mediator* é um padrão de projetos comportamental (Gof) que eu já abordei neste artigo: O padrão de projeto Mediator.

A aplicação deste padrão nos ajuda a garantir um baixo acoplamento entre os objetos de nossa aplicação permitindo que um objeto se comunique com outros objetos sem saber nada de sua estrutura e também centraliza o fluxo de comunicação entre os objetos facilitando sua manutenção.

Neste artigo faremos uma implementação deste padrão em uma aplicação ASP .NET Core utilizando a biblioteca MediatR que foi criada por *Jimmy Bogard* e que pode ser obtida via Nuget usando os comandos abaixo na janela do Package Manger Console:

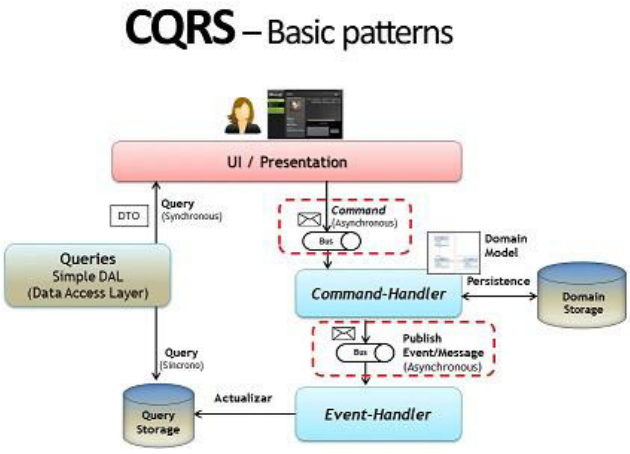
- Install-Package MediatR
- Install-Package MediatR.Extensions.Microsoft.DependencyInjection

Se você estiver usando o VS Code com a linha de comando NET CLI os comandos são:

- dotnet add package MediatR
- dotnet add package MediatR.Extensions.Microsoft.DependencyInjection --version 8.0.0

O primeiro é o pacote do MediatR e o segundo pacote é usado para gerenciar suas dependências.

Ocorre que se em sua aplicação houver um grande fluxo de requisições entre os objetos, o objeto *mediator* pode ser o gargalo da aplicação e para contornar isso é comum realizar a implementação do CQRS - Command Query Responsability Segregation que traduzindo é *Segregação de Responsabilidade de Comando e Consulta.*



O CQRS é mais um padrão de projeto separa as operações de *leitura e de escrita* da base de dados em dois modelos :

1. Queries - São responsáveis pelas leituras ou consultas e retornam objetos *DTOs ou ViewModels* e não alteram o estado dos dados;

2. Commands - São responsáveis pelas ações que realizam alguma alteração na base de dados e não retornam nada. *(operações para incluir, alterar e deletar).* Para gerenciar a comunicação entre os objetos nos *Commands* usamos o Mediator.

Nota: *Vale lembrar que não precisamos do padrão Mediator para implementar o padrão CQRS.*

Como funciona ?

Aqui temos dois componentes principais chamados de Request e Handler, que são implementados usando as interfaces IRequest e IRequestHandler<TRequest> :

- Request → Representa a mensagem a ser processada;
- Handler → Faz o processamento de determinada(s) mensagen(s);

Onde, um Request contém propriedades que são usadas para fazer o input dos dados para os Handlers.

Para que esses dois componentes funcionem precisamos de um mediador que faz o *meio de campo* recebendo um request e invocando o Handler associado a ele.

É aqui que o componente Mediator, que implementa a interface IMediator, entra em cena, e por onde deveremos interagir com as demais classes. Como usamos uma interface (*IMediator*) apenas enviamos o request para o Mediator que vai chamar a classe que vai executar o comando; tudo de forma transparente.

Aplicando isso à ASP .NET Core podemos ter controladores mais enxutos que vão receber os *requests* HTTP e executar o Mediator.

