COMPILADO COM EXEMPLOS DE CONCEITOS AVANÇADOS, APLICADOS AO DESENVOLVIMENTO DE SOFTWARES.

Sumário

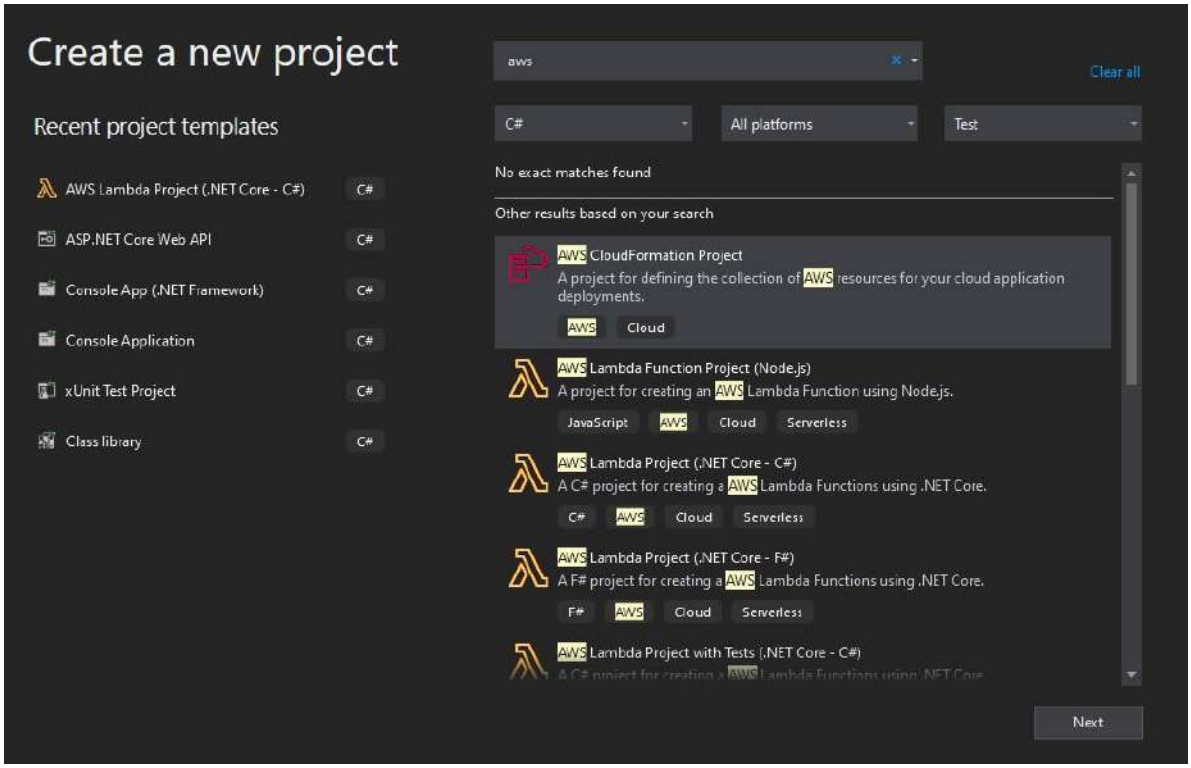
AWS LAMBDA

Antes de tudo foi instalado o pacote da Aws para o visual studio.

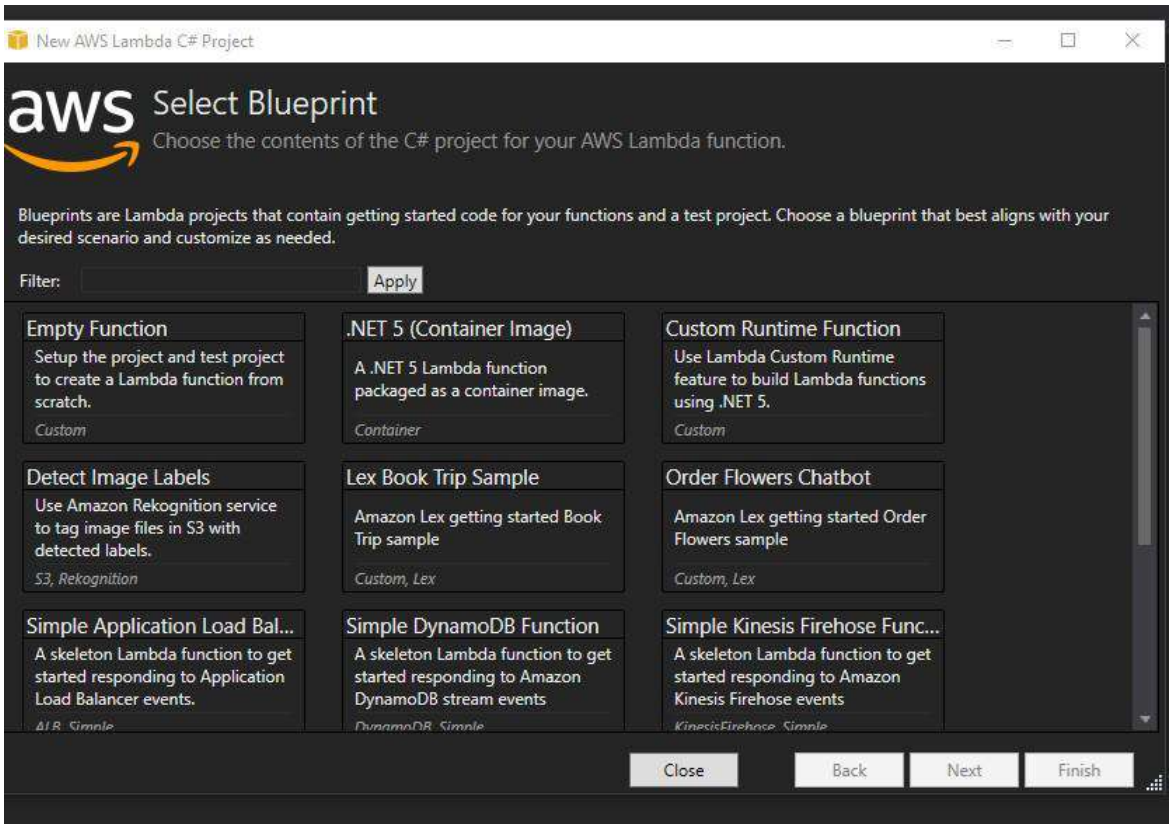
https://aws.amazon.com/pt/visualstudio/

No canto superior clique em baixar e instalar somente selecionando next.

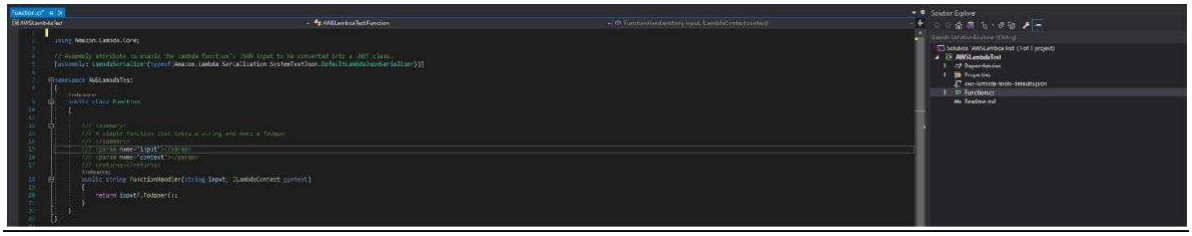
Agora para criar o novo projeto você deve selecionar File -> New Project e escrever aws conforme a imagem abaixo.



Neste meu exemplo irei selecionar Aws Lambda Project (.NET Core – C#) e em seguida definir a pasta onde será criado o projeto aparecerá esta outra janela.



