* Definindo as entidades

-> Criar a pasta Models

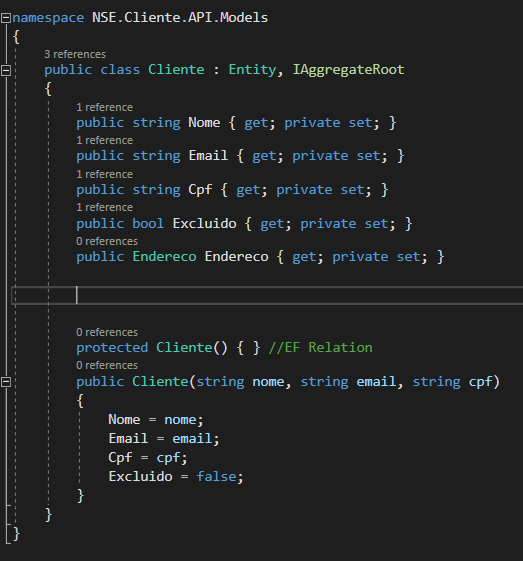
-> Criar e modelar a classe de clientes

->Herdar de Entity e de IAggregateRoot

-> Configurar a relação dela com o Entity e com a classe de

Endereços

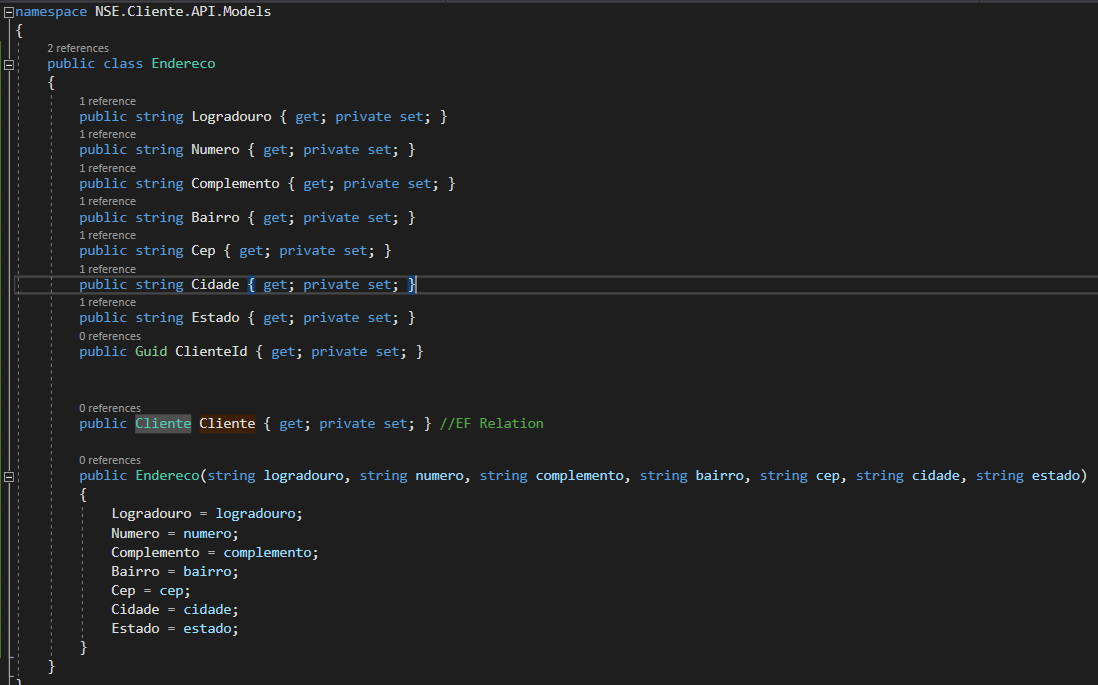
-> Configurar construtor da própria classe como também o que será usado apenas como relação com o Entity.



-> Criar e modelar a classe de Endereço

-> Configurar a relação dela com classe de clientes

-> Configurar construtor



* Objetos de valor e validações

-> No building blocks/core/NSE.Core/DomainsObjects, criar as classes cpf e email para que sempre seja possível verificar essas duas variáveis

-> No objeto de valor do CPF:

-> Verificar o seu tamanho máximo e se contêm apenas números.

-> Construtor da classe

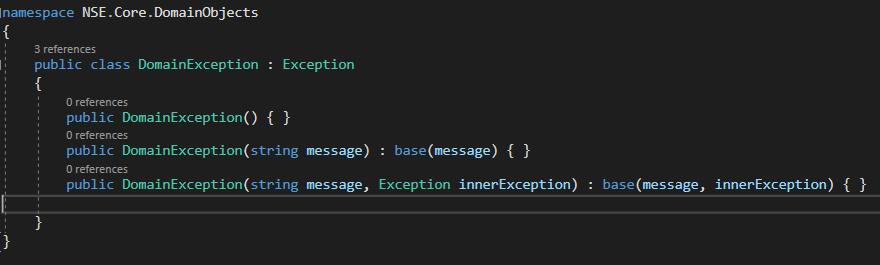
-> Construtor do Entity

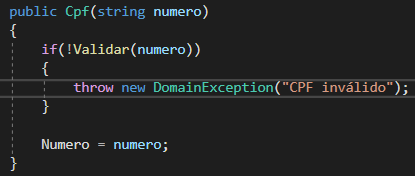
-> Caso esse construtor não seja feito dentro do objeto de valor, na hora que isso for mapeado para o banco e for materializado da consulta do banco para o objeto, o Entity Framework não saberá construir.

-> Criar um código de validação de CPF

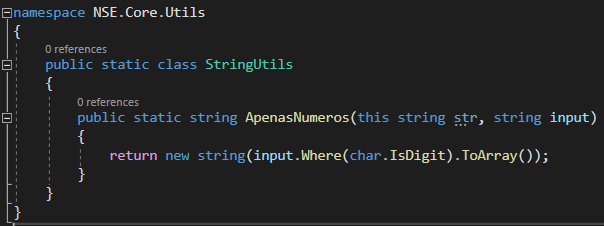


-> Não permitir que a entidade seja montada de forma inconsistente. Para isso deve-se criar uma classe DomainException e utilizá-la no construtor.





