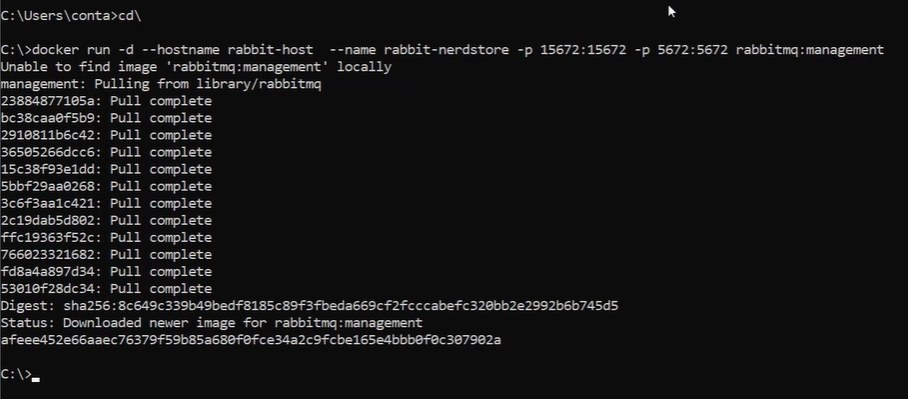
**Comunicação entre API’s com RabbitMQ**

* **Setup do RabbitMQ**

->Installar o RabbitMQ via docker

->docker run -d --hostname rabbit-host --name rabbit-nerdstore -p 15672:15672 -p 5672:5672 rabbitmq:management



-> Acesso RabbitMQ : localhost:15672/#/

->guest guest

->Instalar o monitoramento de containers Portainer .

-> [Install Portainer with Docker on WSL / Docker Desktop - Portainer Documentation](https://docs.portainer.io/v/ce-2.11/start/install/server/docker/wsl)

->Acesso : localhost:9443/#!

-> admin admin

* **O básico que você precisa sobre o RabbitMQ**

->Tutoriais estão disponíveis em rabbitmq.com/getstarted.html

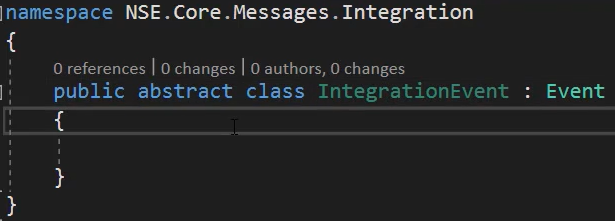
->Usaremos também o EasyNetQ, disponível em easynetq.com

* **Request Response Pattern – Criação do projeto Message Bus**

->Em Building blocks/NSE.Core/Menssages, criar uma pasta chamada Integration

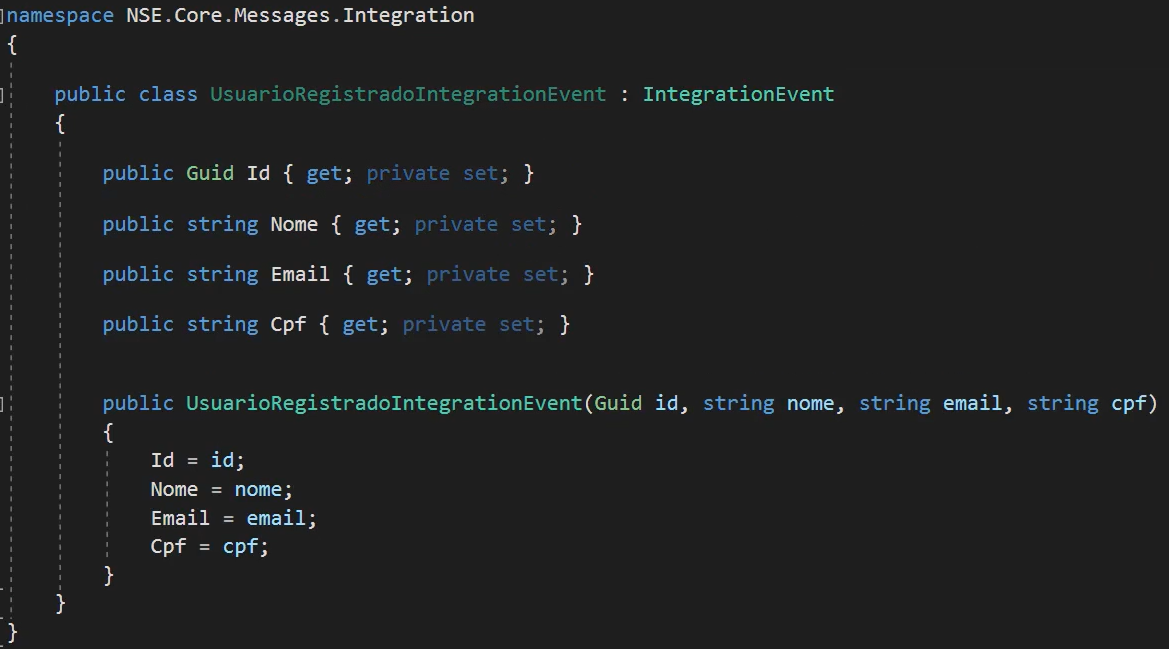
->Dentro dessa pasta, criar uma classe abstrata IntegrationEvent