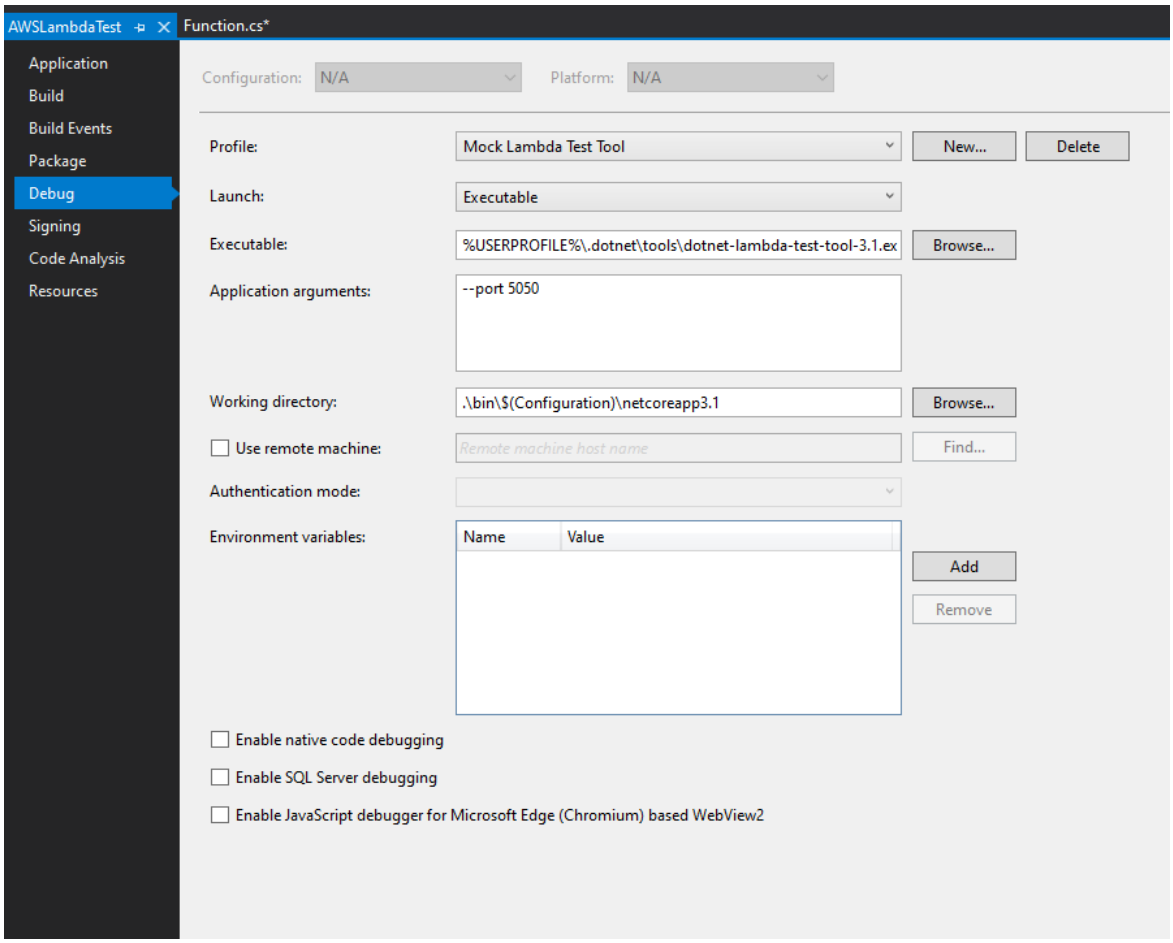
Para este exemplo criei um projeto com Empty Function com o nome de AwsLambdaTest.

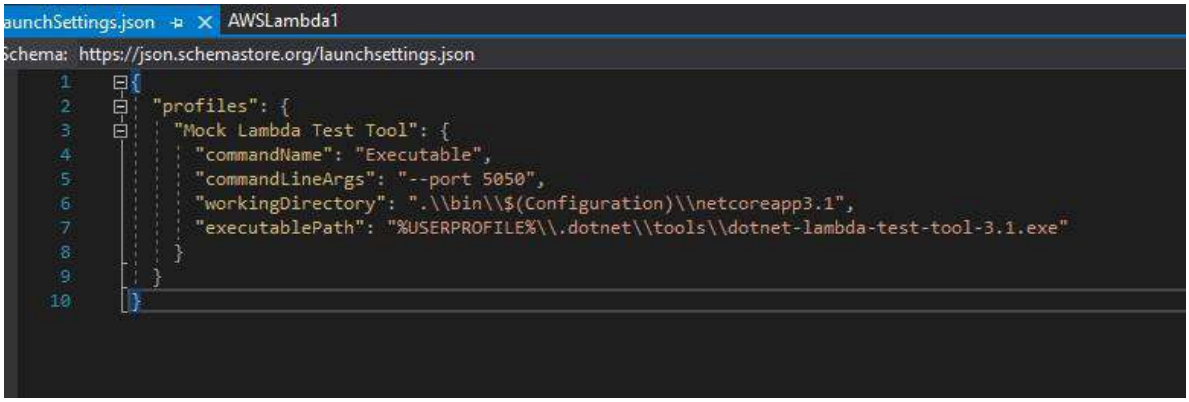


Projeto criado agora é só colocar para rodar.

Mas antes deverá clicar com botão direito em cima do projeto e selecionar “Properties”

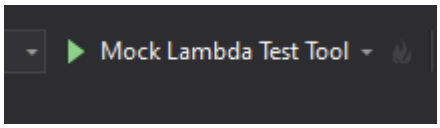
Nesta tela de debug, deverá criar uma nome para um profile e após popular o “launchSettings.json” conforme a seguir.





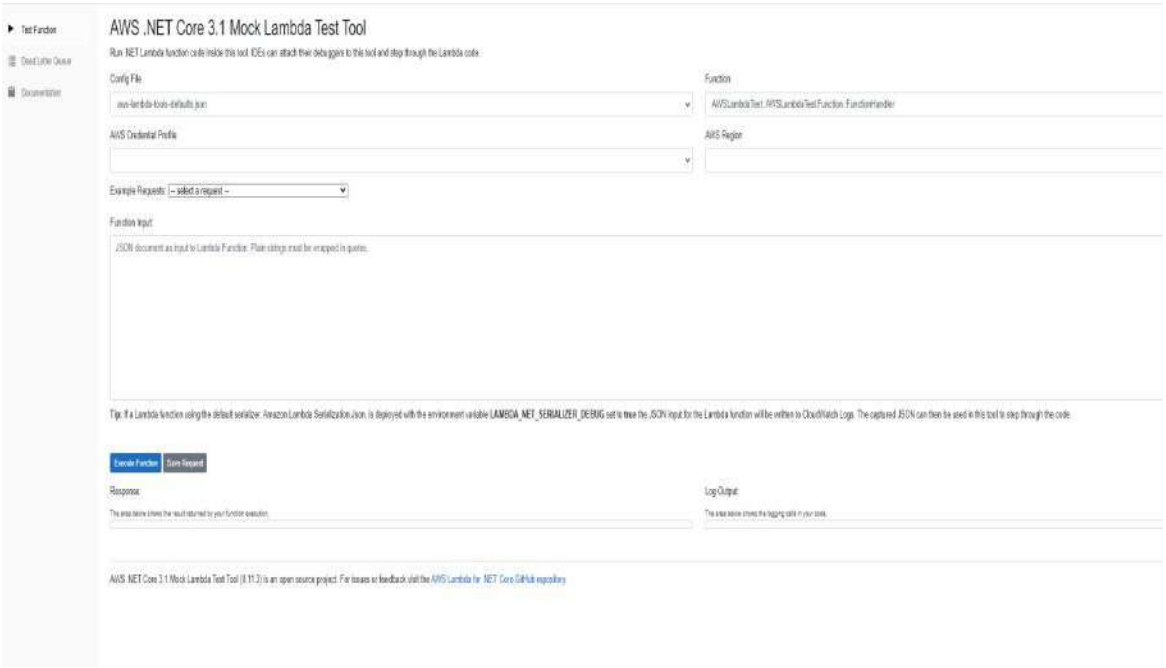

```
{
  "profiles": {
    "Mock Lambda Test Tool": {
      "commandName": "Executable",
      "commandLineArgs": "--port 5050",
      "workingDirectory": ".\\bin\\$(Configuration)\\netcoreapp3.1",
      "executablePath": "%USERPROFILE%\\.dotnet\\tools\\dotnet-lambda-test-tool-3.1.exe"
    }
  }
}
```

Pronto agora é só você selecionar o profile para debugar.