A seguir vamos criar uma aplicação *ASP .NET Core* onde vamos aplicar o padrão Mediator usando a biblioteca MediatR e usar os conceitos de CQRS via *mediator*.

recursos usados:

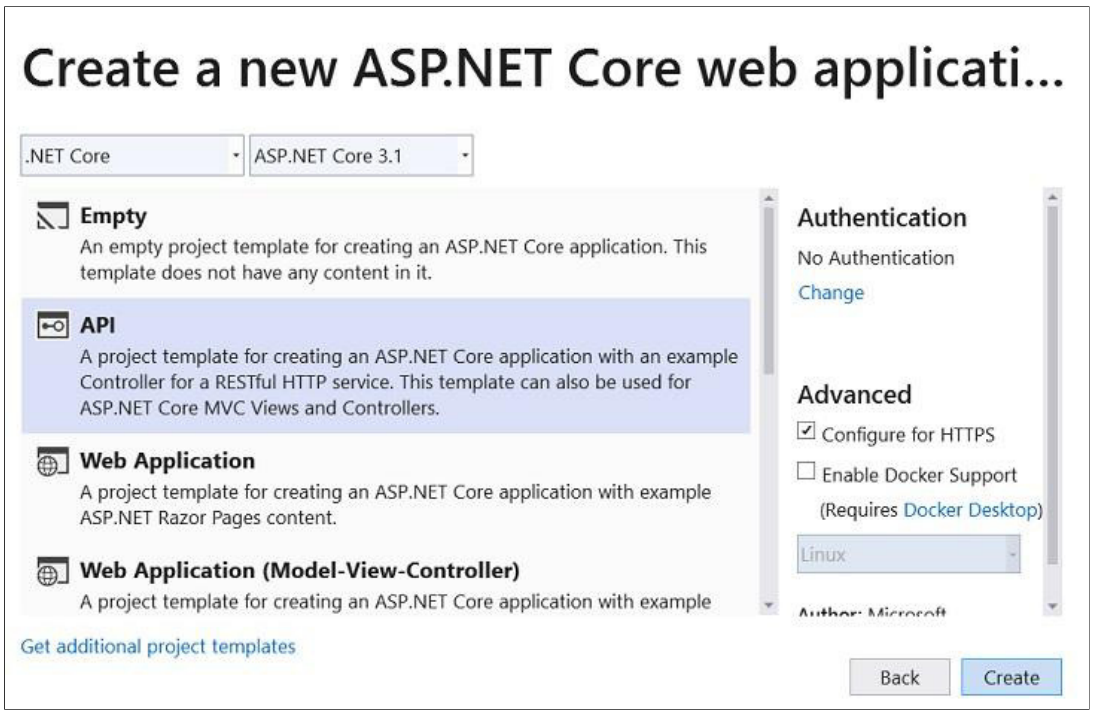
- VS 2019 Community
- ASP .NET Core 3.1
- MediatR

Criando o projeto inicial no VS 2019

Abra o VS 2019 Community e crie um novo projeto via menu File-> New Project;

Selecione o template ASP .NET Core Web Application, e, Informe o nome da solução AspNet_MediatR1 *(ou outro nome a seu gosto)*.

A seguir selecione .NET Core e ASP .NET Core 3.1 e marque o template API e as configurações conforme figura abaixo:



Depois que o projeto foi criado, precisamos adicionar a referência aos seguintes pacotes:

- MediatR 8.0.1
- MediatR.Extensions.Microsoft.DependencyInjection 8.0.0

Nota: *Para instalar use o comando:* ***Install-Package <nome> --version X.X.X***

Definindo o modelo de domínio, os comandos e os handlers

Vamos agora organizar as pastas do Domínio.

Crie uma pasta Domain no projeto e nesta pasta crie as pastas:

- Entity - onde vamos criar a nossa entidade;
- Command - onde criaremos os comandos;
- Handler - onde criamos os *handlers*;

Vamos iniciar criando a nossa entidade Produto na pasta Entity:

```
public class Produto
{
      public int Id { get; set; }
      public string Nome { get; set; }
      public decimal Preco { get; set; }
}
```

A classe Produto representa a nossa entidade de domínio Produto, e, geralmente, possui estado, comportamento e suas regras de negócio (*neste exemplo não foram usadas para tornar mais simples o exemplo*).

Criando os Commands

Agora na pasta Command vamos implementar os comandos relativos às ações que iremos executar. Para isso vamos implementar o padrão Command que define um objeto que encapsula toda a informação para executar uma ação.

