Desafio BC

Documento Visão Geral  
*Esse documento tem por finalidade explanar como foi desenvolvido a solução.*

É abordado sobre o problema, uma possível solução estrutural, como as duas Apis estão estruturadas, e de que forma foi implantado o banco de dados.

# Problema Proposto

*Segundo a empresa, o problema proposto é: “A empresa possui o cadastro via APP, e nessa onda, habilitamos o cadastro via API para Onboard de nossos clientes.Tendo esse problema, você deverá criar uma API para o cadastro destes cliente”*.

# Solução Arquitetura

Eu optei por criar duas Apis, que são: Api.RegisterCustomer e Api.PostalCode conforme pode ser visto na **Figura 1**.

A Api.PostalCode tem responsabilidade em enviar requisição a um link externo para obter o endereço completo. Já Api.RegisterCustomer tem um escopo um pouco maior, é possível cadastrar e buscar por clientes.

Diagrama

Descrição gerada automaticamente

Figura 1: Solução arquitetural para o problema proposto.

Na minha opnião, por desconhecer o sistema por completo e em virtude de se ter uma maior escalabilidade, portanto é interessante a criação de duas Apis que funcione de forma isolada, com escopo de funções menores e em virtude de indisponibilidade do serviço, as demais partes do sistema continuarão a funcionar. Ressaltando que o desenvolvimento é rápido e o custo é menor ao comparar-se com um sistema maior e uma api de responsabilidade única.

# Estrutura da Aplicação

Eu acredito que a estrutura da aplicação deve ser simples e muito bem feita, em razão da Api ter poucas responsabilidades. Desta forma, as Apis estão estruturadas seguindo o conceito de arquitetura limpa, dando prioridade a teste de unidade em conjunto com os testes de mutação, o projeto está de fácil entendimento, desta forma ganhando tempo de desenvolvimento e manutenções futuras, trazendo lucros maiores para a empresa no presente e menores prejuízo no futuro.

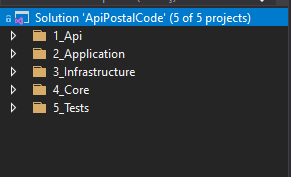
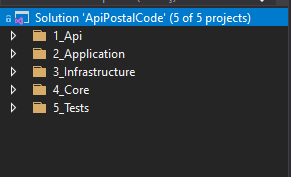
 

Figura 2: Estrutura das Apis com o conceito de arquitetura limpa.

Foi utilizado o Visual Studio Enterprise 2019, os projetos foram criandos com WebApi .net 5, Entity Framework Core para abstração do banco de dados, FluentValidation para validar a entrada de dados, Fluent Assertions nos testes de unidade com Xunit e o por fim, versionamento de código feito no [GitHub](https://github.com/Jeffconexion/Challenge_BC).

**Na Figura 3**, foi realizado teste de unidade em conjuto com o teste de mutação, dando prioridade a unidades de maior importância. É possível ver na imagem, mais de 80 mutantes foram mortos.