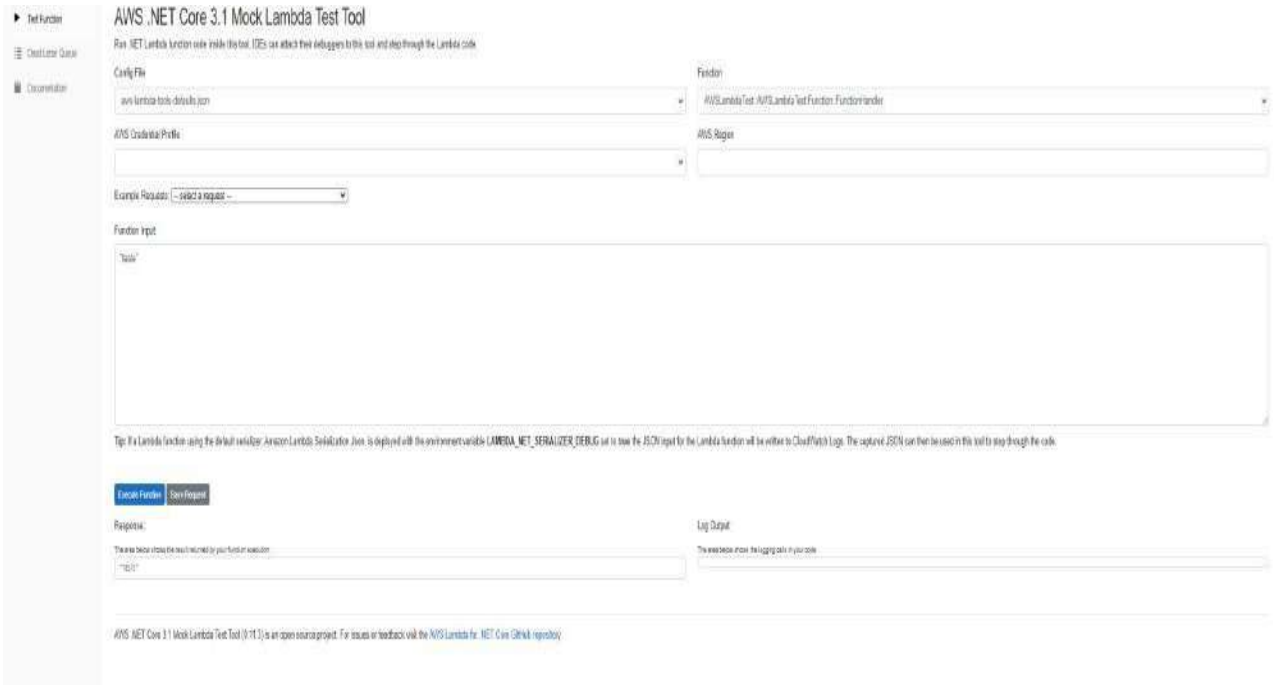


E rodar o projeto.

Subirá uma tela conforme a imagem abaixo.



Após adicionar o input de teste, logo retornara o resultado abaixo.



PADRAO – CQRS (COMMAND QUERY RESPONSIBILITY E SEGREGATION)

Primeiramente devemos entender o que significa este

padrão

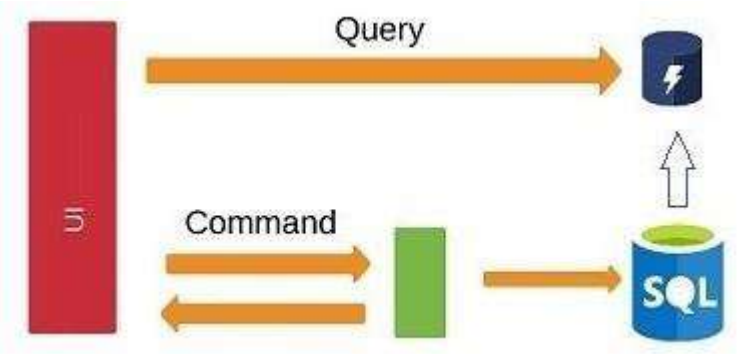
Command seria os updates , insert e deletes comando para executar

algo.Query seriam as consultas enviadas ao banco de dados.

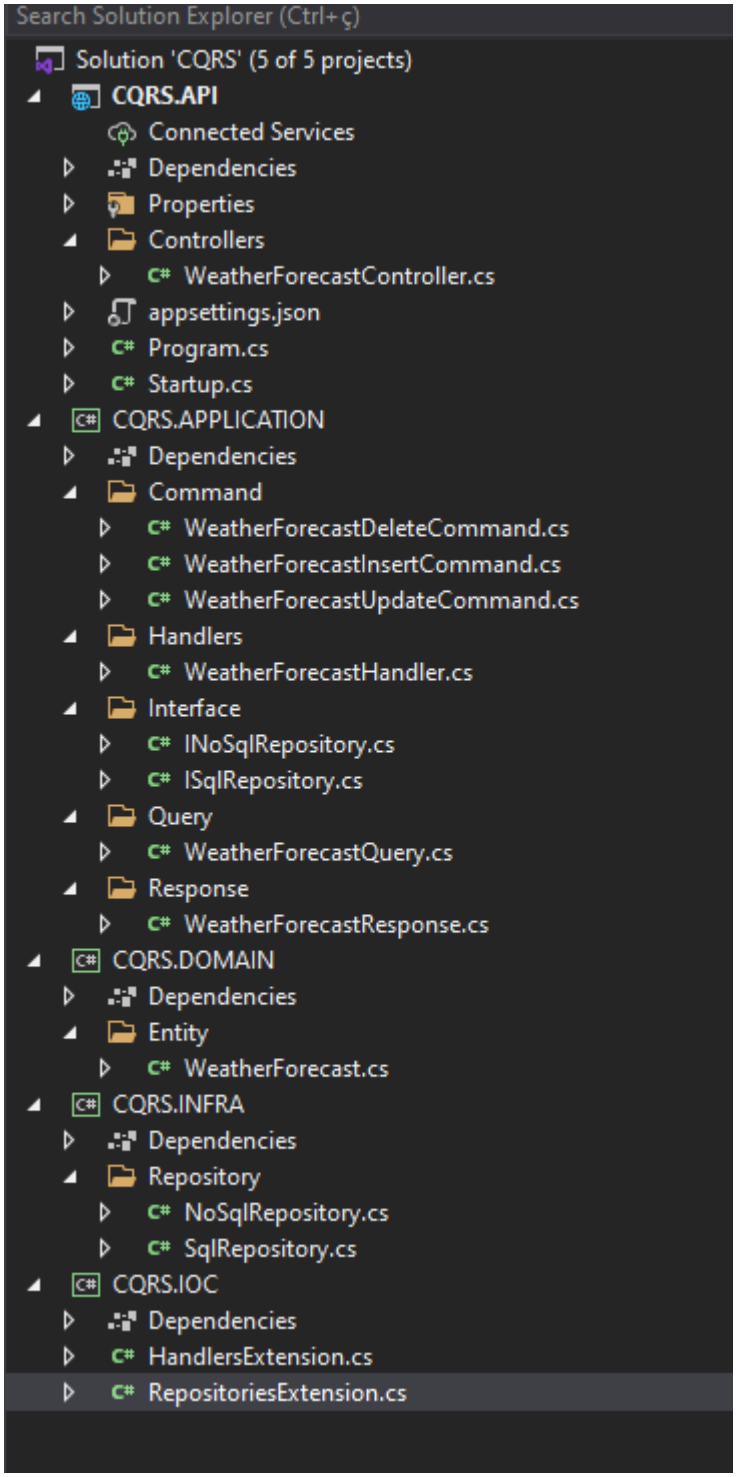
Responsabilidade e segregação, seria cada uma com suas responsabilidades de segregação portipos.

Para termos mais eficiência nas consultas podemos utilizar uma consulta no banco de dados emcachê que seria muito mais rápida a utilização.

E para efeitos de comandos, no exemplo utilizei o banco de dados Sql Server .



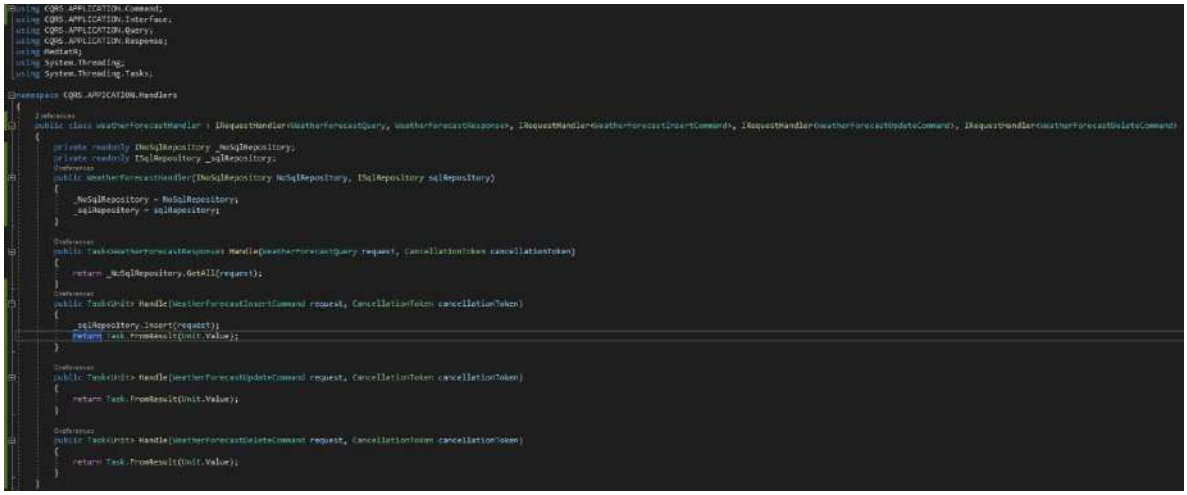
Nesta imagem podemos exemplificar a arquitetura CQRS, pois supondo que a query faz a consulta direta no DataBase em cachê, e o command passaria por uma outra camada de validação , regras de negócios podendo até exigir uma Transaction. Por isso trata-se de uma camada mais complexa do que simplesmente uma query.