É aqui que temos a utilização do CQRS pela implementação do padrão Command composto de dois objetos:

1. Command - Define ações que irão alterar o estado dos dados e os objetos;
2. Command Handler - São responsáveis por executar as ações definidas pelos objetos Command;

1- ProdutoCreateCommand

```
using MediatR

public class ProdutoCreateCommand : IRequest<string>
{
      public string Nome { get; private set; }
      public decimal Preco { get; private set; }
}
```

Note que esta classe implementa a interface IRequest que é a interface do MediatR usada para indicar que esse é um comando usado pelas classes Handlers que iremos criar a seguir.

2- ProdutoDeleteCommand

```
using MediatR

public class ProdutoDeleteCommand : IRequest<string>
{
      public int Id { get; set; }
}
```

3- ProdutoUpdateCommand

```
using MediatR

public class ProdutoUpdateCommand : IRequest<string>
{
      public int Id { get; private set; }
      public string Nome { get; private set; }
      public decimal Preco { get; private set; }
}
```

Todas as classes implementam IRequest<T> onde especificamos o tipo de dados que será retornado quando o comando for processado, e, também, através da qual vinculamos os comandos com as classes Command Handlers. É assim que a MediatR sabe qual objeto deve ser invocado quando um *request* for gerado.

Assim para cada Command vamos criar um Command Handler, embora podemos implementar um único objeto *Command Handler* para tratar todos os Commands criados na aplicação.

Criando o Repositório

Não vamos usar um banco de dados, ao invés disso, vamos criar uma interface IRepository<T> e implementar esta interface para salvar os dados em uma coleção estática usando a classe ProdutoRepository.

Vamos criar uma pasta Repository no projeto e nesta pasta criar a interface e a classe que a implementa:

1- IRepository

```
using System.Collections.Generic;
using System.Threading.Tasks;

public interface IRepository<T>
{
        Task<IEnumerable<T>> GetAll();
        Task<T> Get(int id);
        Task Add(T item);
        Task Edit(T item);
        Task Delete(int id);
}
```

2 - ProdutoRepository

```
using AspNet_MediatR1.Domain.Entity;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

 public class ProdutoRepository : IRepository<Produto>
 {
       private static Dictionary<int, Produto> produtos = new Dictionary<int,Produto>();

       public async Task<IEnumerable<Produto>> GetAll()
       {
          return await Task.Run(() => produtos.Values.ToList());
       }

       public async Task<Produto> Get(int id)
       {
          return await Task.Run(() => produtos.GetValueOrDefault(id));
       }

       public async Task Add(Produto produto)
       {
          await Task.Run(() => produtos.Add(produto.Id, produto));
       }

       public async Task Edit(Produto produto)
       {
          await Task.Run(() =>
          {
             produtos.Remove(produto.Id);
             produtos.Add(produto.Id, produto);
          });
       }

       public async Task Delete(int id)
       {
          await Task.Run(() => produtos.Remove(id));
       }
    }
```

Criando Notificações

As notificações são necessárias para informar que uma requisição foi concluída com sucesso visto que as requisições Command não retornam nenhuma informação.

Para isso podemos invocar o método Publish() no método Handler da classe Command Handler passando por parâmetro uma *notificação*, assim, todos os *Event Handlers* que estiverem "*ouvindo*" as notificações do tipo do objeto "*publicado*" serão notificados e poderão processá-lo.

O método Publish() é o responsável por emitir a notificação em todo sistema, e, ele vai procurar a classe que possui a herança da interface INotificationHandler<T> e invocar o método Handler() para processar aquela notificação.

Para implementar as notificações temos que definir os objetos *notification*.

Vamos criar uma pasta Notifications no projeto e a seguir criar três classes que herdam da interface INotification.