-> Dentro da NSE.Core, criar uma pasta Utils e nela criar uma classe estática que retorne apenas os números da string CPF.



-> No objeto de valor Email

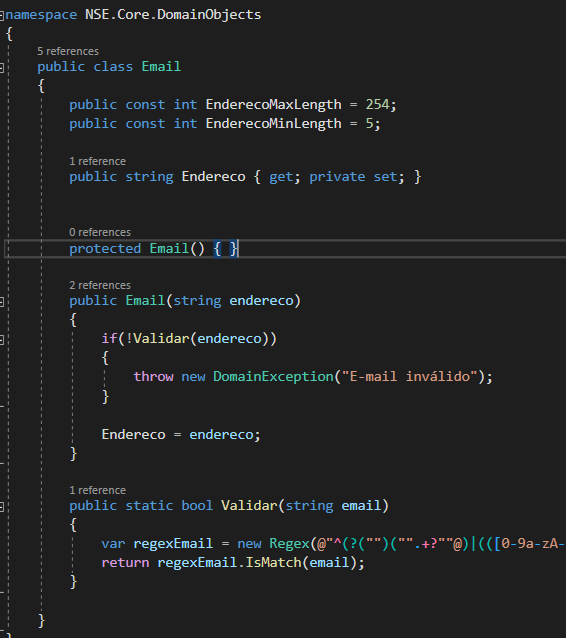
-> Verificar seu tamanho máximo e mínimo

-> Construtor da classe

-> Retornar uma exception caso o email não seja válido.

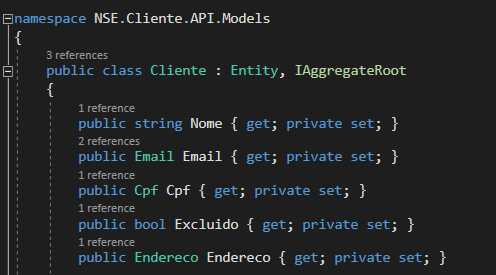
-> Construtor do Entity

-> Método de validação do email.

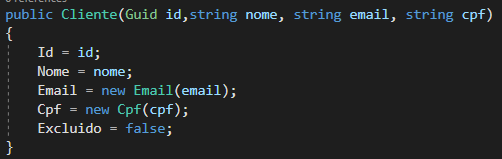


-> Na api de cliente, na model Cliente:

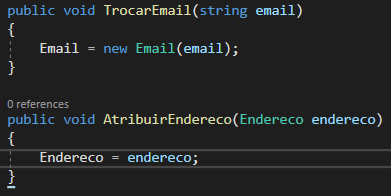
->Mudar as variáveis Cpf e Email do tipo string para o tipo Cpf e Email respectivamente.



-> No construtor, para o email e para o cpf, deve criar uma nova instância para ambos. Adicionar um guid id no construtor do Cliente. Alimentar o id dentro do construtor.



->Criar métodos de trocar e-mail e atribuir endereço

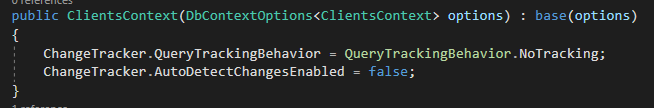


* Mapeamento das Tabelas

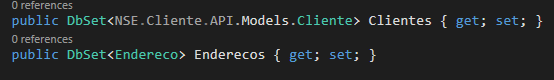
->Dentro da pasta Data, criar o ClienteContext.

-> Deve herdar de DbContext e IUnitOfWork

-> Desabilitar os tracking para melhor performance



-> Relacionar os DbSets

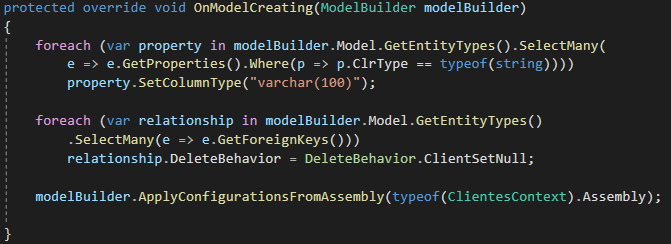


-> Dentro do OnModelCreating

-> Desabilitar o delete cascade

-> Realizar verificação caso algum campo da tabela não tenha sido mapeado anteriormente. Caso não, aplicar o tipo varchar.

-> Aplicar as configurações



-> Dentro da pasta Data, criar a pasta Mappings. Dentro dessa nova pasta:

-> Criar a classe ClienteMapping e mapear a entidade de cliente

-> Criar a classe EnderecoMapping e mapear a entidade de endereço

-> Lembrar que classes de mapeamento devem herdar de IEntityTypeConfiguration<>

->Configurar a API no start up

-> Criar a pasta Configuration e separar as configurações em suas respectivas classes.

-> Configurar o DbContext na start up

-> Definir a connection string

-> Criar as tabelas do bancos através da migration

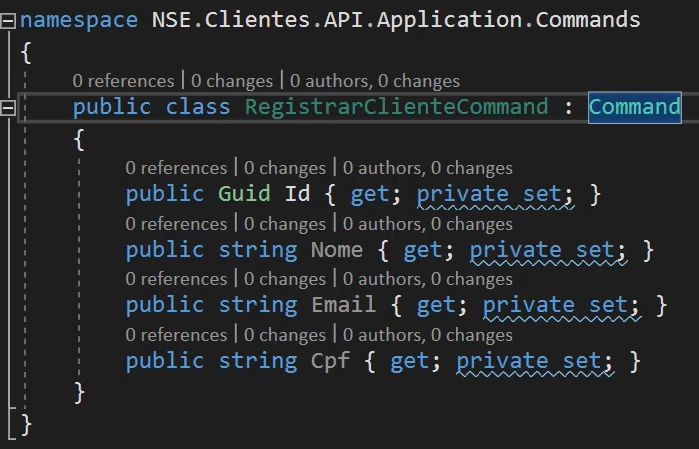
* Command e command handler

->Na api de cliente, criar a pasta Application e dentro dela criar a seguintes pastas: Commands, Events, Queries.