Nesta meu exemplo podemos verificar que possuo uma Controller que faz a primeira camada de UI.

```
namespace CQRS.Controllers
{
    [ApiController]
    [Route("[controller]")]
    3 references
    public class WeatherForecastController : ControllerBase
    {
        private readonly IMediator _mediator;


        private readonly ILogger<WeatherForecastController> _logger;

        0 references
        public WeatherForecastController(ILogger<WeatherForecastController> logger, IMediator mediator)
        {
            _logger = logger;
            _mediator = mediator;
        }

        [HttpGet]
        0 references
        public async Task<WeatherForecastResponse> Get()
        {
            var weatherForecastQuery = new WeatherForecastQuery();
            return await _mediator.Send(weatherForecastQuery);
        }

        [HttpPost]
        0 references
        public async Task Post([FromBody] WeatherForecastInsertCommand weatherForecastInsertCommand)
        {
            await _mediator.Send(weatherForecastInsertCommand);
        }

        [HttpPut]
        0 references
        public async Task Put([FromBody] WeatherForecastUpdateCommand weatherForecastUpdateCommand)
        {
            await _mediator.Send(weatherForecastUpdateCommand);
        }


        [HttpDelete]
        0 references
        public async Task Delete([FromQuery]int id)
        {
            await _mediator.Send(new WeatherForecastDeleteCommand(id));
        }
```

Nesta esta sendo passados os parâmetros de query e commands que seguiram chamando uma aplication



Onde cada uma chamará respectivamente seu repositório.Como

a imagem a seguir.

```
using CQRS.APPLICATION.Interface;
using CQRS.DOMAIN.Entity;
using System.Threading.Tasks;

namespace CQRS.INFRA.Repository
{
    1 reference
    public class SqlRepository : ISqlRepository
    {
        1 reference
        public Task Delete(int id)
        {
            return Task.CompletedTask;
        }

        2 references
        public Task Insert(WeatherForecast weatherForecast)
        {
            return Task.CompletedTask;
        }

        1 reference
        public Task Update(WeatherForecast weatherForecast)
        {
            return Task.CompletedTask;
        }
    }
}
```

INJEÇÃO DE DEPENDENCIA COM MEDIATOR

Foi criado uma aplicação WebApi para demonstrar a injeção e seus padrões.

Instalar no pacote nuget conforme imagem abaixo.

Após a instalação do pacote podemos inicializar o Mediator na classe startUp

```
0 references
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();

    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "InjecaoMediator", Version = "v1" });
    });
    services.AddMediatR(typeof(Startup));
    services.AddScoped<IWeatherForecastRepository, WeatherForecastRepository>();
}
```

Por padrão do Mediator obrigatoriamente utiliza-se do mesmo padrão do CQRS(Command, Query, Responsability e Segregation).

Portanto foi criado uma classe WeatherForecastQuery que irá herdar do IRequest do MediaTr e deverá sempre conter o Response conforme imagem abaixo.

```
using InjecaoMediator.Response;
using MediatR;

namespace InjecaoMediator.Query
{
    5 references
    public class WeatherForecastQuery : IRequest<WeatherForecastResponse>
    {
    }
}
```

O Response nada mais é do que a resposta enviada pela consulta, conforme o exemplo criado.

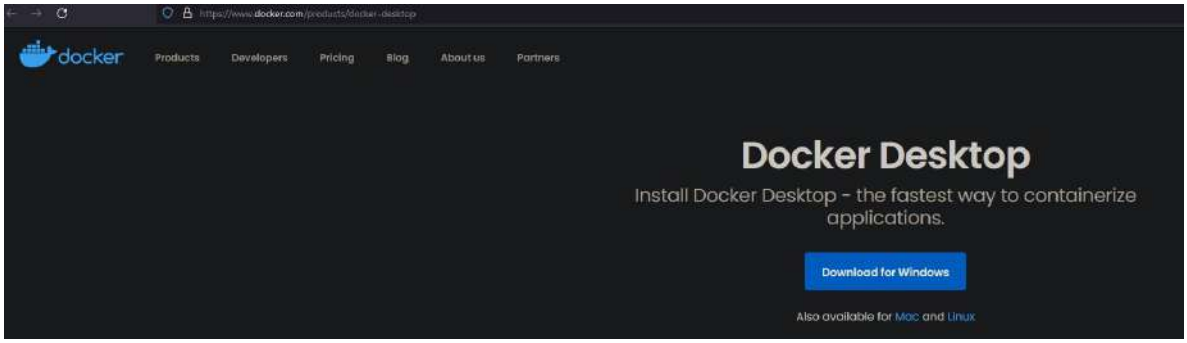
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace InjecaoMediator.Response
{
    7 references
    public class WeatherForecastResponse
    {
        1 reference
        public DateTime Date { get; set; }

        2 references
        public int TemperatureC { get; set; }

        0 references
        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

        1 reference
        public string Summary { get; set; }
    }
}
```

Feito isto, podemos implementar a classe Handle que herdará da Interface IRequestHandler, que por sua ver obrigará a implementar o handle. No entanto foi criado um repositório para efetuar a busca ou a inserção dos objetos.

MONGODB

Para iniciar o ambiente sera necessário instalar o Docker.

https://www.docker.com/products/docker-desktop



Após instalar verifique se o docker esta rodando corretamente.

Utilize um terminal ou prompt de comand e digite “docker ps”

Se estiver ok, grave o arquivo a seguir no diretório que desejar e abra o prompt de comando na mesma pasta do arquivo e digite o seguinte comando “docker-compose up”.

Neste arquivo contém as informações que precisa como login e senha de conexão para o banco de dados do mongodb.

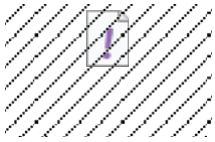
docker-compose.yml

```yaml
version: '3'

services:
  mongo-express:
    image: mongo-express
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_BASICAUTH_USERNAME: everton
      ME_CONFIG_BASICAUTH_PASSWORD: everton
      ME_CONFIG_MONGODB_PORT: 27017
      ME_CONFIG_MONGODB_ADMINUSERNAME: everton
      ME_CONFIG_MONGODB_ADMINPASSWORD: everton
    links:
      - mongo
    networks:
      - mongo-compose-network

  mongo:
    image: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: everton
      MONGO_INITDB_ROOT_PASSWORD: everton
    ports:
      - "27017:27017"
    volumes:
      - /home/everton/Docker/Volumes/MongoDB:/data/db
    networks:
      - mongo-compose-network

networks:
    mongo-compose-network:
      driver: bridge
```

Texto do docker compose

```yaml
version: '3'


services:

  mongo-express:

    image: mongo-express
    ports:

      - 8081:8081

    environment:
```

```
      ME_CONFIG_MONGODB_ADMINUSERNAME: everton
      ME_CONFIG_MONGODB_ADMINPASSWORD: everton
    links:
      - mongo
    networks:
      - mongo-compose-network