1- ProdutoCreateNotification

```
using MediatR;

namespace AspNet_MediatR1.Notifications
{
    public class ProdutoCreateNotification : INotification
    {
        public int Id { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
    }
}
```

2- ProdutoUpdateNotification

```
using MediatR;

namespace AspNet_MediatR1.Notifications
{
    public class ProdutoUpdateNotification : INotification
    {
        public int Id { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
        public bool IsConcluido { get; set; }
    }
}
```

3- ProdutoDeleteNotification

```
using MediatR;

namespace AspNet_MediatR1.Notifications
{
    public class ProdutoDeleteNotification : INotification
    {
        public int Id { get; set; }
        public bool IsConcluido { get; set; }
    }
}
```

4- ErroNotification

```
using MediatR;

namespace AspNet_MediatR1.Notifications
{
    public class ErroNotification : INotification
    {
        public string Erro { get; set; }
        public string PilhaErro { get; set; }
    }
}
```

Precisamos criar uma classe do tipo *Notification Handler* que deverá "*escutar*" todas as notificações, pois todas serão apenas registradas no console.

Para isso vamos criar a pasta EventsHandlers no projeto e criar a classe LogEventHandler :

```
using AspNet_MediatR1.Notifications;
using MediatR;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace AspNet_MediatR1.EventsHandlers
{
    public class LogEventHandler :
                    INotificationHandler<ProdutoCreateNotification>,
                    INotificationHandler<ProdutoUpdateNotification>,
                    INotificationHandler<ProdutoDeleteNotification>,
                    INotificationHandler<ErroNotification>
    {
```

```
        public Task Handle(ProdutoCreateNotification notification, CancellationToken cancellationToken)
        {
            return Task.Run(() =>
            {
                Console.WriteLine($"CRIACAO: '{notification.Id} " +
                    $"- {notification.Nome} - {notification.Preco}'");
            });
        }

        public Task Handle(ProdutoUpdateNotification notification, CancellationToken cancellationToken)
        {
            return Task.Run(() =>
            {
                Console.WriteLine($"ALTERACAO: '{notification.Id} - {notification.Nome} " +
                    $"- {notification.Preco} - {notification.IsConcluido}'");
            });
        }

        public Task Handle(ProdutoDeleteNotification notification, CancellationToken cancellationToken)
        {
            return Task.Run(() =>
            {
                Console.WriteLine($"EXCLUSAO: '{notification.Id} " +
                    $"- {notification.IsConcluido}'");
            });
        }

        public Task Handle(ErroNotification notification, CancellationToken cancellationToken)
        {
            return Task.Run(() =>
            {
                Console.WriteLine($"ERRO: '{notification.Erro} \n {notification.PilhaErro}'");
            });
        }
    }
}
```

Podemos definir tantas classes *Notification Handlers* quanto forem necessárias. Caso uma notificação seja "*ouvida*" por mais de um Notification Handlers, todos serão invocados quando a notificação for gerada.

Criando os Command Handlers

Para cada objeto Command devemos ter um objeto Command Handler.

Vamos então criar os Command Handlers na pasta Handlers:

1- ProdutoCreateCommandHandler

```
using AspNet_MediatR1.Domain.Command;
using AspNet_MediatR1.Domain.Entity;
using AspNet_MediatR1.Notifications;
using AspNet_MediatR1.Repository;
using MediatR;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace AspNet_MediatR1.Domain.Handler
{
    public class ProdutoCreateCommandHandler : IRequestHandler<ProdutoCreateCommand, string>
    {
        private readonly IMediator _mediator;
        private readonly IRepository<Produto> _repository;

        public ProdutoCreateCommandHandler(IMediator mediator, IRepository<Produto> repository)
        {
            this._mediator = mediator;
            this._repository = repository;
        }
        public async Task<string> Handle(ProdutoCreateCommand request, CancellationToken cancellationToken)
        {
            var produto = new Produto { Nome = request.Nome, Preco = request.Preco };

            try
            {
                await _repository.Add(produto);
                await _mediator.Publish(new ProdutoCreateNotification { Id = produto.Id, Nome = produto.Nome, Preco = produto.Preco });
                return await Task.FromResult("Produto criada com sucesso");
            }
            catch (Exception ex)
            {
```

```
                await _mediator.Publish(new ProdutoCreateNotification { Id = produto.Id, Nome = produto.Nome, Preco = produto.Preco });
                await _mediator.Publish(new ErroNotification { Erro = ex.Message, PilhaErro = ex.StackTrace });
                return await Task.FromResult("Ocorreu um erro no momento da criação");
            }
        }
    }
}
```