->Na pasta Commands, criar uma classe chamada RegistrarClienteCommand

-> Lembrando que um comando representa uma intenção de alteração do estado da entidade, seja ela uma inclusão, alteração ou remoção.

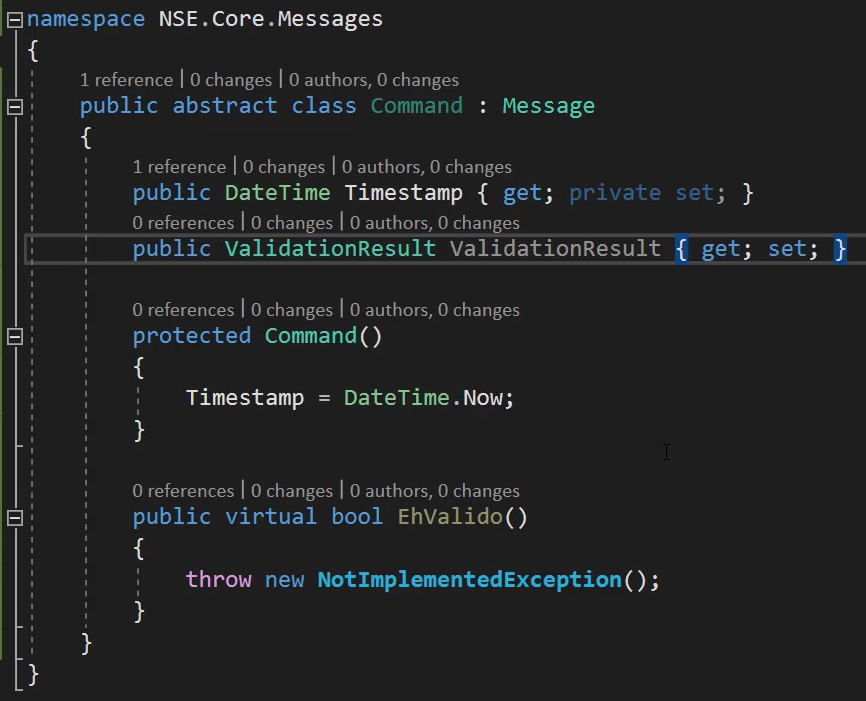
-> Essa classe deve herdar de Command, que será a próxima classe a ser criada



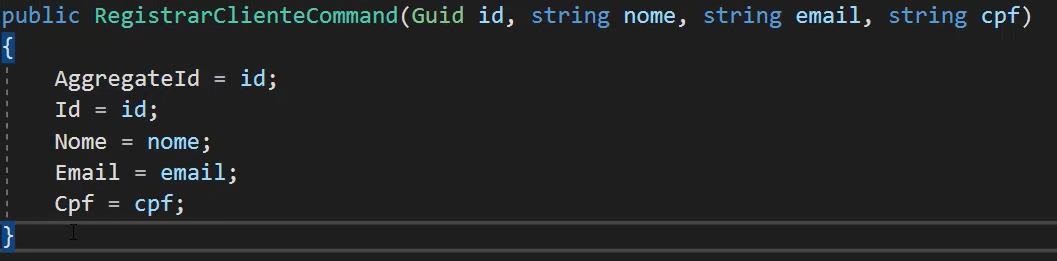
-> Na pasta srv/building blocks/core/nse.core criar a pasta Messages e dentro dela criar uma classe abstrata chamada Command, que será a classe base para RegistrarClienteCommand.

->Herdar de Menssage.cs, que será a próxima classe a ser criada.

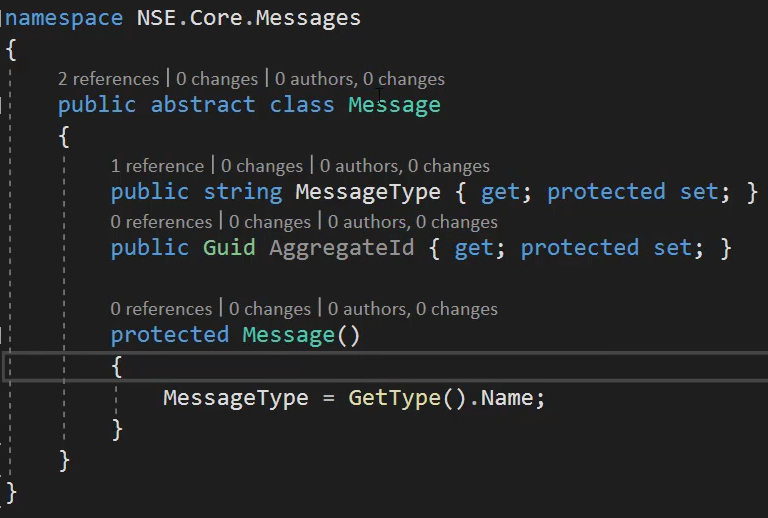
->Lembrar que também existe um data annotation com esse nome. Na hora de importar, o que deve ser utilizado é o FluentValidation.