  mongo:
    image: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: everton
      MONGO_INITDB_ROOT_PASSWORD: everton
    ports:
      - "27017:27017"
    volumes:
      - /home/everton/Docker/Volumes/MongoDB:/data/db
    networks:
```



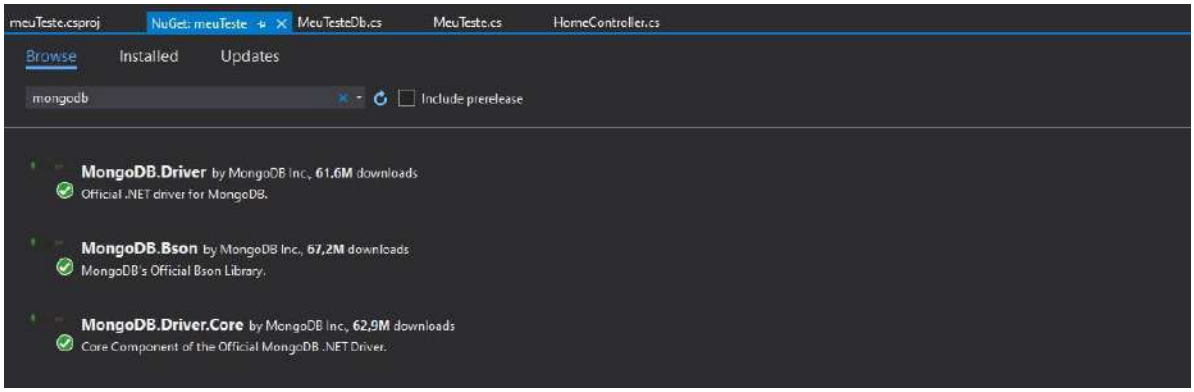
Após rodar o comando, rode o comando “docker ps” verificando se o mesmo subiu com sucesso.

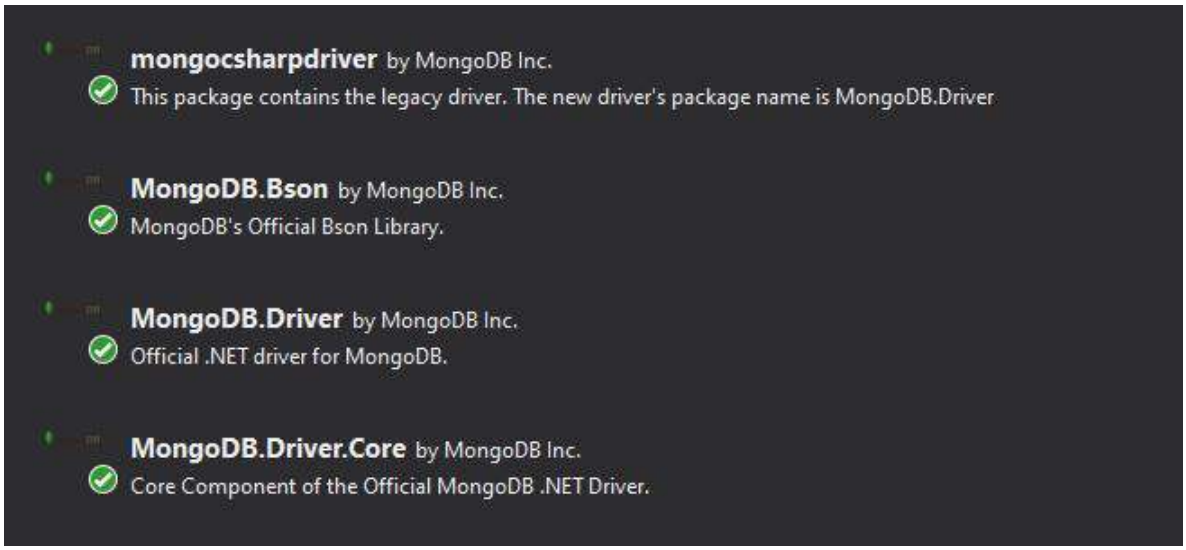
```
λ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED       STATUS         PORTS                                           NAMES
daff70c93469   mongo     "docker-entrypoint.s…"   3 weeks ago   Up 7 seconds   0.0.0.0:27017->27017/tcp, :::27017->27017/tcp   default_mongo_1
```

Criei uma aplicação webApi para fazer a interface com o Mongodb

Foram instalados os seguintes drivers para conectar com o MongoDb.

Acesse o Gerenciador de pacotes do Nuget e digite mongodb

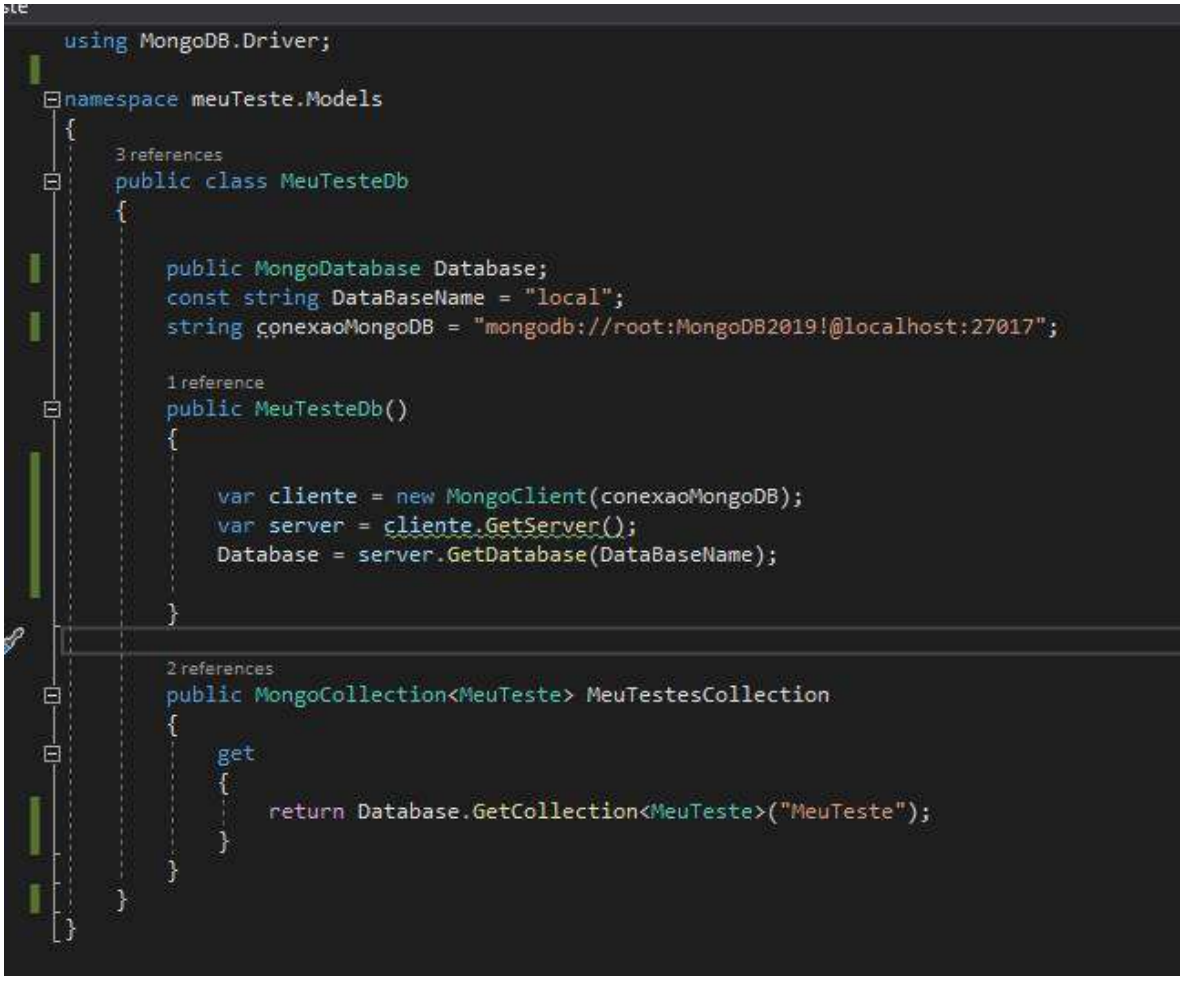




Temos uma classe modelo que representa a entidade no mongodb, conforme imagem abaixo podemos notar que existem tags que fazem dos atributos como campos requeridos elementos do Documento e do Id.

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using System.ComponentModel.DataAnnotations;

4 references
public class MeuTeste
{
    [BsonId()]
    0 references
    public ObjectId Id { get; set; }
    [Required]
    [Display(Name = "Nome")]
    [BsonRequired()]
    [BsonElement("Nome")]
    0 references
    public string Nome { get; set; }
    [Required]
    [Display(Name = "Cargo")]
    [BsonElement("Cargo")]
    [BsonRequired()]
    0 references
    public string Cargo { get; set; }
}
```

Nesta outra classe MeuTesteDb, podemos efetuar a conexão com o Banco MongoDb e retornamos o context do Banco de dados.

