**.NET6 API + GraphQL + EF Core + Bogus**

**O que você vai encontrar neste artigo…**

Eu já estava a um bom tempo para falar sobre GraphQL, a ideia é mostrar de forma mais real possível a utilização desta tecnologia.

Ao longo da criação do projeto de exemplo eu acabei incluindo coisas muito bacanas, que se encaixam muito bem aos projetos de verdade, portanto espere encontrar bons exemplos com as seguintes tecnologias:

***(o código fonte estará disponível no GitHub e o link está no final deste artigo.)***

* Bogus:
* HotChocolate
* Entity Framework Core
* Injeção de Dependência
* Paginação com GraphQL
* Queries
* Mutations

**Um pouco da história do GraphQL**

*Mas o que é o GraphQL?*

*“GraphQL é um idioma de consulta para API’s e um tempo de execução para cumprir essas consultas com seus dados existentes. O GraphQL fornece uma descrição completa e compreensível dos dados em sua API, dá aos clientes o poder de pedir exatamente o que precisam e nada mais, facilita a evolução das API’s ao longo do tempo e permite criar poderosas ferramentas de desenvolvedor.”*

Esta é a definição formal de GraphQL, que inclusive está lá no site oficial ([GraphQL](https://graphql.org/)), mas voltando um pouco no tempo, foi em 2012 que surgiu, a partir de um projeto interno do Facebook, mas foi de fato lançado publicamente em 2015. Já em 2018 foi finalmente transferido para o a GraphQL Fundation, uma organização Linux sem fins lucrativos, seu criador Lee Byron afirma que o produto tem seguido seu objeto desde o início, que é ser onipresente em todas as plataformas web. O projeto aconteceu na verdade porque a Apple concentrou todos os seus recursos em aplicativos móveis nativos, enquanto o google não se preocupava muito com a experiência do utilizador quanto app’s web/mobile. Neste mesmo ano o Facebook construiu seu aplicativo apostando muito em HTML5, (grande erro segundo Mark Zuckerberg), a transição do mercado web para mobile não estava maduro o suficiente. O Facebook então contratou muitos engenheiros seniores de iOS e com isso decidiu reescrever o aplicativo todo para iOS, começando pelo feed de notícias.

Como padrão utilizaram uma API RESTful, e isso trouxe diversos problemas:

* **Lentidão na rede**: a API não dava conta de retornar todos os dados, então o cliente era forçado a fazer mais e mais solicitações para diferentes API’s
* **Problema de documentações**: Por conta da complexidade e quantidade de implementações, a documentação por vezes ficava desatualizada
* **Manter a Aplicação**: Sobre as respostas de API, caso a API mudasse seu formato de resposta, o código do cliente deveria mudar de acordo. Os engenheiros tiveram de manter manualmente as classes de modelos do cliente, a lógica da rede e outras coisas para garantir que os dados certos são carregados no momento certo antes de renderizar a exibição.

**Vantagens e Desvantagens?**

Mas o que realmente é o GraphQL? Quais são suas vantagens, desvantagens?

***A resposta é muito rápida, GraphQL é uma linguagem de consulta que é usada para buscar apenas os dados que o cliente deseja do banco de dados.***

**Sobre Vantagens**

* **Rápido na rede:** Ao contrário da abordagem RESTful, o GraphQL traz apenas o necessário para o cliente, isso diminui a quantidade de chamadas e o tamanho das requisições.
* **Tipos estáticos robustos:** Permite que os clientes saibam quais dados estão disponíveis e que tipo de dados são.
* **Capacitando a evolução do cliente:** O formato da resposta, é controlado inteiramente pela consulta do cliente. Assim, o código do lado do servidor torna-se muito mais simples e fácil de manter. Quando você remove campos antigos de uma API, esses campos serão preteridos, mas ainda assim continuarão funcionando. Este processo gradual de compatibilidade retrógrada remove a necessidade de versionamento. Depois de tantos anos, o Facebook ainda está na versão 1 de sua API GraphQL.
* **Documentação:** A documentação é gerada automaticamente e sempre atualizada.
* **Código arbitrário de consulta:** O GraphQL é adaptável, não se trata de bancos de dados. Ele pode ser adotado em cima de uma API RESTful existente e pode trabalhar com ferramentas de gerenciamento de API existentes.