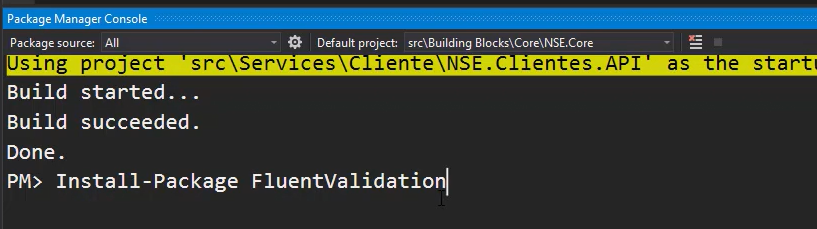
->Na API De cliente,na classe RegistrarClienteCommand, criar construtor com as propriedades necessárias, incluindo o aggreteId da classe base Message



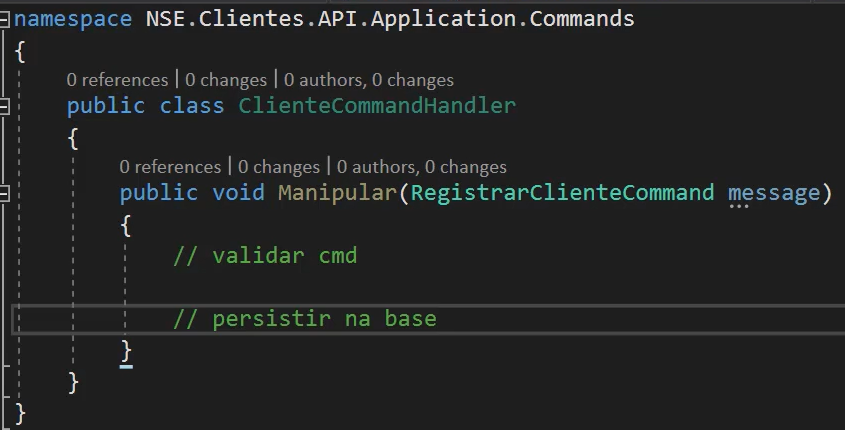
-> Em srv/building blocks/core/nse.core/Messages, criar uma outra classe abstrata chamada Message, que será a classe base para Command.



->Instalar o package FluentValidation. Esse pacote deve ser instalado no projeto NSE.Core, que fica dentro da pasta src/building blocks/core

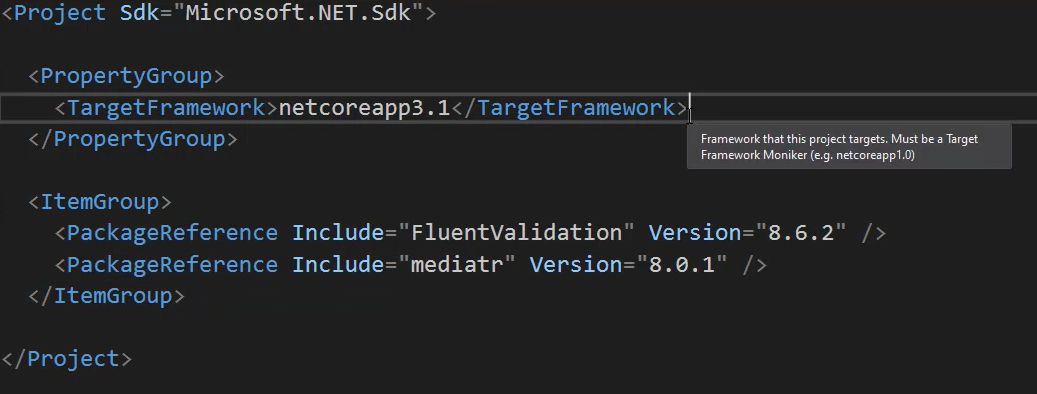


->Na API de cliente, na pasta Application/Commands, criar outra classe chamda ClienteCommandHandler, que será o manipulador do comando



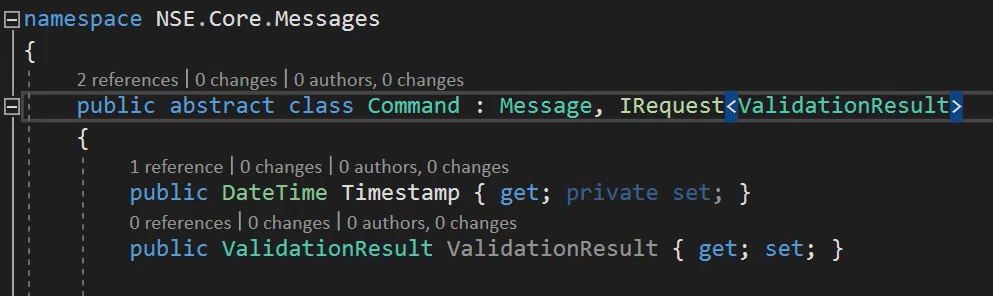
* MediatR e Command Handler

->Instalar o MediatR na classe NSE.Core na pasta building blocks



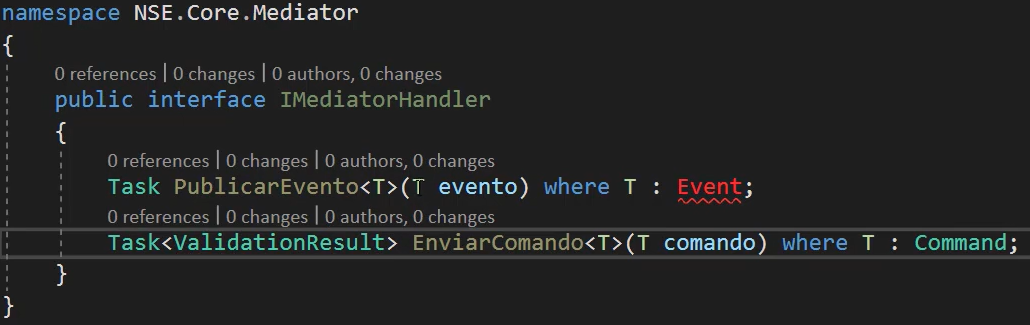
->No Command.cs, implementar o IRequest para que todos os comandos possam ser interpretados pelo MediatR.

-> O IRequest não precisa necessariamente retornar algo, mas nesse caso iremos retornar o objeto de validação.



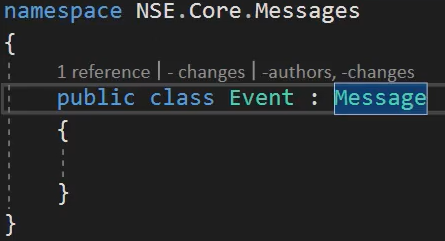
->Criar uma abstração do MediatR. Na pasta building blocks/nse.core, criar uma pasta Mediator, e dentro dela criar uma interface IMediatorHandler com os seus respectivos métodos.