2- ProdutoUpdateCommandHandler

```
using AspNet_MediatR1.Domain.Command;
using AspNet_MediatR1.Domain.Entity;
using AspNet_MediatR1.Notifications;
using AspNet_MediatR1.Repository;
using MediatR;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace AspNet_MediatR1.Domain.Handler
{
    public class ProdutoUpdateCommandHandler : IRequestHandler<ProdutoUpdateCommand, string>
    {
        private readonly IMediator _mediator;
        private readonly IRepository<Produto> _repository;

        public ProdutoUpdateCommandHandler(IMediator mediator, IRepository<Produto> repository)
        {
            this._mediator = mediator;
            this._repository = repository;
        }
        public async Task<string> Handle(ProdutoUpdateCommand request,
            CancellationToken cancellationToken)
        {
            var produto = new Produto { Id = request.Id, Nome = request.Nome,
                Preco = request.Preco };

            try
            {
                await _repository.Edit(produto);

                await _mediator.Publish(new ProdutoUpdateNotification
                { Id = produto.Id, Nome = produto.Nome, Preco = produto.Preco });

                return await Task.FromResult("Produto alterado com sucesso");
            }
            catch (Exception ex)
            {
                await _mediator.Publish(new ProdutoUpdateNotification { Id = produto.Id,
                    Nome = produto.Nome, Preco = produto.Preco });


                await _mediator.Publish(new ErroNotification { Erro = ex.Message,
                    PilhaErro = ex.StackTrace });

                return await Task.FromResult("Ocorreu um erro no momento da alteração");
            }
        }
    }
}
```

3- ProdutoDeleteCommandHandler

```
using System.Threading;
using System.Threading.Tasks;

namespace AspNet_MediatR1.Domain.Handler
{
    public class ProdutoDeleteCommandHandler : IRequestHandler<ProdutoDeleteCommand, string>
    {
        private readonly IMediator _mediator;
        private readonly IRepository<Produto> _repository;

        public ProdutoDeleteCommandHandler(IMediator mediator, IRepository<Produto> repository)
        {
            this._mediator = mediator;
            this._repository = repository;
        }
        public async Task<string> Handle(ProdutoDeleteCommand request,
            CancellationToken cancellationToken)
        {
            try
            {
                await _repository.Delete(request.Id);
```

```
            await _mediator.Publish(new ProdutoDeleteNotification
            { Id = request.Id, IsConcluido = true });

            return await Task.FromResult("Produto excluido com sucesso");
        }
        catch (Exception ex)
        {
            await _mediator.Publish(new ProdutoDeleteNotification
            {
                Id = request.Id,
                IsConcluido = false
            });

            await _mediator.Publish(new ErroNotification
            {
                Erro = ex.Message,
                PilhaErro = ex.StackTrace
            });

            return await Task.FromResult("Ocorreu um erro no momento da exclusão");
        }
    }
  }
}
```

Os command handlers implementam a interface IRequestHandler, onde é especificada uma classe Command e o tipo de retorno. Quando esta classe Command gerar uma solicitação, o MediatR irá invocar o *command handler,* chamando o método Handler.

É no método Handler onde são definidas instruções que devem ser realizadas para aplicar a solicitação definida pelo command.

Após a solicitação ser atendida, podemos usar o método Publish() para emitir uma notificação para todo sistema. Aqui o *MediatR* vai procurar pela classe com a implementação da interface INotificationHandler<notificacao> e invocar o método Handler() para processar aquela notificação que implementamos.

Criando o controlador

Agora vamos criar o controlador da nossa aplicação.

Clique com o botão direito do mouse sobre a pasta Controllers e a seguir clique em Add->Controller;

A seguir escolha a opção API Controller - Empty e clique em Add;

Informe o nome ProdutosController e clique em Add.