**Sobre Desvantagens**

* Curva de aprendizado, se você já conhece o REST, possivelmente pode ter algumas dificuldades em sua implementação.
* O armazenamento em cache é mais difícil com GraphQL.
* As consultas sempre retornam código 200.

**Chega de falar! Mãos a obra!!!**

1 – Vamos criar um projeto de API

Graphical user interface, text

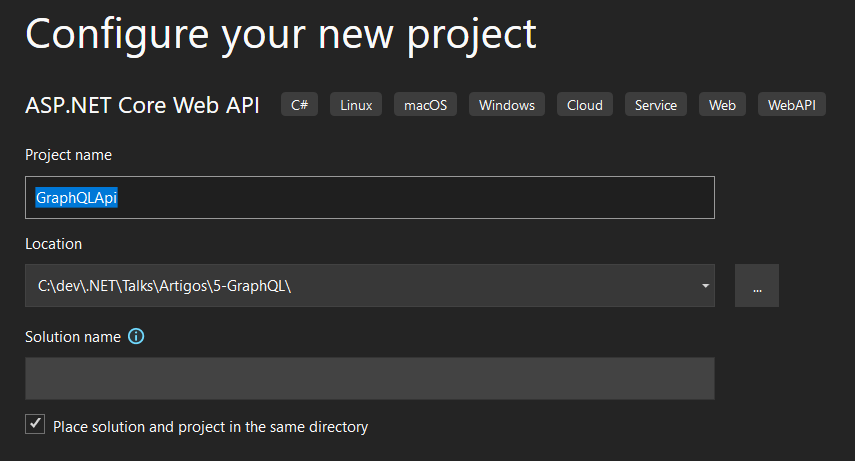
Description automatically generated

2 – Selecione o tipo de projeto ASP.NET Core Web API

A screenshot of a computer

Description automatically generated with medium confidence

3 – Informe um nome para sua solução



4 – Selecione a versão .NET 6.0 LTS e mantenha demais opções como mostra a imagem e clique em Criar

A screenshot of a computer

Description automatically generated with medium confidence

5 – A estrutura da solução deverá ficar desta forma

A screenshot of a computer

Description automatically generated with medium confidence

6 – Crie as seguintes pastas

A picture containing graphical user interface

Description automatically generated

“**Data**” é onde criaremos as classes que vão popular nosso banco de dados SQL Server (seeds). Esta pasta também deverá ter a classe de contexto.

“**GraphQL**” os arquivos de “***Queries”*** e “***Mutations”*** que logo mais falaremos sobre.

“**Interfaces**” como o nome já sugere, criaremos uma interface para tratar dos nossos serviços.

“**Migrations**” criada automaticamente quando corremos a migração.

“**Models**” onde devemos criar nossas classes concretas dos modelos.

“**Services**” deverá conter a classe concreta de serviço, que deve implementar a interface anteriormente criada.

“**ViewModels**” apenas uma classe de transporte.

7 – Agora vamos já instalar os pacotes necessários para este projeto

Install-Package Bogus

Install-Package HotChocolate.AspNetCore

Install-Package HotChocolate.Data.EntityFramework

Install-Package Microsoft.EntityFrameworkCore

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

Install-Package Microsoft.EntityFrameworkCore.Abstractions

Install-Package Microsoft.EntityFrameworkCore.Design

Install-Package Microsoft.EntityFrameworkCore.Relational

8 – Criando as classes “**Customer**” e “**Invoice**” na pasta “**Models**”

Primeiro vamos criar as duas classes que representarão nosso modelo, o objetivo é ter uma classe chamada “***Customer***” e outra “**Invoice**”, a última é uma relação de faturas por cliente, simularemos vários clientes, que por sua vez possuem sequência randómica de faturas, isso será criado automaticamente utilizando alguns ótimos recursos (chegaremos lá).