->A classe base para o método PublicarEvento será criada posteriormente.



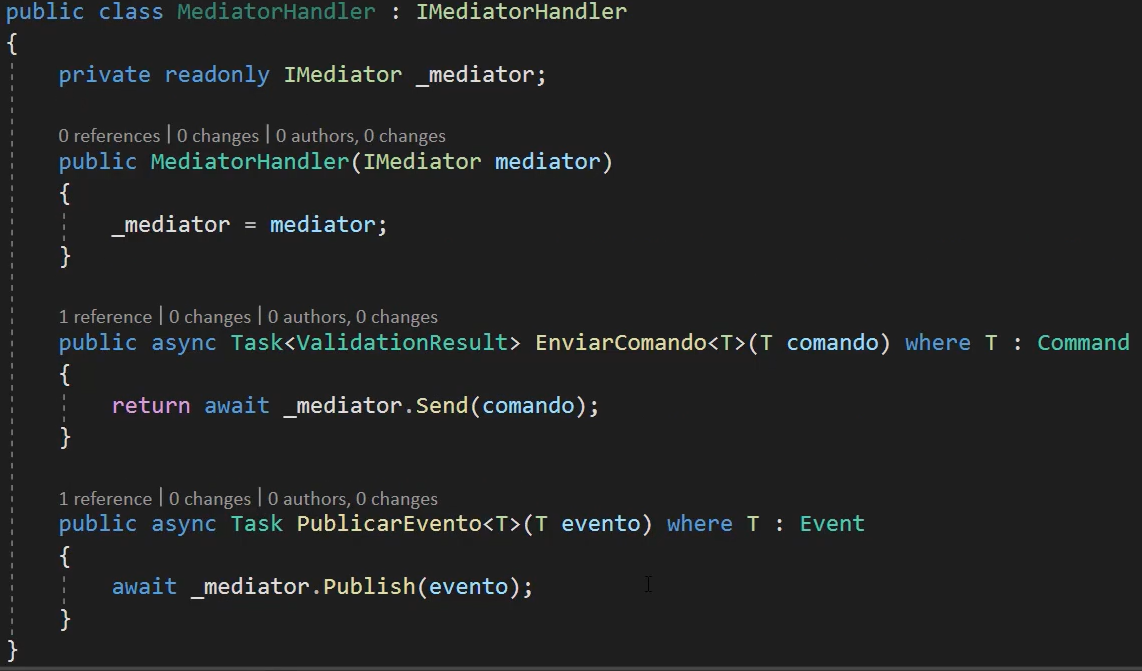
->Na pasta Messages, criar a classe Event que atuará como a classe base para o método da interface anterior.

-> Essa classe herdará de Message

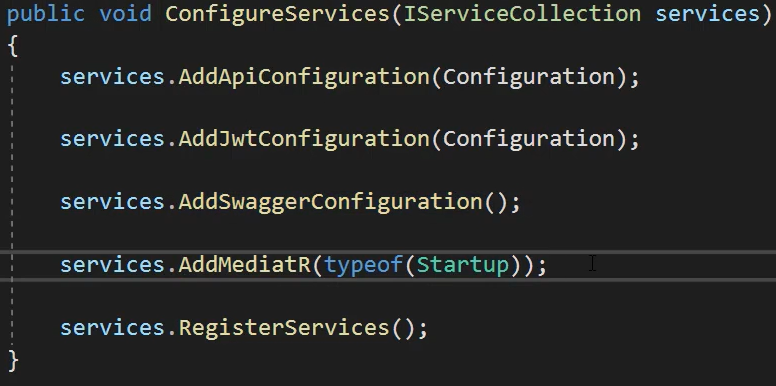


->Na pasta Mediator, criar a classe MediatorHandler que deve implementar a interface IMediatorHandler

-> Essa classe deverá também implementar o IMediator através de uma injeção de dependência



->Na API de Clientes, no seu start up, deve-se adicionar o MediatR. Para isso deve-se instalar o package MediatR.Extensions.Microsoft.DependecyInjection na NSE.Clientes.API.



->No ClienteCommandhandler, implentar a interface IRequestHandler

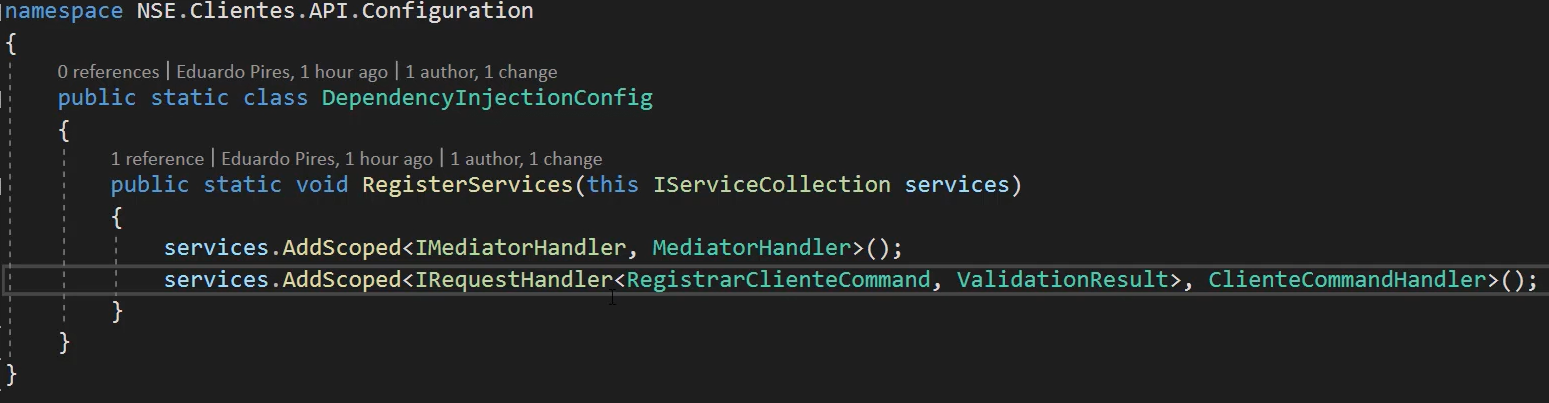
-> Lembar que a task tem de ser assíncrona