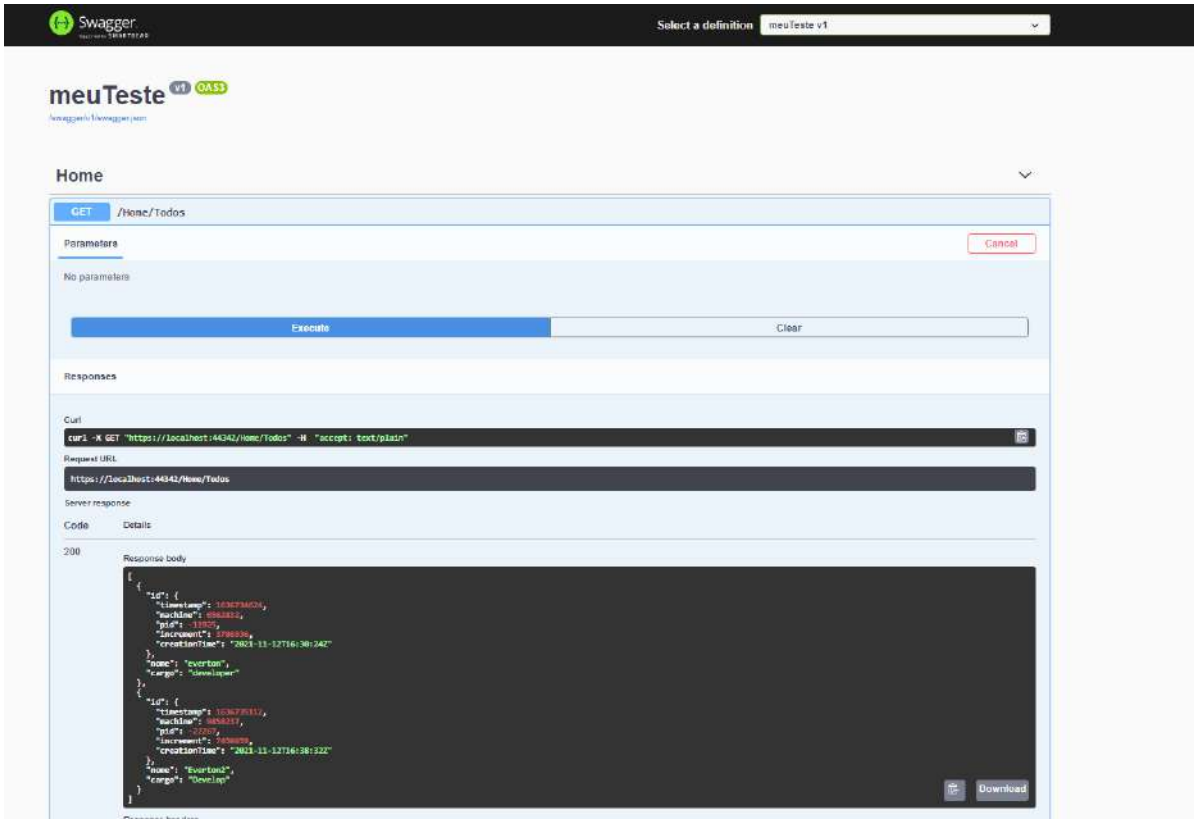


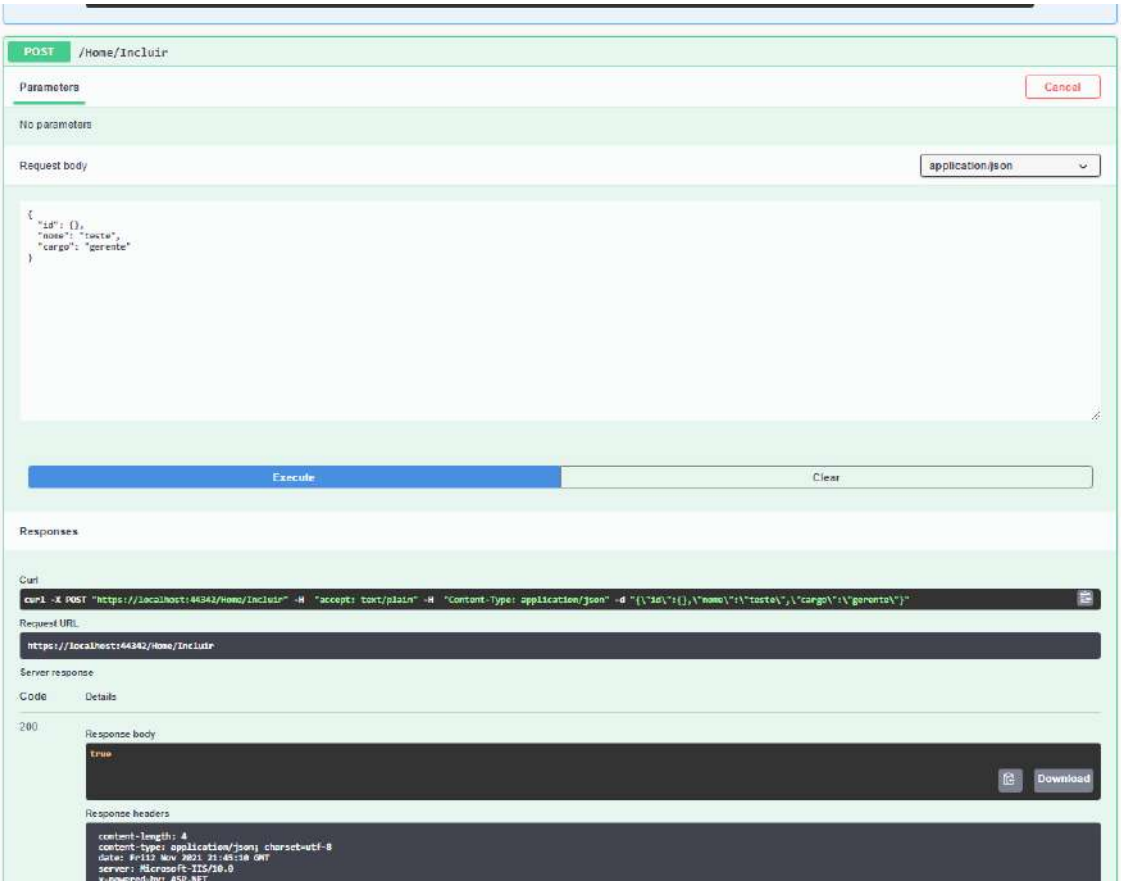
```
using MongoDB.Driver;

namespace meuTeste.Models
{
    3 references
    public class MeuTesteDb
    {

        public MongoDatabase Database;
        const string DataBaseName = "local";
        string conexaoMongoDB = "mongodb://root:MongoDB2019!@localhost:27017";

        1 reference
        public MeuTesteDb()
        {

            var cliente = new MongoClient(conexaoMongoDB);
            var server = cliente.GetServer();
            Database = server.GetDatabase(DataBaseName);

        }

        2 references
        public MongoCollection<MeuTeste> MeuTestesCollection
        {
            get
            {
                return Database.GetCollection<MeuTeste>("MeuTeste");
            }
        }
    }
}
```

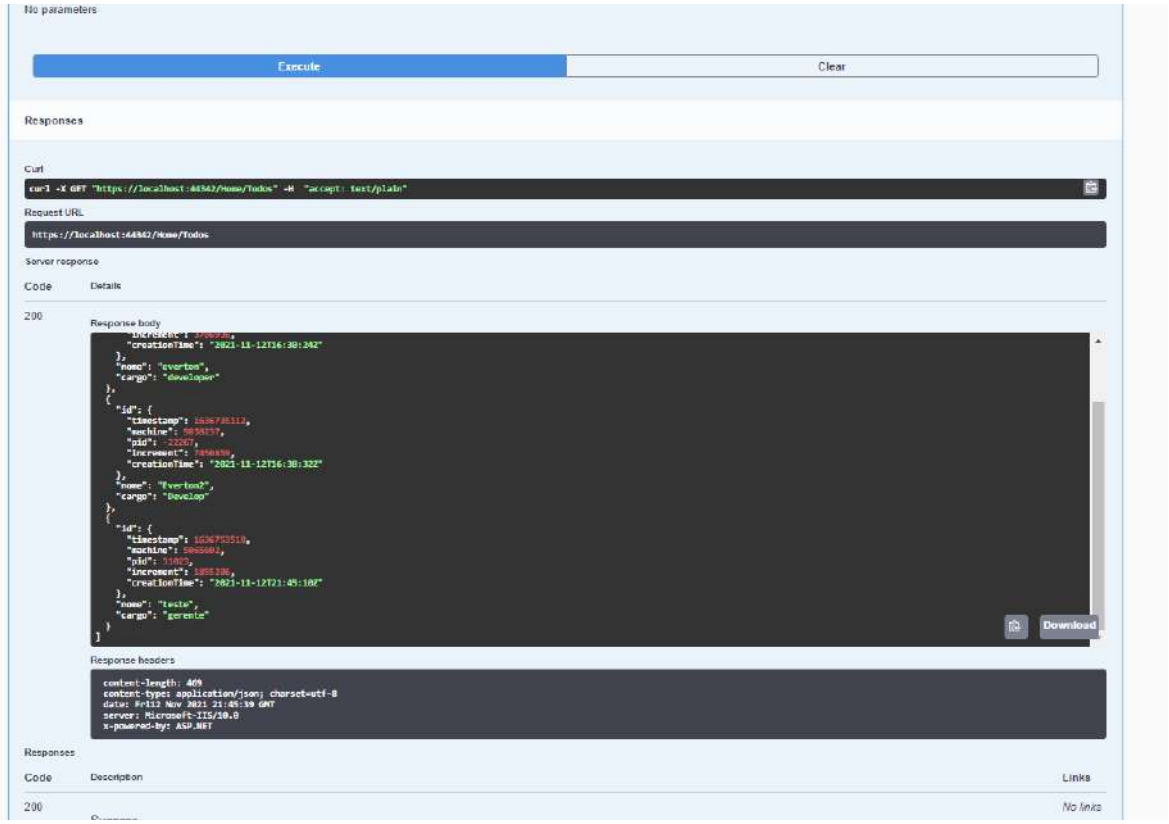
E ja na controller podemos verificar que em posse do contexto efetuamos as query's de consulta e insert, conforme a imagem abaixo.

```
using meuTeste.Models;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;

namespace meuTeste.Controllers
{
    [ApiController]
    [Route("[controller]")]
    0 references
    public class HomeController : ControllerBase
    {
        private readonly MeuTesteDb Context = new MeuTesteDb();

        [HttpGet("Todos")]
        0 references
        public IEnumerable<MeuTeste> Index()
        {
            try
            {
                return Context.MeuTestesCollection.FindAll();
            }
            catch(Exception erro)
            {
                throw new Exception(erro.Message);
            }

        }

        [HttpPost("Incluir")]
        0 references
        public bool Incluir(MeuTeste meuTeste)
        {
            try
            {
                Context.MeuTestesCollection.Insert(meuTeste);

                return true;
            }
            catch(Exception erro)
            {
                throw new Exception(erro.Message);
            }
        }
    }
}
```

Após rodar esta aplicação podemos identificar o funcionamento das chamadas.