Ao final será criado o controlador onde vamos incluir o código abaixo:

```
using AspNet_MediatR1.Domain.Command;
using AspNet_MediatR1.Domain.Entity;
using AspNet_MediatR1.Repository;
using MediatR;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace AspNet_MediatR1.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProdutosController : ControllerBase
    {
        private readonly IMediator _mediator;
        private readonly IRepository<Produto> _repository;

        public ProdutosController(IMediator mediator, IRepository<Produto> repository)
        {
            this._mediator = mediator;
            this._repository = repository;
        }

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            return Ok(await _repository.GetAll());
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> Get(int id)
        {
```

```
            return Ok(await _repository.Get(id));
        }

        [HttpPost]
        public async Task<IActionResult> Post(ProdutoCreateCommand command)
        {
            var response = await _mediator.Send(command);
            return Ok(response);
        }

        [HttpPut]
        public async Task<IActionResult> Put(ProdutoUpdateCommand command)
        {
            var response = await _mediator.Send(command);
            return Ok(response);
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete(int id)
        {
            var obj = new ProdutoDeleteCommand { Id = id };
            var result = await _mediator.Send(obj);
            return Ok(result);
        }
    }
}
```

Injetamos no construtor uma instância do repositório e da interface IMediator pois precisamos enviar as requisições dos nosso objetos Command usando o método Send.

Aqui a interface IMediator faz o papel da classe mediadora que usa o método Send par chamar os *comand handlers* definidos.

Assim para tudo isso funcionar precisamos adicionar o MediatR como um serviço e registrar o serviço do repositório no método ConfigureServices:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddMediatR(typeof(Startup));
    services.AddSingleton<IRepository<Pessoa>, PessoaRepository>();
}
```

Para testar podemos simplesmente definir alguns produtos em memória no Repositório ProdutoRepository:

```
public class ProdutoRepository : IRepository<Produto>
    {
        private Dictionary<int, Produto> produtos = new Dictionary<int, Produto>();

        public Dictionary<int, Produto> GetProdutos()
        {
            produtos.Add(1, new Produto { Id = 1, Nome = "Caneta", Preco = 3.45m });
            produtos.Add(2, new Produto { Id = 2, Nome = "Caderno", Preco = 7.65m });
            produtos.Add(3, new Produto { Id = 3, Nome = "Borracha", Preco = 1.20m });
            return produtos;
        }

        public ProdutoRepository()
        {
            produtos = GetProdutos();
        }
        ...
    }
```

E podemos alterar o arquivo launchSettings.json definindo a *url inicial* de acesso para : api/produtos

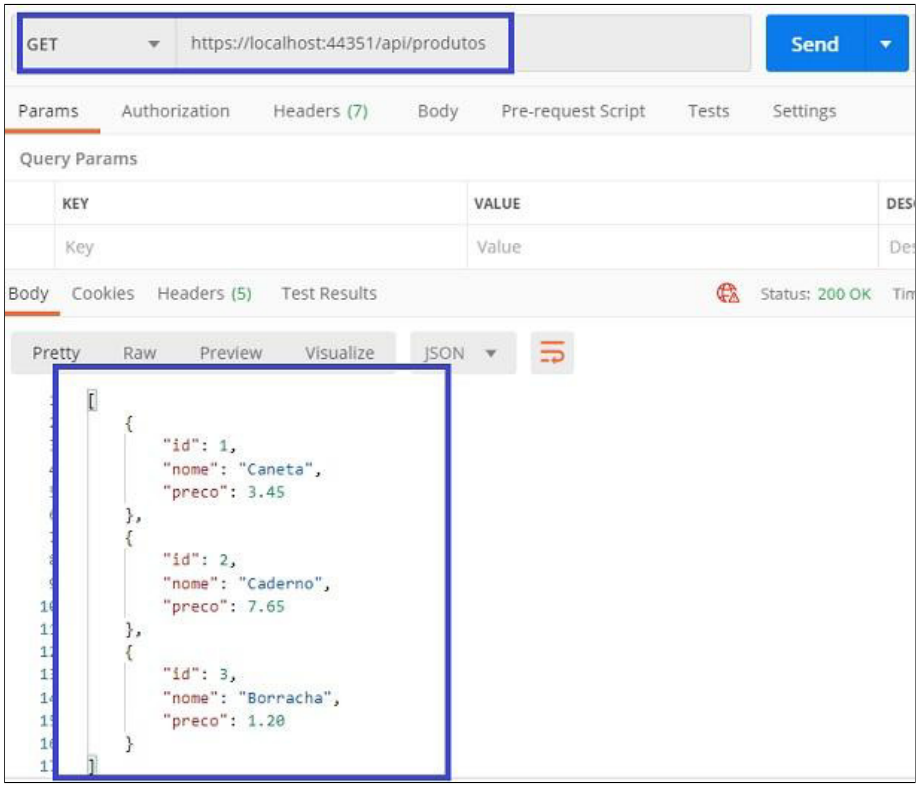
```
...
"profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "api/produtos",
      "environmentVariables": {
```