->Essa classe será uma classe base para os eventos de integração



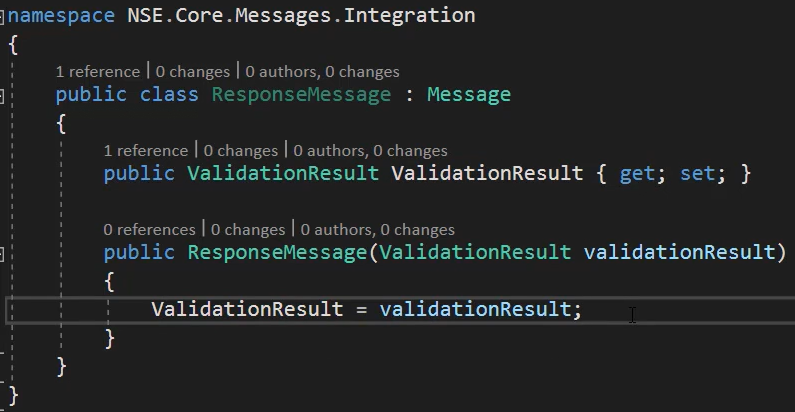
->Na mesma pasta, criar a classe de integração para quando o usuário for registrado, que será a UsuarioRegistradoIntegrationEvent

-> Os dados de clienteUsuario devem ser passados nessa classe

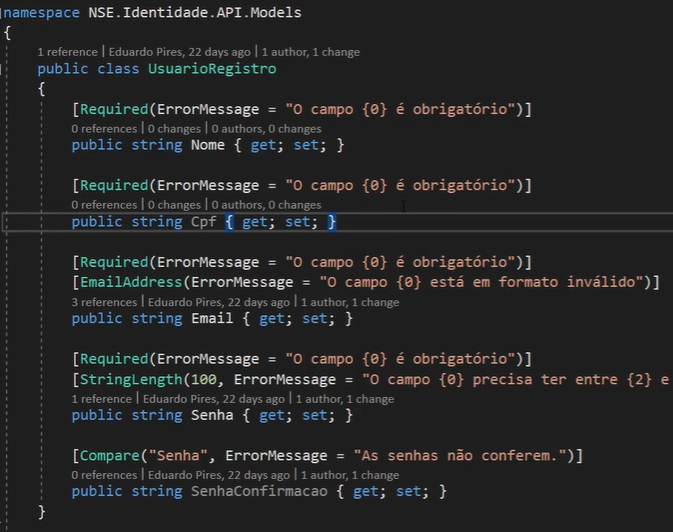


->Na mesma pasta, criar a classe ResponseMessage, que será a classe de resposta

->Nesse caso podemos usar como objeto de resposta o próprio objeto do fluent validation.