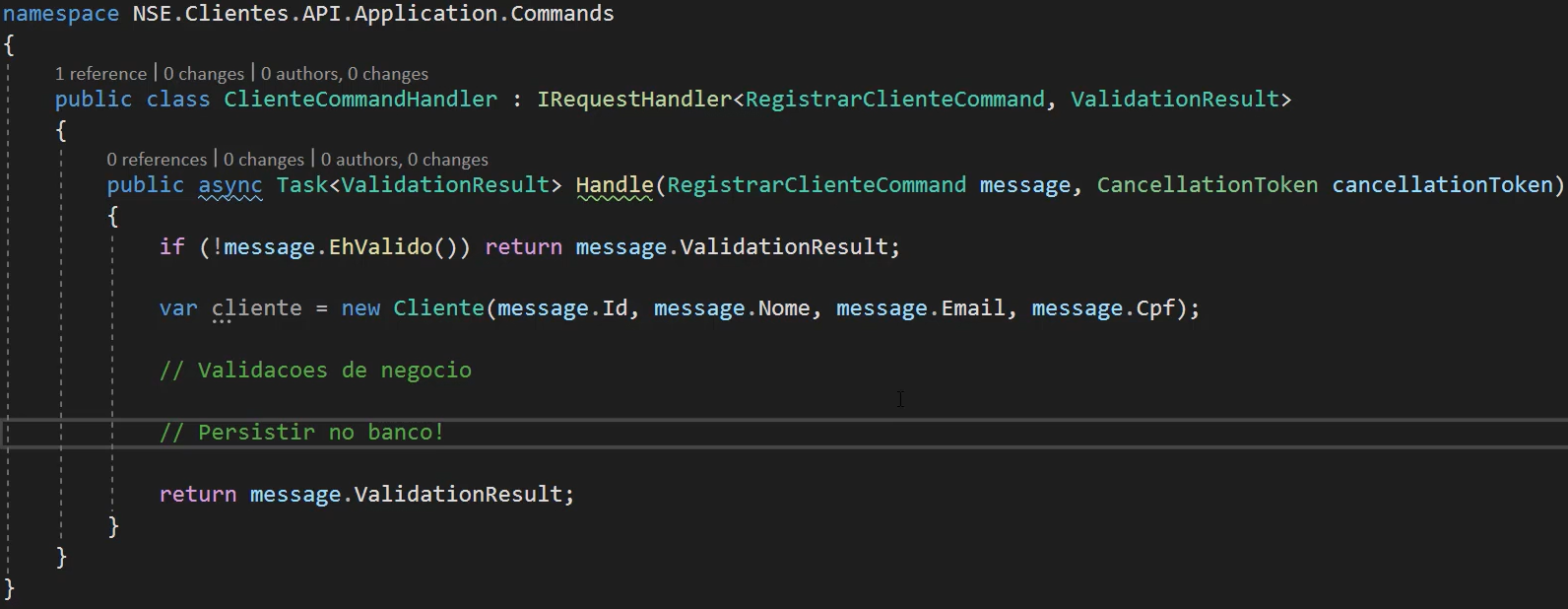
->Na API de Cliente/DependecyInjectionConfig, realizar a configuração de da injeção de dependência.

-> De início deve-se resolver o Mediator Handler

-> Depois Handler do command handler. Nesse caso o RegistrarClientCommand será entregue via IRequestHandler e retornará um ValidationResult, será manipulado pelo ClienteCommandHandler



->Agora o ClienteCommandhandler deve ser estruturado para que se possa validar todos os pontos.

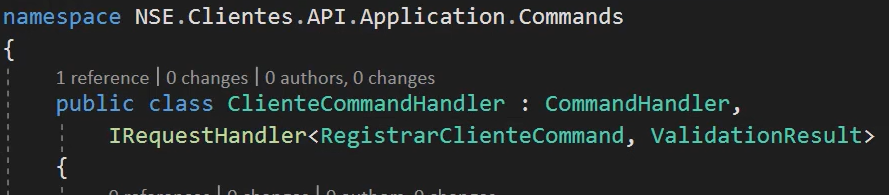


->Em nse.core/Messages, criar uma classe abstrata com suas respectivas propriedades. Já deve-se setar uma instância do validation result.

-> Adicionar o método AdicionarErros



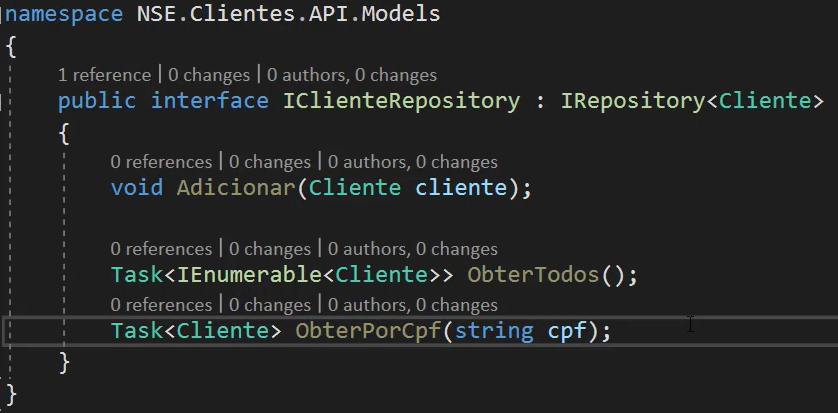
->Na API de Cliente/Application/Commands, deve fazer com que o ClienteCommandHandler herde também da classe base CommandHandler