**Classe Customer**

Crie primeiro uma interface na pasta “**Interfaces**” chamada “**IAggregateRoot**”.



Temos dois construtores na classe “**Customer**”, o primeiro é utilizado meramente para o Entity Framework.

Podemos ver também que a classe faz referência a “**Invoice**”, que criaremos na próxima etapa, somente com “**Customer**” será mostrado um erro, até que a classe “**Invoice**” seja criada.

Fiz uso de algumas técnicas importantes:

* Vejam que a classe “**Customer**” implementa uma interface chamada “**IAggregateRoot**” onde sua única função é ser uma interface de marcação. É uma boa prática e muito aplicada em projetos que utilizam o design DDD, serve para mostrar que “**Customer**” é uma raiz de agregação, essa técnica garante coesão, expressividade e integridade do domínio uma vez que o acesso a essas entidades só pode ser feito a partir da entidade raiz, que no nosso caso é “**Customer**”, importante levar em consideração que é apenas uma demonstração, na prática esta técnica aplica-se a projetos maiores que possuem mais contextos (Achei por bem compartilhar).
* Outra importante característica é a configuração das propriedades da classe decorando as mesmas com validações, fazendo uso de Data Annotations conseguimos diretamente na classe criar validações que vão nos ajudar a definir quais campos são obrigatórios, ranges, tamanho de campos, comparações de datas, regex e muito mais.
* Todos os nossos “set’s” são privados, escondido do mundo externo, dando a responsabilidade à própria classe para a criação de uma nova instância, desta forma garantimos o estado da nossa classe e somente será possível alterar a partir dos métodos providos por ela.

Aproveitei para indicar a chave primária e a relação que teremos de 1: N com a classe “**Invoice**”. Eu particularmente não crio desta forma, em projetos maiores faz sentido criar as classes de mapeamento devidamente separadas, além do mais a classe concreta fica com o código mais limpo e fácil de entender.

Text

Description automatically generated

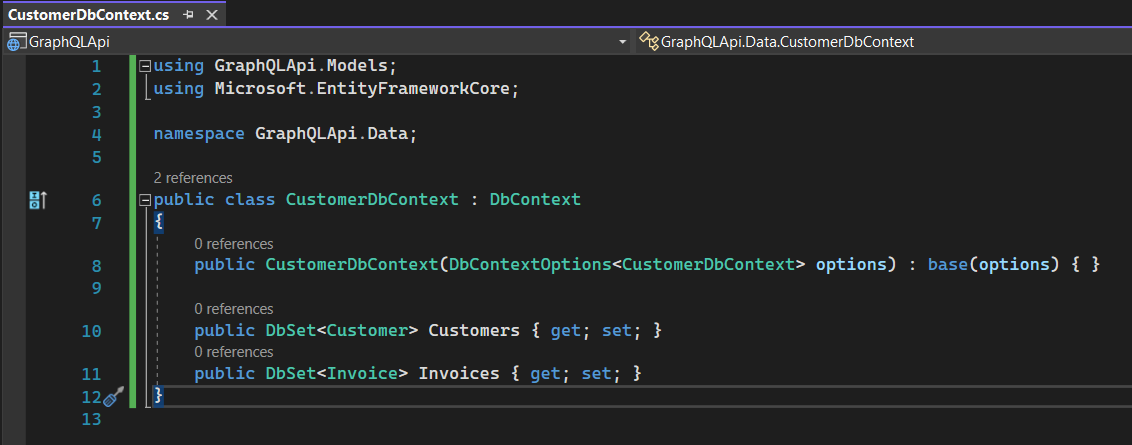
**Classe Invoice**

Muito parecida com a classe “**Customer**” em relação a estrutura. A linha 35, 36 e 37 referenciam a classe “Customer” para efeitos de criação das tabelas e seus relacionamentos utilizados no Entity Framework.