->Na API de identidade/Models/UserViewModels , adicionar os campos Nome e Cpf com suas respectivas validações



->Em Building Blocks, criar a pasta MessageBus

->Dentro dessa pasta, criar uma Class Library chamada NSE.MessageBus

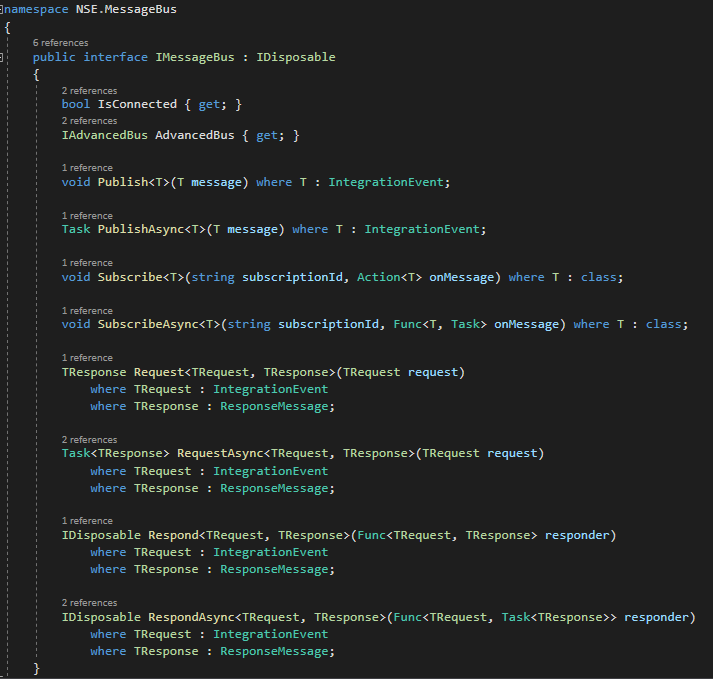
-> Instalar o pacote EasyNetQ no projeto

-> Install-Package EasyNetQ.Management.Client

->Dentro do projeto criar a interface IMessageBus

-> Essa implementação específica permite realizar conexões e reconexões mesmo que o docker venha a cair por algum motivo

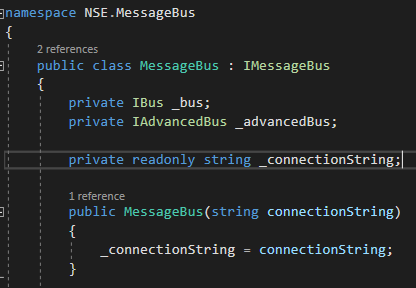
->Os métodos possuem request e responses específicos, ou seja, existe uma personalização dos métodos



->Criar a classe MessageBus, que herda de IMessageBus

->A conexão com o Bus ocorrerá nessa classe

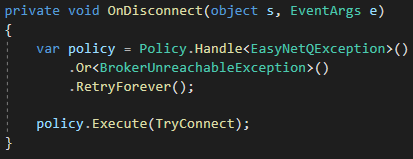
->Antes de importar e personalizar os métodos, deve-se instanciar o Bus e o AdvancedBus



->Agora deve-se criar o método de reconexão que abranja também a reconexão caso o docker venha a cair. Para isso vamos criar o método TryConnect e OnDisconnect

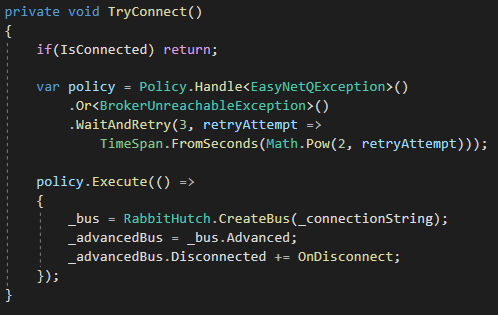
->Criar o Evento OnDisconnect

->Caso o Policy não apareça, instalar o pacote Polly. Install-Package Polly



->Criar o método TryConnect

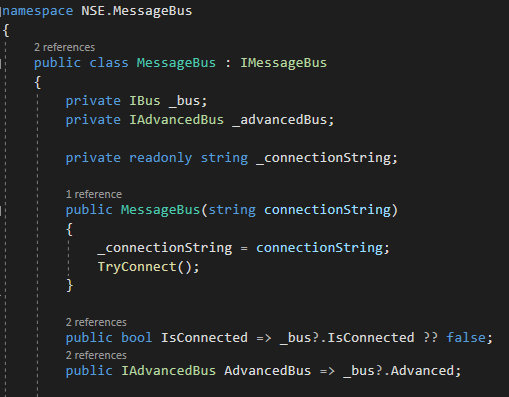
->Aqui se está tratando exceções tanto do EasyNetQ quanto do RabbitMQ



->Acrescentar o TryConnect no construtor.