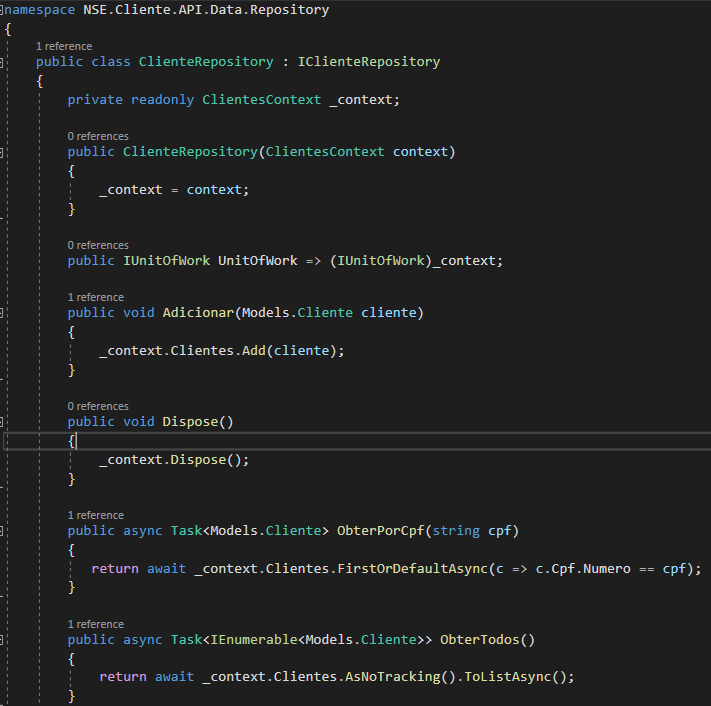
* Persistência via Command Handler

->Em NSE.Cliente.API/Models/ criar uma interface chamada IClienteRepository

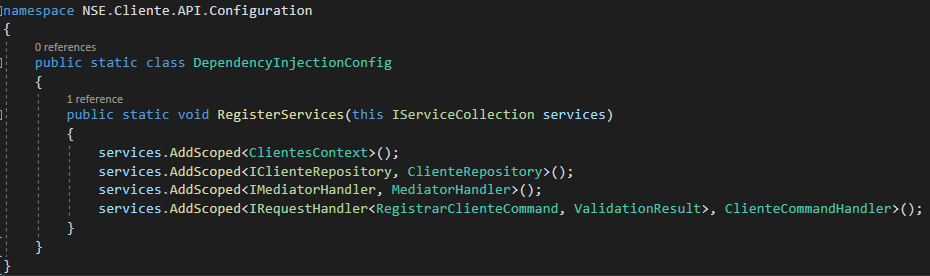
-> Implementar o IRepository<Cliente>



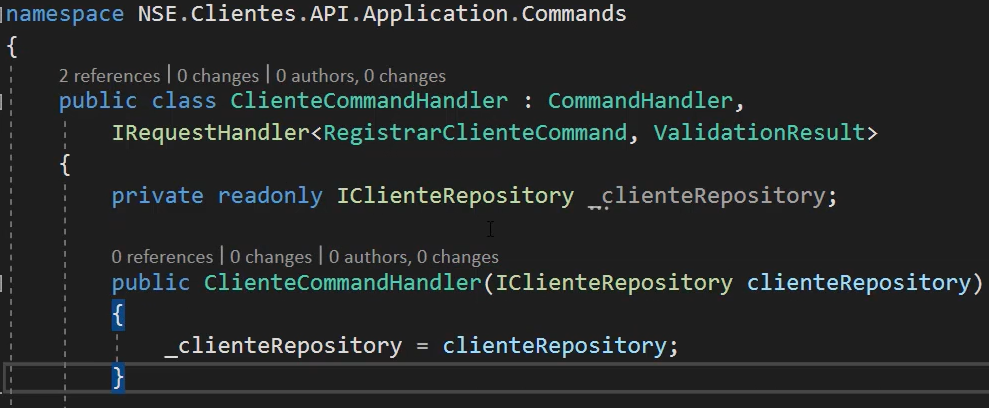
-> Na pasta Data, criar uma pasta chamada Repository e dentro dela criar uma classe chamada ClienteRepository.



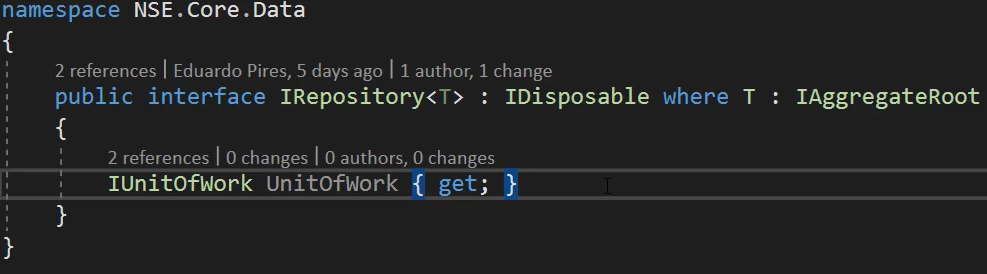
-> Em Clientes.API/Configuration/DependecyInjectionConfig, e resolver as dependências necessárias



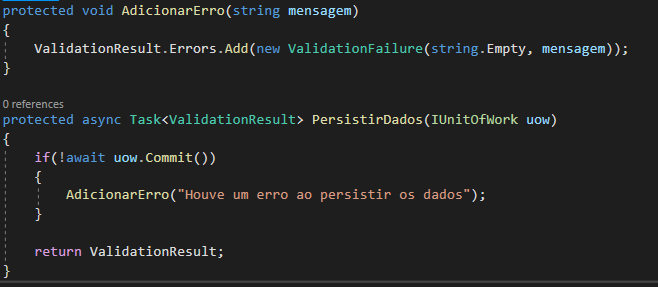
-> Injetar a dependência do repositório no ClienteCommandHandler



-> Em NSE.Core/Data/Repository, implementar o IUnitOfWork para que seja possível manipular pelo próprio repositório.