Swagger
Select a definition
meuTeste v1
meuTeste
Home
GET
/Home/Todos
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X GET "https://localhost:44342/Home/Todos" -H "accept: text/plain"
Request URL
https://localhost:44342/Home/Todos
Server response
Code
Details
200
Response body
Download

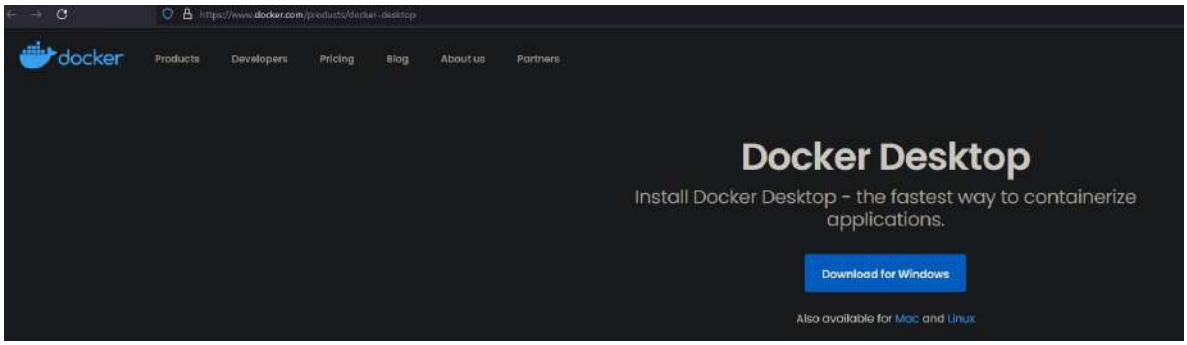

POST
/Home/Incluir
Parameters
Cancel
No parameters
Request body
application/json
Execute
Clear
Responses
Curl
curl -X POST "https://localhost:44342/Home/Incluir" -H "accept: text/plain" -H "Content-Type: application/json" -d "{\"id\":{},\"nome\":\"teste\",\"cargo\":\"gerente\"}"
Request URL
https://localhost:44342/Home/Incluir
Server response
Code
Details
200
Response body
true
Download
Response headers
content-length: 4
content-type: application/json; charset=utf-8
server: Microsoft-IIS/10.0


No parameters
Execute
Clear
Responses
Curl
curl -X GET "https://localhost:44342/Home/Todos" -H "accept: text/plain"
Request URL
https://localhost:44342/Home/Todos
Server response
Code
Details
200
Response body
"creationTime": "2021-11-12T16:38:24Z"
"nome": "everton",
"cargo": "developer"
"timestamp": 1636735312,
"machine": 8858157,
"pid": -22267,
"increment": 7858899,
"creationTime": "2021-11-12T16:38:32Z"
"nome": "Everton2",
"cargo": "Develop"
"timestamp": 1636753510,
"machine": 5866602,
"pid": 31025,
"increment": 1855186,
"creationTime": "2021-11-12T21:45:10Z"
"nome": "teste",
"cargo": "gerente"
Download
Response headers
content-length: 469
content-type: application/json; charset=utf-8
date: Fri12 Nov 2021 21:45:39 GMT
server: Microsoft-IIS/10.0
x-powered-by: ASP.NET
Responses
Code
Description
Links
200
No links

RABBITMQ

Para iniciar o ambiente sera necessário instalar o Docker.

https://www.docker.com/products/docker-desktop



Após instalar verifique se o docker esta rodando corretamente.

Utilize um terminal ou prompt de comand e digite “docker ps”

```
λ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Se estiver ok, grave o arquivo a seguir no diretório que desejar e abra o prompt de comando na

mesma pasta do arquivo e digite o seguinte comando “docker-compose up”.

```
version: '3.4'

services:
  rabbit:
    image: rabbitmq:3-management
    ports:
      - "15672:15672"
      - "5672:5672"
```

version: '3.4'

services:

rabbit:

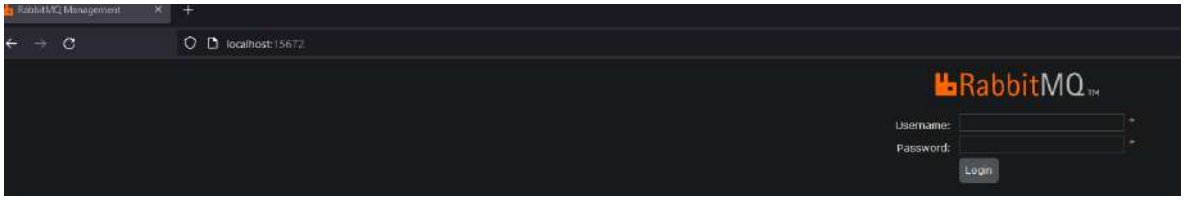
image: rabbitmq:3-management

ports:

- "15672:15672"

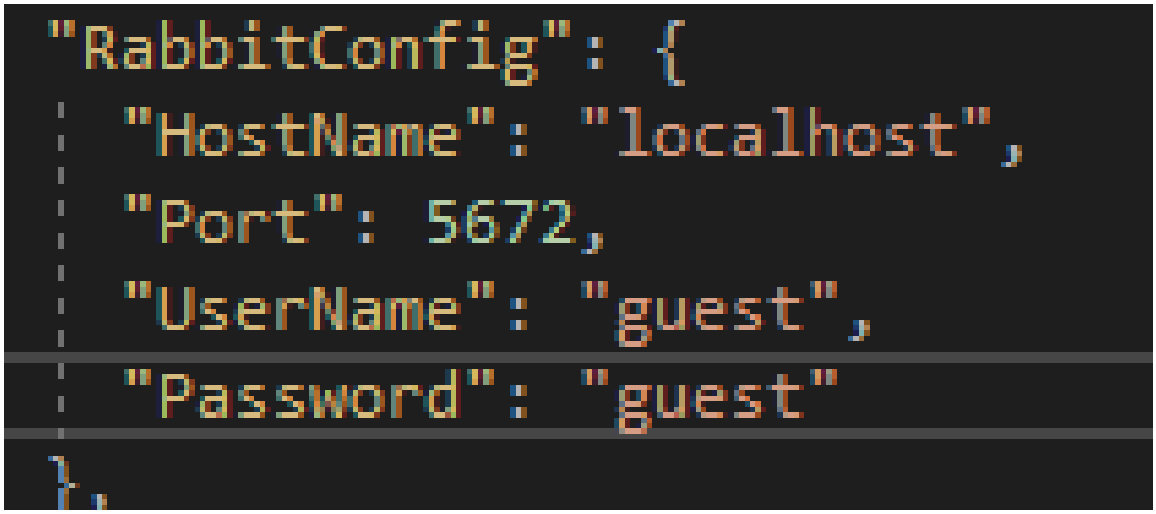
- "5672:5672"

Após o comando executado com sucesso, podemos verificar abrindo um webBrowser chamando a url a seguir.



No campo username deverá inserir o “guest” e a senha “guest” padrões para o Rabbit.

Agora falando sobre o projeto, foi criado uma webApi com arquivo de configuração appsettings.json



Podendo identificar a configuração do Rabbit para criar a conexão conforme exemplo abaixo.