```
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
  "AspNet_MediatR1": {
    "commandName": "Project",
    "launchBrowser": true,
    "launchUrl": "api/produtos",
    "applicationUrl": "https://localhost:5001;http://localhost:5000",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  }
...
```

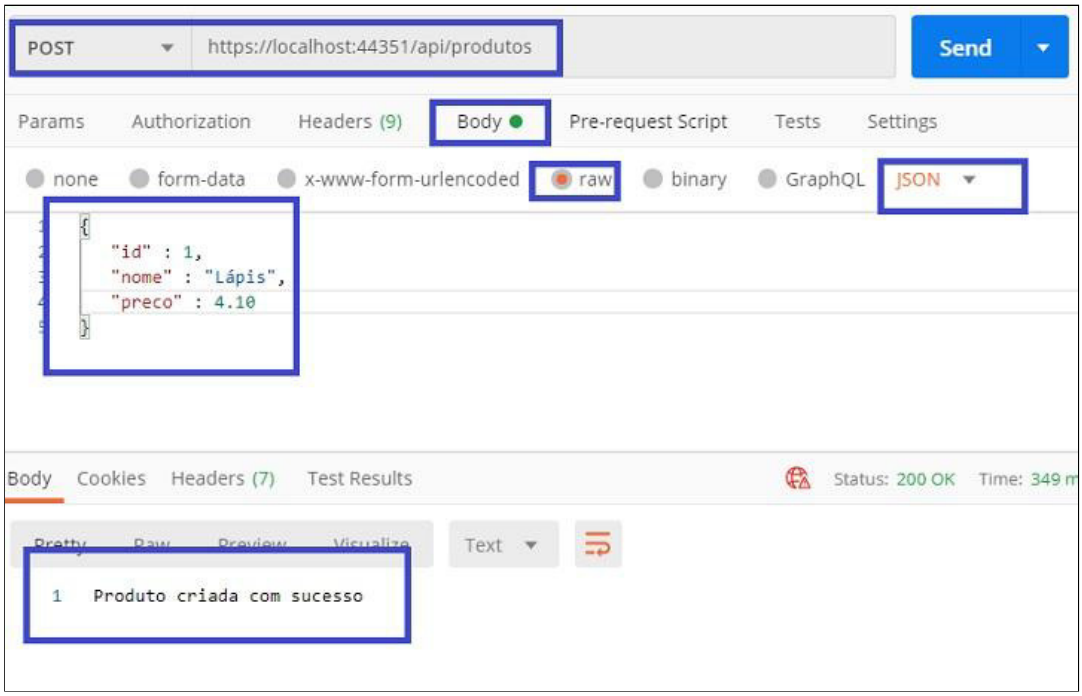
Agora é só alegria.

Executando o projeto teremos e usando o Postman podemos testar nossa implementação.

1- Fazendo um GET



2- Fazendo um POST



O projeto completo aqui: **AspNet_MediatR1.*zip*** *(sem as referências)*

"E, chegando-se Jesus, falou-lhes, dizendo: É-me dado todo o poder no céu e na terra."
Mateus 28:18

Referências:

- Seção ASP .NET do site Macoratti .net
- Curso Básico VB .NET - Vídeo Aulas
- Curso C# Básico - Vídeo Aulas
- Curso Fundamentos da Programação Orientada a Objetos com VB .NET NEW
- Macoratti .net | Facebook
- macoratti - YouTube
- ASP .NET Core - Implementando a segurança com ... - Macoratti
- ASP.NET Core MVC - Criando um Dashboard ... - Macoratti.net
- C# - Gerando QRCode - Macoratti
- ASP .NET - Gerando QRCode com a API do Google

José Carlos Macoratti