Text

Description automatically generated

9 – Crie a classe “**CustomerDbContext**” dentro da pasta “**Data**”, ela vai herdar de “**DbContext**” que faz parte do Entity Framework, basicamente ela representa um tipo de sessão com o banco de dados e desta forma nos permite fazer consultas, salvar instâncias etc…



10 – Criação de Dados Automáticos com Bogus

Um recurso que uso muito em projetos é o pacote “**Bogus**”, ele nos ajuda a criar dados fictícios de forma muito inteligente e eficiente, possui uma série de métodos que atendem a todo o tipo de geração de dados, como por exemplo dados financeiros, datas, dados humanos como data de nascimento, gênero, nomes, dados matemáticos e por aí vai…

Crie uma pasta dentro da pasta “**Data**” e chame ela de “**Seeders**”, dentro desta última pasta crie uma classe chamada “**DbInitializer**”.

Veja que o método “Seed” vai criar uma instância de “**Customer**” e na linha 23 informamos quantos registos pretendemos criar, e de forma aleatória ele vai criar para você uma série de “**Customers**”, “**Invoices**“ funciona da mesma forma, com um detalhe importante, na linha 35 o método “**PickRandom**” vai obter um “**Customer**” já criado para relacionar as “**Invoices**”.

Essa é uma ótima ferramenta para usar em testes unitários.

Text

Description automatically generated

Agora voltamos na classe “**CustomerDbContext**” e incluímos a chamada para a classe “**DbInitializer**”, na sobrecarga do método “**OnModelCreating**”, que é executado toda vez que fazemos o update-database de uma migração.

A screenshot of a computer

Description automatically generated

10 – Registar as funcionalidades na API

Abra a classe “**Program.cs**” e inclua estas linhas, aqui estamos configurando nosso DbContext na aplicação. Mais tarde vamos voltar a alterar esta classe.

A screenshot of a computer

Description automatically generated

Perceba que a linha 7 faz referência a string de conexão que será utilizada para persistência de dados, portanto abra o arquivo “**appsettings.json**” e registe a sua.

Text

Description automatically generated

11 – Adicionar a migração – Criando Banco de Dados + Tabelas

Se tudo correu bem até agora, este é o momento certo para correr dois comandos, o primeiro vai criar a migração com base nas “**Models**” e o seguinte vai de fato executar os comandos SQL para nosso banco de dados.

Portanto no Package Manage Console correr o primeiro comando:

1 – Comando principal

2 – O nome que identifica a migração, no meu caso utilizei “initial”

3 – Parâmetro que identifica o contexto

4 – Nome do contexto

5 – Significa Verbose, não obrigatório, eu sempre utilizo, pois, o processo todo será mais detalhado no output

Text

Description automatically generated

Se correu bem deverá ver um arquivo parecido com este, é gerado automaticamente a cada migração e contém todos comandos necessários para criação/alteração do nosso banco de dados e respetivas tabelas, índices etc…

Text

Description automatically generated

O segundo comando é o que vai de fato executa os comandos na base de dados.

1 – Comando principal

2 – Parâmetro que identifica o contexto

3 – Nome do contexto

4 – Significa Verbose, não obrigatório, eu sempre utilizo, pois, o processo todo será mais detalhado no output

A screenshot of a computer

Description automatically generated with medium confidence

Este último é muito importante, pois, é neste momento que a classe “**DbInitializer**” será chamada para criar os nossos dados “fakes”, tudo isso automaticamente, abra seu gerenciador de SQL Server e veja se tudo foi criado corretamente.

Podemos ver que foram criadas 3 tabelas, a primeira da imagem é de controle do próprio Entity Framework, a segunda é a Customers e a terceira é a Invoices.