->Instanciar a flag IsConnected para que ele possar retornar corretamente quando não estiver conectado.

->Incrementar o AdvancedBus como propriedade.



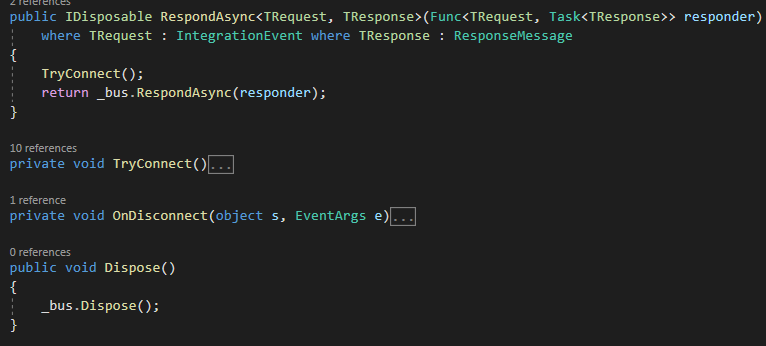
->Implementar Interface

->Acrescentar o TryConnect em cada um

->Chamar o próprio método. Isso faz parte da abstração cujo intuito é possibilitar a inclusão de tratativas personalizadas, que nesse caso é método TryConnect.

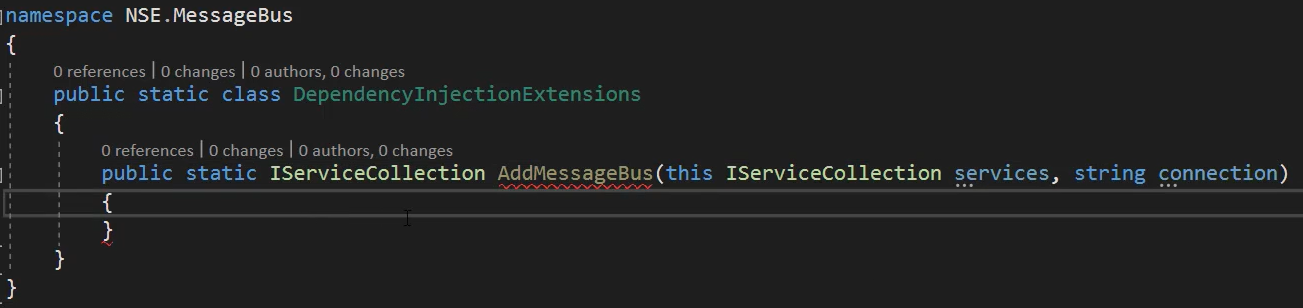




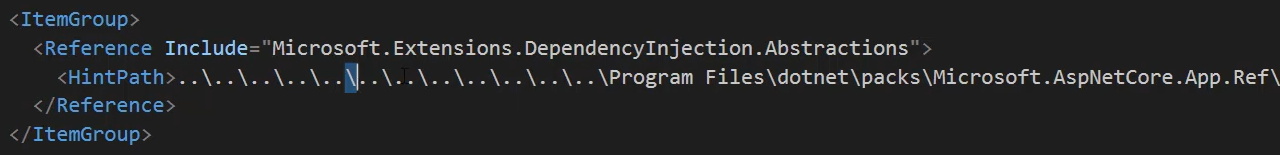


->Para que as APIs implementem o MessageBus da melhor forma possível, vamos criar uma extensão de injeção de dependência.