->Em NSE.Core/Messages/CommandHandler base, escrever o método PersistirDados

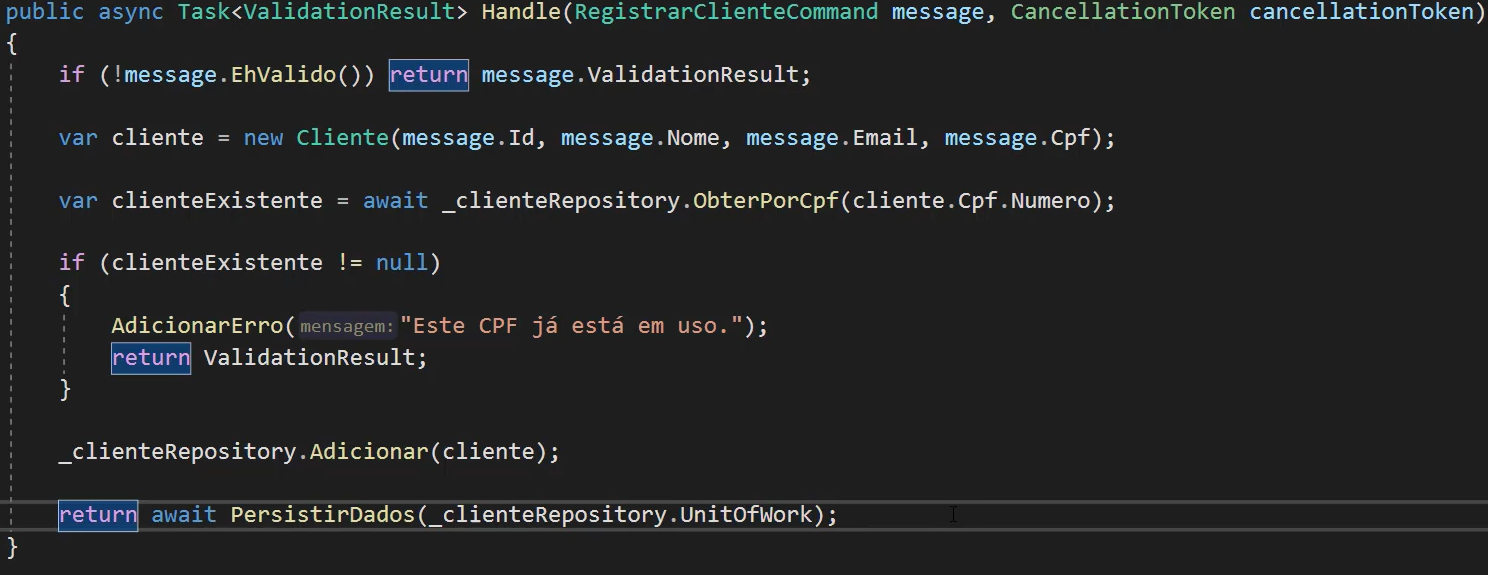


-> Em ApiCliente/Commands/ClienteCommandHandler, na task Handle, implementar:

->Verificar se o cliente existe

->Adicionar o cliente ao repositório

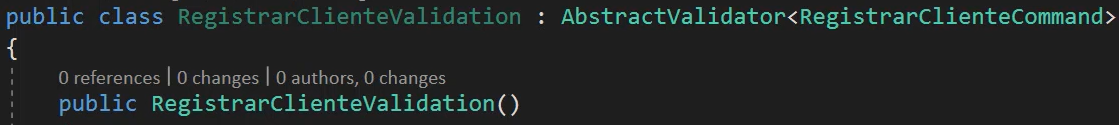
->Verificação de erro de persistência, utilizando o método do command handler



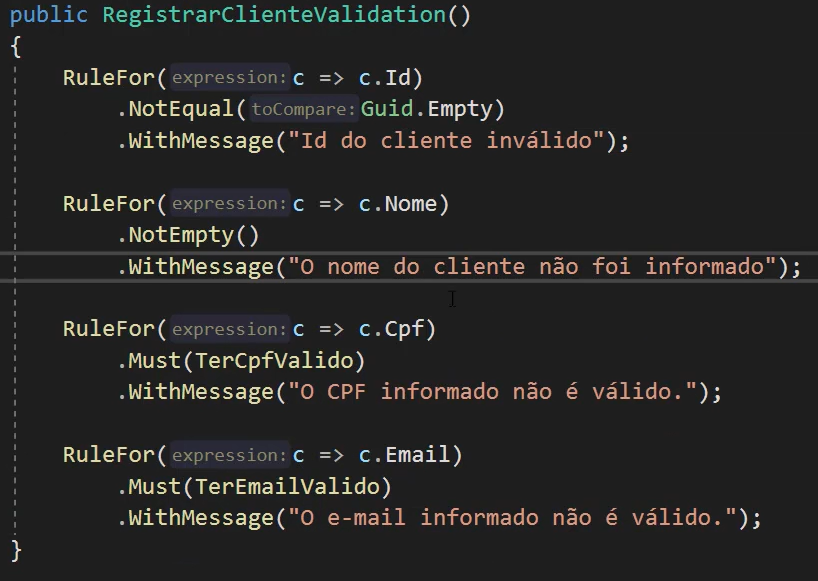
* Validação de Comandos

->Em NSE.Cliente.API/Application/Commands/RegistrarClienteCommand criar a classe RegistrarClienteValidation.

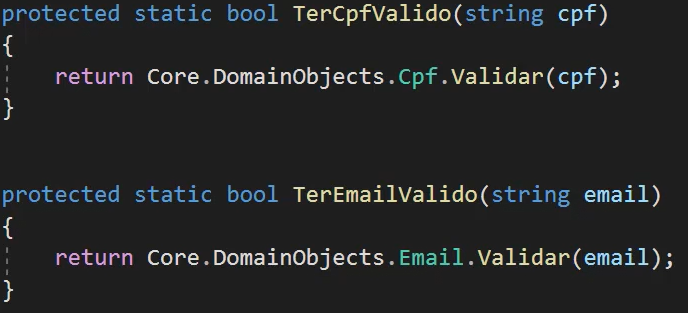
-> Implementar o AbastractValidator, que é uma classe do Fluent Validator.



-> Escrever as regras de validação



-> Criar os métodos que realizam a validação.



-> Realizar um override no método de validação