Graphical user interface, application

Description automatically generated

Os dados já foram criados conforme imagem

Text, table

Description automatically generated

12 – Criação de Serviços

Vamos agora criar uma interface “**ICustomerService**” que deverá conter os métodos para persistência dos nossos dados.

A screenshot of a computer

Description automatically generated

Na sequência vamos criar o serviço que deverá implementar esta interface.

Primeira parte, implementamos os métodos para incluir “Customer” e “Invoice”

A screenshot of a computer

Description automatically generated with medium confidence

Depois os métodos de obter, editar e deletar dados.

Text

Description automatically generated

13 – Registar os serviços em “**Program.cs**”

Text

Description automatically generated

14 – Agora finalmente vamos ao tema, GraphQL

Existem duas operações primárias no GraphQL

* Query => Serve para pesquisas de dados
* Mutation => Serve para alterações de dados

***Query***

Na prática a ***Query*** funciona de forma fluída, você vai informar os campos que pretende utilizar, mas sem a necessidade de ter no servidor uma view model para cada serviço que pretende consumir e retornar valores

Ex.: A esquerda mostra um tipo de consulta onde por mais que o serviço possa retornar 10 campos, eu informo somente os que de fato vou precisar.

A direita a lista de “pets” somente com os campos que solicitei.

Timeline

Description automatically generated

Este outro exemplo mostra como podemos buscar um dado pelo código

Graphical user interface, text, application, Teams

Description automatically generated

***Mutations***

As mutações, como o nome sugere, é responsável por alterações de dados, sempre que quisermos editar, incluir ou deletar um dado, vamos utilizar este tipo.

Ex.: Informamos que queremos utilizar uma mutation, depois basta adicionar o método previamente configurado e o input de valores, isso já bastaria para fazer a inclusão deste dado.

Text

Description automatically generated

15 – Criando nossas queries

Crie uma pasta chamada “**Queries**” dentro da pasta “**GraphQL**”, logo em seguida crie uma classe chamada “**CustomerQuery**”, devendo ficar assim:

Text

Description automatically generated

Primeiro vamos fazer a injeção de dependência do serviço que retorna os dados, depois crie 3 métodos, sendo o primeiro para retornar todos os “**Customers**”, o segundo deverá obter um “**Customer**” por Id e o terceiro vamos retornar “**Customers**” informando um nome.

Text

Description automatically generated

16 – “**Program.cs**”, registar GraphQL

Agora precisamos registar o GraphQL em “**Program.cs**”

Text

Description automatically generated

17 – Altere configuração do projeto

Altere a configuração nas propriedades do projeto, clicando com botão direito no projeto, última opção “Properties” e busque por env.

Desta forma a abertura será a partir da página

A screenshot of a computer

Description automatically generated

E execute sua aplicação, deverá ter uma resposta como esta:

Graphical user interface, application

Description automatically generated

18 – Vamos a nossa primeira consulta

Clique em “**Create Document**” e escreva a seguinte Query…

Text

Description automatically generated

E “***Voilà***” temos nossa primeira Query com sucesso!!!

Algumas características importantes:

Podemos verificar que eu informo “**query**” e logo na sequência eu posso dar um nome para esta query, no meu caso “**FindAll**”, digo que quero somente “id”, “name” e “createdAt”, invoices a mesma coisa.

Na lateral direita já conseguimos ver o resultado, a lista de “**Customers**” e suas respetivas “**Invoices**”.

Text

Description automatically generated

19 – Agora podemos pesquisar por “Nome”, exemplo:

Pesquisando somente “**Customers**” que contenham no nome “**Carlos**”, primeira imagem é a pesquisa e logo abaixo podemos ver o método que foi utilizado no server. Veja que agora estou trazendo somente 3 campos de cliente.