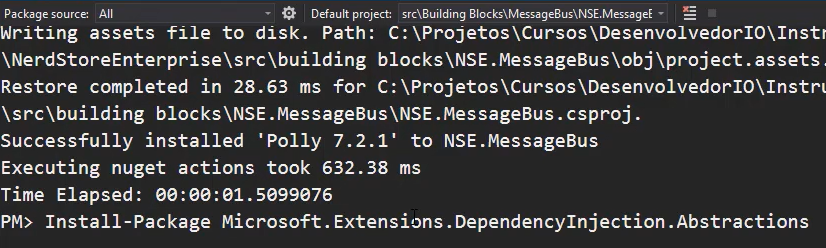
->Em Building Blocks/MessageBus/NSE.MessageBus, criar uma classe estática chamada DependencyInjectionExtensions



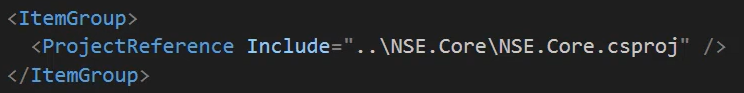
->Tomar cuidado quando for referenciar. A referência não é para ser na dll. Se a referência for deixada desse jeito, ocorrerá o erro quando for feito o deploy. A imagem abaixo é como não deve estar.



->Caso isso ocorra, remover esse itemGroup e realizar a instalação pelo package manager console.

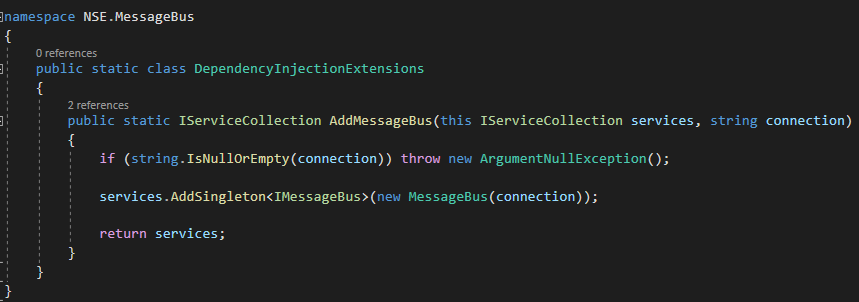


-> Com isso a instalação se dará da forma correta



-> Verificar se a string está vazia e Registrar o MessageBus como singleton

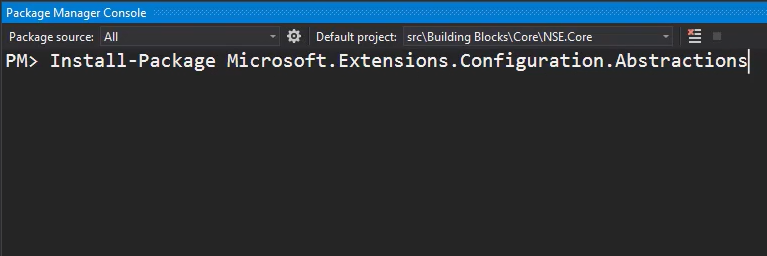
-> A connection string não será definida dentro do MessageBus. Essa definição caberá a cada API separadamente.



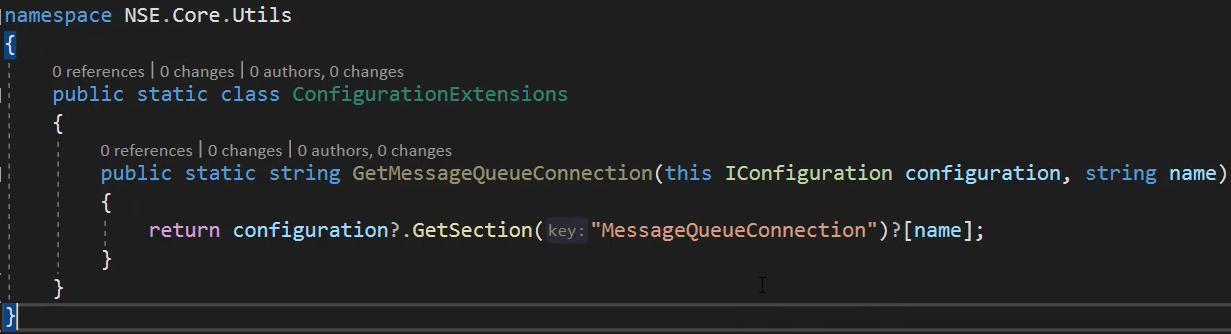
-> Para que se possa buscar a connection string no app settings de cada projeto, vamos criar um método para isso.