```
var factory = new ConnectionFactory
{
    HostName = _configuration.HostName,
    Port = _configuration.Port,
    Password = _configuration.Password,
    UserName = _configuration.UserName
};
_connection = factory.CreateConnection();
```

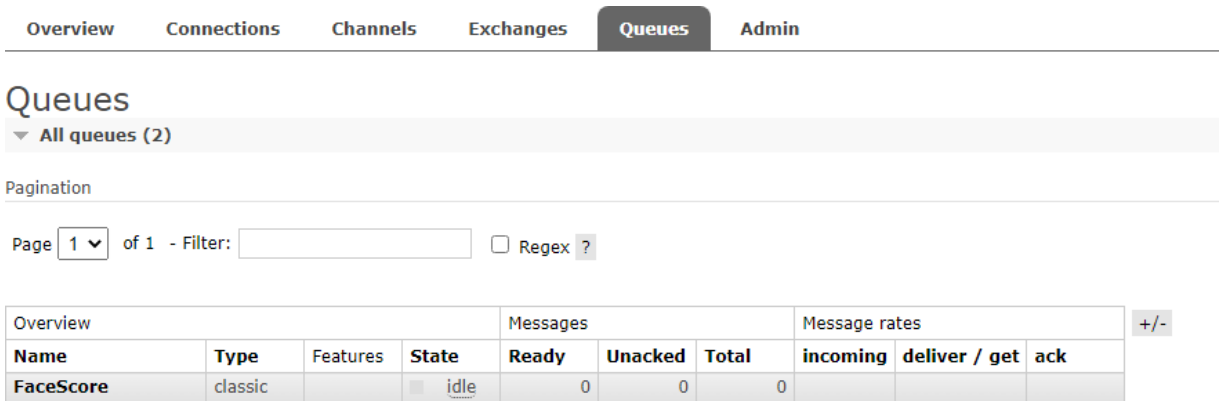
Para criar nome da fila deixamos também no arquivo de configuração.

```
"Queue": {
  "FaceScoreConsumer": {
    "PrefixForRetriable": "FaceScore",
    "MaxRetries": 3,
    "TtlTimeout": 300000
  }
```

Com o Rabbit já conectada podemos criar a fila

```
_channel = _connection.CreateModel();
_channel.QueueDeclare(
            queue: _queue.PrefixForRetriable,
            durable: false,
            exclusive: false,
            autoDelete: false,
            arguments: null);
```

Neste caso podemos identificar o exemplo criado conforme a interface do Rabbit



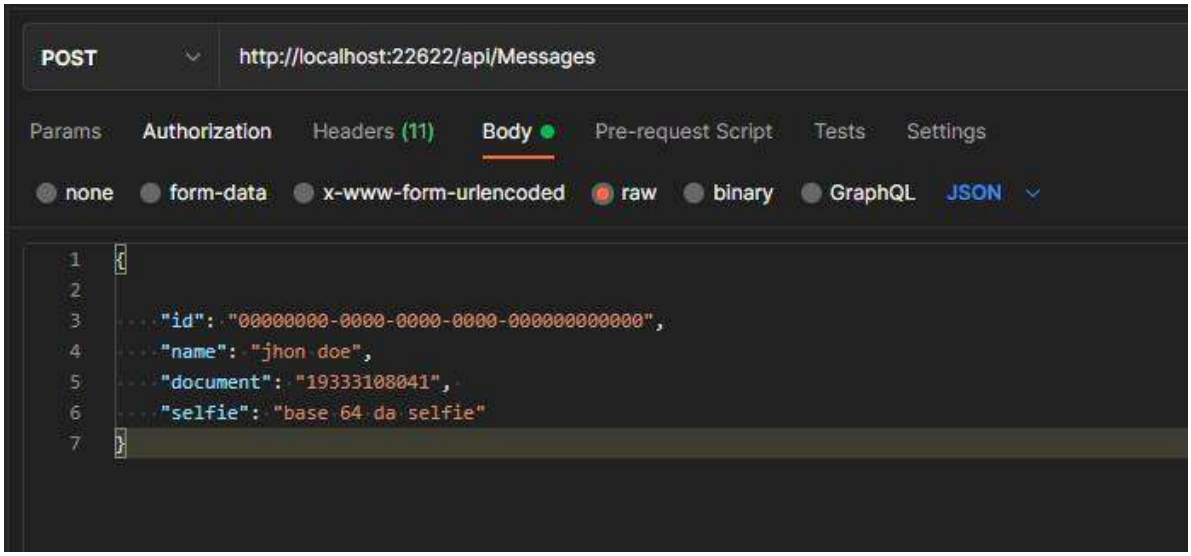
Para o producer, foi criado uma Controller para o publish da message para o broker.

```
private readonly ConnectionFactory _factory;
private readonly QueueConfig _queue;
private readonly RabbitConfig _config;
0 references
public MessagesController(IOptions<RabbitConfig> options, IOptions<QueueConfig> queueConf)
{
    _config = options.Value;
    _queue = queueConf.Value;

    _factory = new ConnectionFactory
    {
        HostName = _config.HostName
    };
}
```

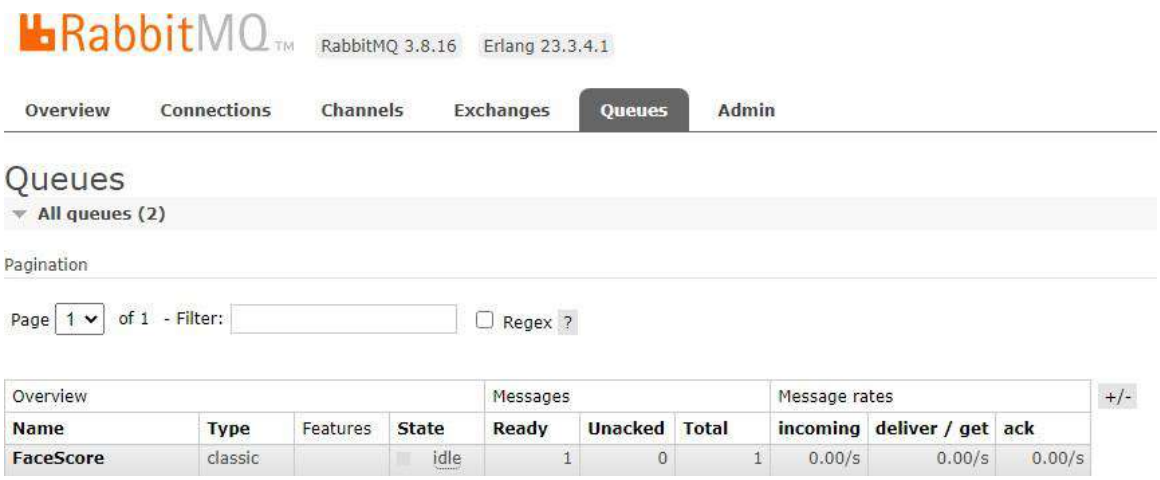
```
[HttpPost]
0 references
public IActionResult PostMessage([FromBody] MessageModel message)
{
    using (var connection = _factory.CreateConnection())
    {
        using (var channel = connection.CreateModel())
        {
            var stringfiedMessage = JsonConvert.SerializeObject(message);
            var bytesMessage = Encoding.UTF8.GetBytes(stringfiedMessage);

            channel.BasicPublish(
                exchange: "",
                routingKey: _queue.PrefixForRetriable,
                basicProperties: null,
                body: bytesMessage);
        }
    }

    return Accepted();
}
```

Nesta controller esta carregando os arquivos de configuração para poder se conectar no Rabbit para fazer a conexão e também carrega a configuração da fila, para poder publicar.

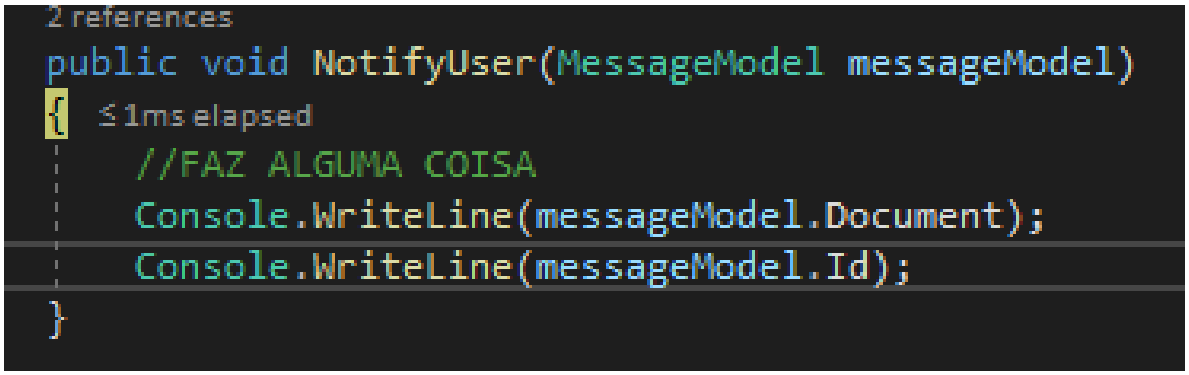
Fiz uma chamada via postman, conforme a imagem abaixo.



E confirmamos o recebimento da publicação na interface do Rabbit



E neste exemplo o consumer recebeu a informação que disponibilizei conforme a imagem abaixo.



```
19333108041
00000000-0000-0000-0000-000000000000
```