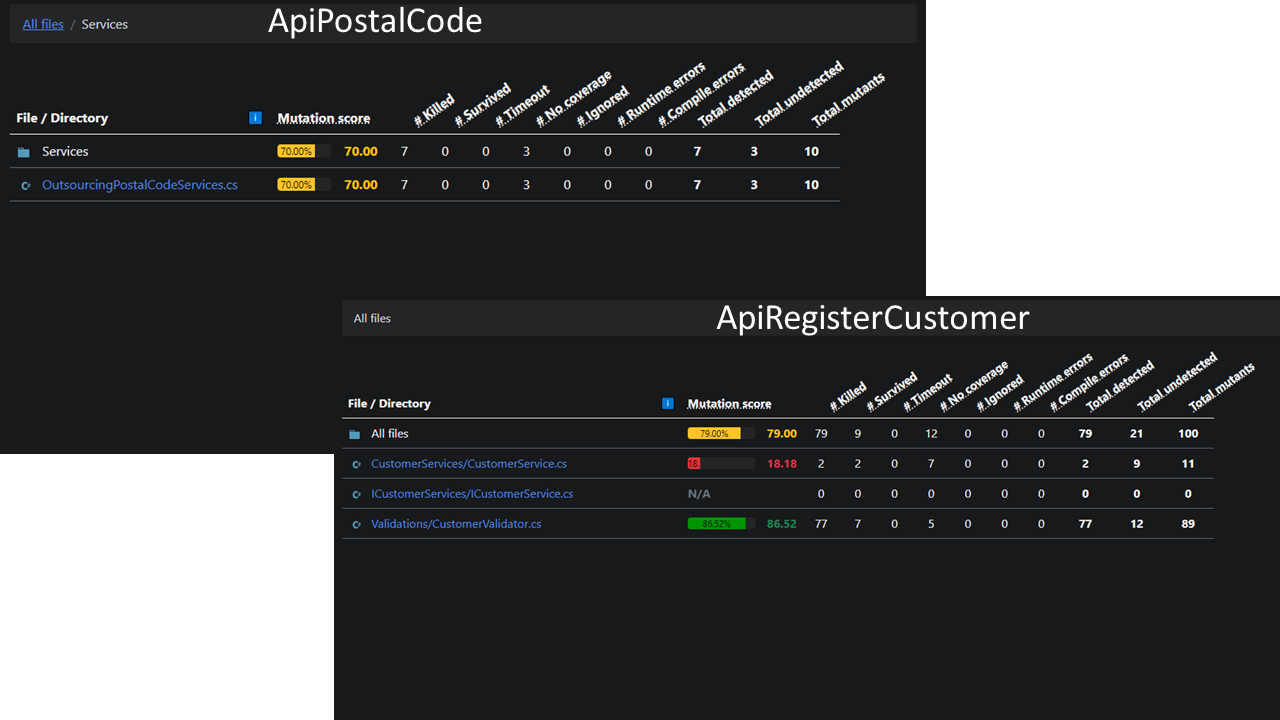


Figura 3: Validaando os teste de unidade com os teste de mutação.

Na pasta 3\_Scripts/2\_PostmanCollection vai estar a documentação utilizada no Postman.

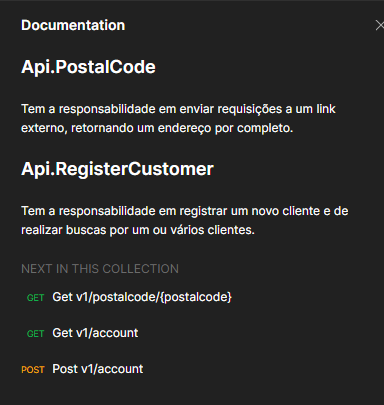


Figura 4: Documentação gerada pelo Postman.

Além do mais, foi realizado teste de integração para detecção de erros e integralidade entre componentes. Para execução dos testes, foi criado um arquivo appsettings com foco em teste, conforme a imagem a seguir.

Texto

Descrição gerada automaticamente

Figura 5: Configurações para Multiplos appsettings.

Para realização do teste de integração, é necessário baixar uma imagem do Sql Server e levantar o container.

1. Baixar e criar o container.

docker run -e "ACCEPT\_EULA=Y" -e "SA\_PASSWORD=04319283Numsei#2022" -p 1433:1433 --name sqlserverdb -d mcr.microsoft.com/mssql/server:2019-latest

1. Configurar o appsettings.Testing.json

"SqlConnection": "Server=localhost,1433;Database=ProductsDb;User Id=sa;Password=04319283Numsei#2022"