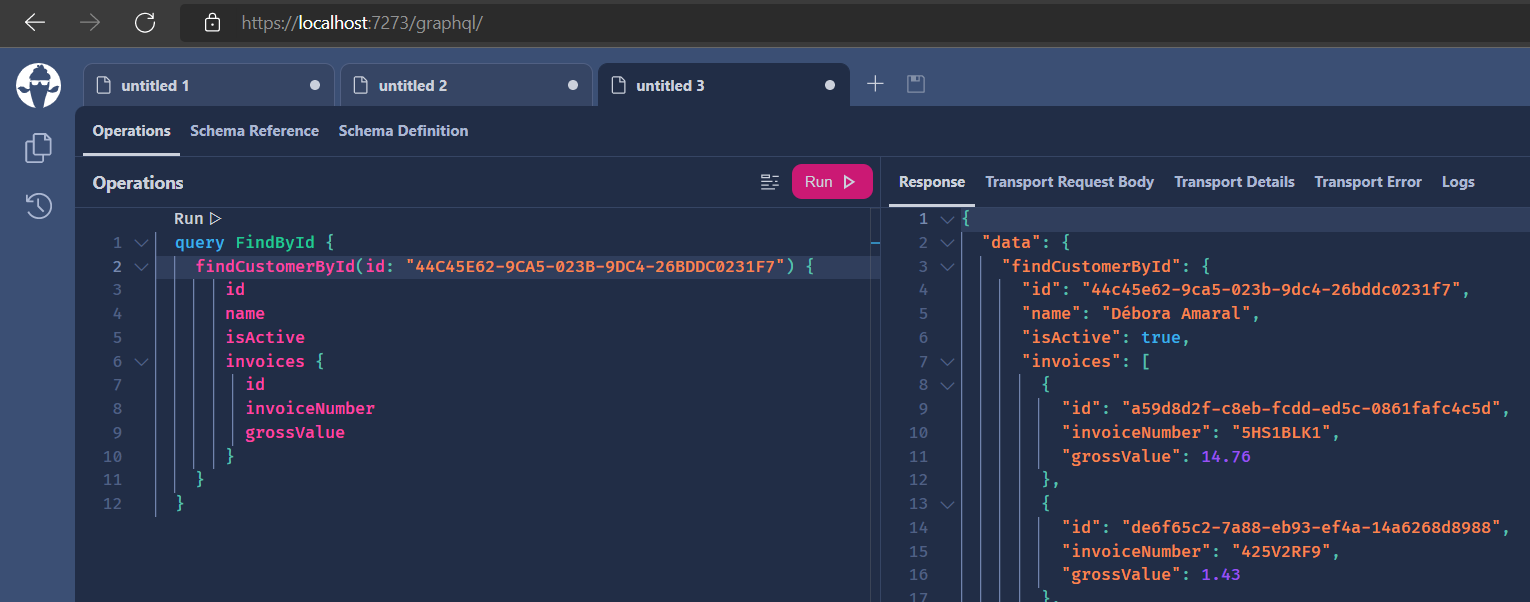
Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

20 – Vamos agora pesquisar por Id, obtenha um Id aleatório na tabela “**Customer**”, veja que trouxe somente dados referente ao “Customer” com o Id = “44C45E62-9CA5-023B-9DC4-26BDDC0231F7”



21 – ***Agora uma dica valiosa***, vamos implementar uma paginação para conseguir visualizar nossos “**Customers**”, crie um método conforme abaixo:

Text

Description automatically generated

Decoramos nosso método primeiro com o atributo que nos ajuda aceder o contexto para o GraphQL, logo sem seguida mais um atributo que configura o uso de paginação, existem 2 tipo de paginação:

1 – UsePaging => middleware retorna um cursor que deve ser utilizado posteriormente para cada chamada, desta forma ele segue para as próximas páginas.

2 – UseOffsetPaging => Modelo mais tradicional, onde são informados e retornados metadados contendo número da página, tamanho da página, entre outros que veremos.