->Em building blocks/Core/NSE.Core/Utils, criar uma classe estática ConfigurationExtensions para assim poder criar uma extensão do IConfiguration.

->Lembrar de realizar a referência através da instalação do pacote no Package Manager Console. Isso evitará que a referência seja feita a dll.



-> Incrementar o método para que essa busca possa ocorrer na seção adequada

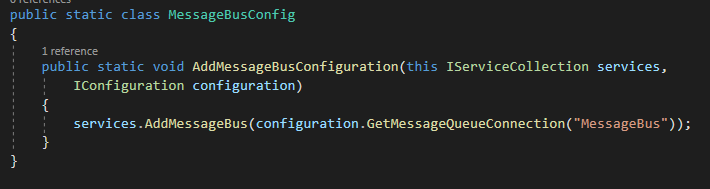


-> Esse projeto já está pronto para ser utilizado por outras APIs . Basta referenciar o projeto e utilizar na construção do método.

* **Preparação das APIs para utilização do Message Bus**

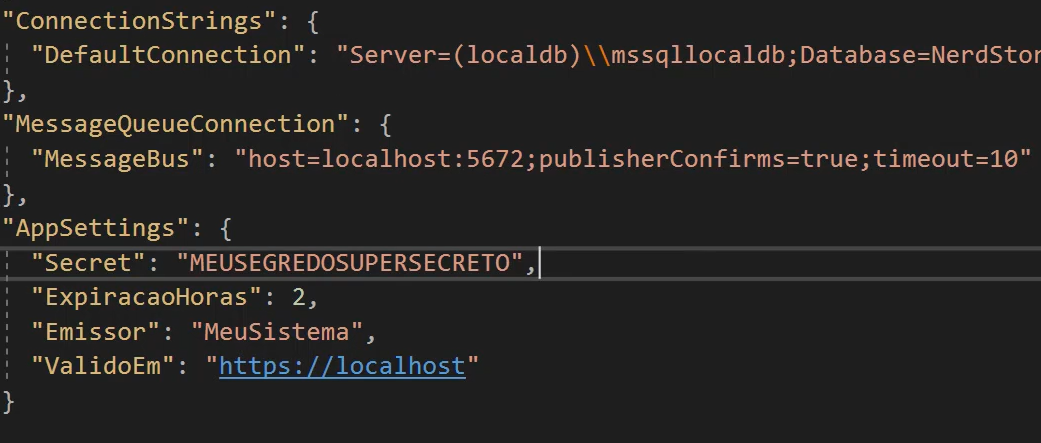
->Na API de Clientes/Configuration, criar uma classe estática chamada MessageBusConfig

->Essa classe será responsável por obter a connection string correta

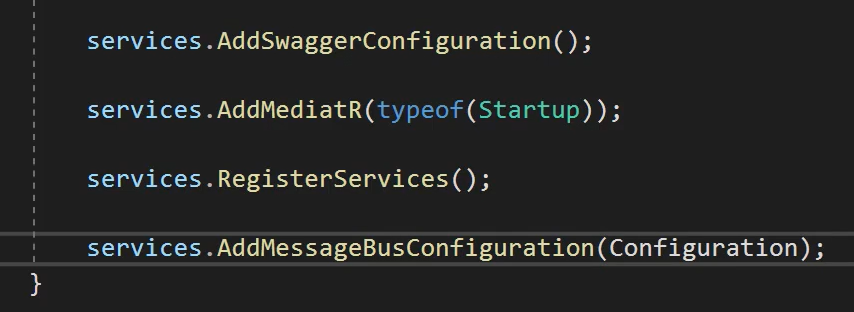


->No appsettings.Development, delimitar qual será a nossa Connection String para a api de Cliente

->A opção publisherConfirms=true diz que tem de receber uma confirmação que a mensagem foi recebida. Caso essa confirmação não seja recebida, a aplicação retornará uma exception.



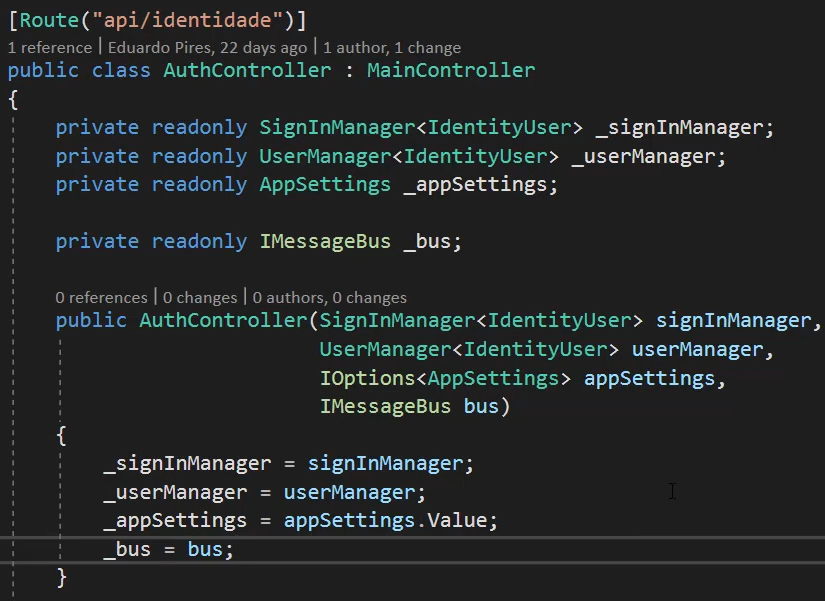
->Chamar o serviço do AddMessageBusConfiguration na startup



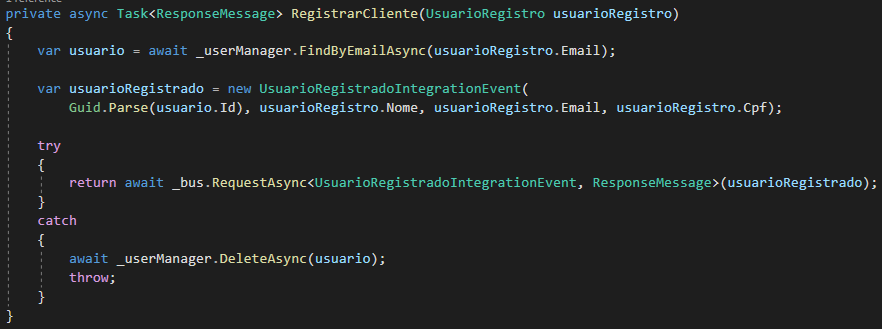
->Realizar a mesma configuração da API de Identidade

* **Utilizando o projeto MessageBus na API de Identidade(Request)**

-> Injetar o MessageBus na AuthController



->Na AuthController, criar a task RegistrarCliente



->Com esse método criado podemos realizar as alterações necessárias no método registrar



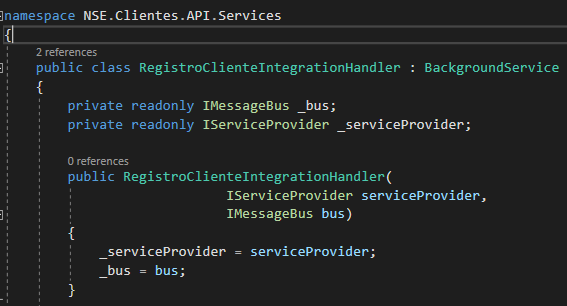
* **Utilizando o projeto MessageBus na API de Cliente(Response)**