Execute a aplicação e faça uma nova query desta forma:

Text

Description automatically generated

Observe na sequência

1 – Informe os parâmetros skip e take que respetivamente representam a página e o número de itens por página

2 – O PageInfo é fornecido pelo GraphQL

3 – Items é a lista de itens retornados com base nos parâmetros informados

4 – O retorno com PageInfo preenchido

5 – No console podemos ver a query que foi executada no SQL Server, **retorna o que precisamos? Sim. Está correto? Não**

Ainda temos um problema, imagina uma tabela com 50 mil itens, mesmo implementando a paginação, a mesma só é aplicada depois que a consulta já foi feita na base de dados, portando da forma como está implementado, para cada requisição, todas as vezes será obtido 50 mil registos para depois paginar, isso com certeza trará grande dor de cabeça com relação a performance. Isso ocorre porque utilizamos um IEnumerable como forma de pesquisa, ou seja, primeiro a consulta é feita na base de dados, e posteriormente filtrada posteriormente em memória.

22 – Resolvendo o problema da paginação

Altere o método para que fique assim:

Text

Description automatically generated

Veja que passou a retornar um IQueryable, que faz a consulta à base de dados já passando filtros/parâmetros de paginação, trazendo de fato somente os dados que precisamos.

O resultado é o mesmo, só que podemos ver que o comando no SQL Server já recebe os parâmetros diretamente na query.

Text

Description automatically generated

23 – Vamos criar as nossas mutações

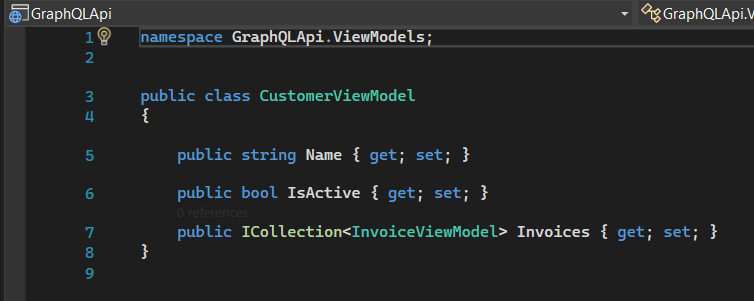
Relembrando, as “mutations” são utilizadas quando precisamos alterar o estado das nossas entidades, seja para edição/inclusão/remoção.

A primeira coisa é criar uma pasta chamada “**Mutations**” dentro da pasta “**GraphQL**”. Dentro da nova pasta criamos a classe “**CustomerMutations.cs**”.

Graphical user interface

Description automatically generated

Dentro da pasta “**ViewModels**” devemos criar duas View Models, “**CustomerViewModel**” e “**InvoiceViewModel**”, utilizado para receber os parâmetros das requests.

 A screenshot of a computer

Description automatically generated with medium confidence

A classe “**CustomerMutation**”, assim como a Query, vai conter o código responsável por consumir o serviço “**CustomerService**”.

Então podemos ver que possuímos aqui os métodos “**AddCustomerAsync**”, “**UpdateCustomerAsync**” e “**DeleteCustomerAsync**”.

Text

Description automatically generated

24 – Registar a mutation

Novamente no “**Program.cs**” vamos incluir o seguinte código:

Altere seu AddGrapQLServer para conter a sua Mutation.

Text

Description automatically generated

25 – Adicionar um registo

Graphical user interface, application, Teams

Description automatically generated

Incluir uma Invoice

Graphical user interface, text

Description automatically generated

26 – Alterar um registo

Graphical user interface, text, application

Description automatically generated

27 – Remover um registo

A screenshot of a computer

Description automatically generated with medium confidence

***Conclusão***

GraphQL é uma ótima opção para construção de API’s, realmente o desempenho é ótimo, temos menos arquivos no projeto, tendo em vista que não precisamos criar vários endpoints. Permite que o cliente (Mobile, Web, Angular React)

tenha mais liberdade quanto ao uso e tratamento de queries, e não fique tão dependente do lado de servidor. Entender o tipo de projeto e se seu time está preparado para performar com GraphQL como faz com REST também é um fator de atenção.