->No projeto NSE.Clientes.Api, criar uma pastar chamada Services

->Dentro dela, criar uma classe chamada RegistroClienteIntegrationHandler

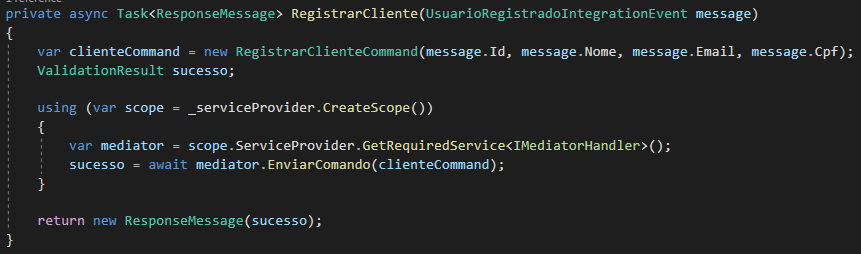
->Essa classe trabalhará como um background service

->Injetar os sevicos e projetos necessários

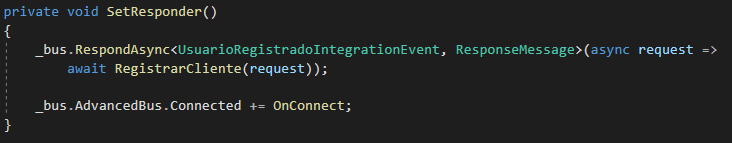


->Criar a task RegistrarCliente

->Para que seja possível utilizar o serviço singleton num objeto scoped, devemos utilizar o serviceProvider na criação do escopo.

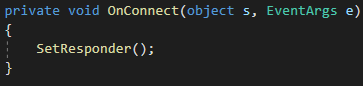


-> Criar o método SetResponder



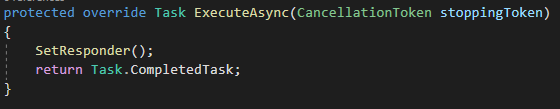
->Criar o evento OnConnect

->Esse método renovará a subscription do RabbitMQ logo assim que ele voltar a ficar on.



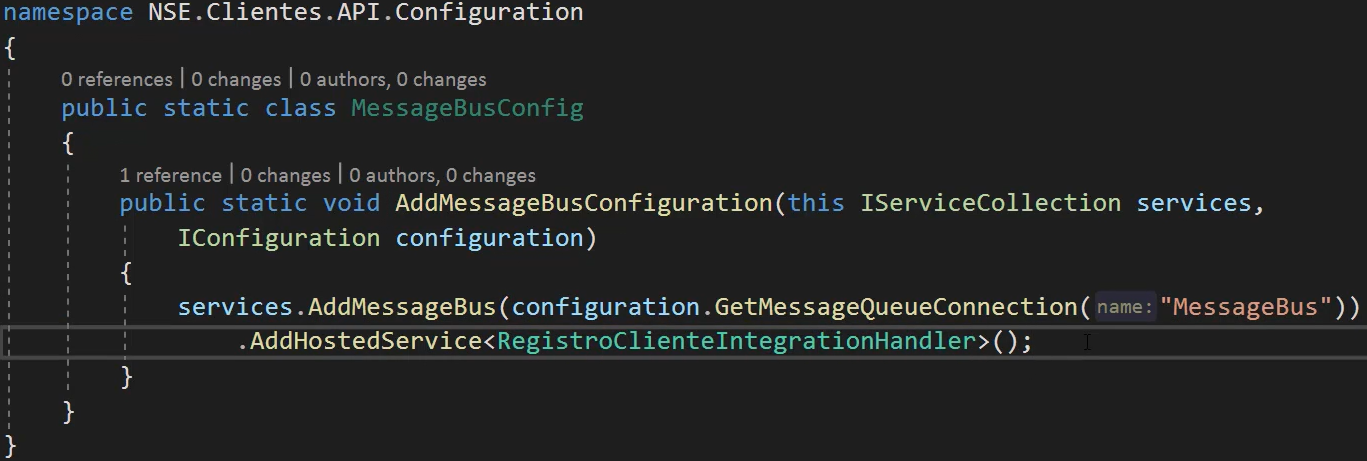
->Criar a Task ExecuteAsync

->Esse método será chamado assim que a aplicação subir



->Na pasta Configuration/MessageBusConfiguration, registrar o HostedService

->Esse tipo de serviço é utilizado quando se tem de trabalhar com singleton sendo que o objeto é Scoped, que nesse caso é o Mediator



->Testar a aplicação