1. Criar o banco e tabelas para o Sql Server, disponível na pasta **3\_Scripts/1\_SqlServerScript**

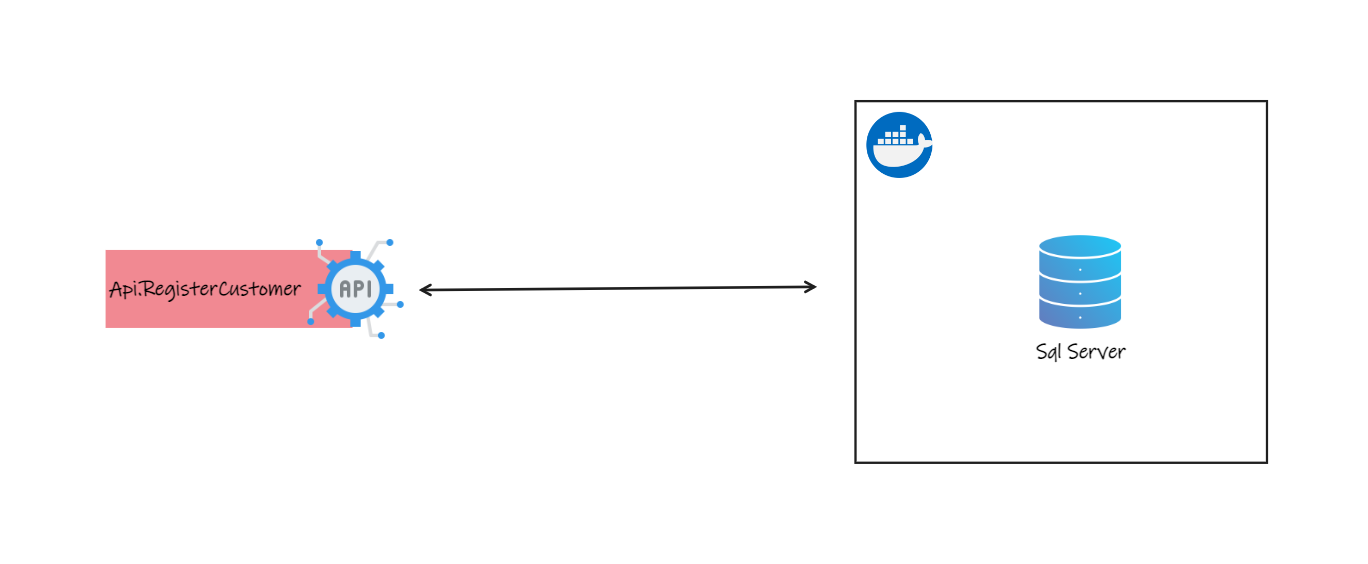


Figura 6: Persistindo dados no container por uma aplicação em execução local.

# Branchs

Foi criado duas branchs, a **master** uso container para teste de integração, porém na **branch\_with\_twoContainer** eu criei um dockerfile e orquestração por meio do docker-compose.

1. Criação do container

docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d

1. Acessando o container

<http://localhost:8000/swagger/Index.html>

1. Problemas

Eu tive vários problemas em relação ao Swagger, ao gerar o xml, ao gerar o container não era possível localizar o xml.

# Banco de Dados

Para realização da persistência dos dados, foi utilizado o Sql Server Express, sem a geração do migration pelo Visual Studio, dessa forma seguindo o conceito de database first. Entretanto, tomando os devidos cuidados para trazer as regras de negócio para responsabilida da aplicação. A seguir um dicionário de dados e os scripts para executar no Sql Server.

Na **Tabela 1 e Tabela 2**, mostro as tabelas que foram criadas como regra de negócio para aplicação. Inclusive foi criado uma view, com os campos essenciais das três tabelas, ganhando dessa forma um maior desempenho ao realizar uma consulta.

Tabela : Descrição das tabelas e view.

|  |  |
| --- | --- |
| **Tabela** | **Descrição** |
| TB\_CUSTUMER | Tabela para registrar o novo cliente cadastrado no sistema. |
| TB\_ADDRESS | Tabela para registrar o endereço mediante o cep informado. |
| TB\_STATUS\_ADDRESS | Tabela para registrar o status da TB\_ADDRESS. |
| VW\_FULLDATA\_CUSTUMER | View com os campos essenciais das 3 tabelas. |

Tabela 2: Campos e descrições da view criada.

|  |  |  |
| --- | --- | --- |
| **Tabela** | **Coluna** | **Descrição** |
| VW\_FULLDATA\_CUSTUMER | name | Nome do cliente |
|  | tax\_id | Cpf ou Cnpj validado na aplicação. |
|  | phone\_number | Telefone do cliente. |
|  | created\_at | Data de criação da conta. |
|  | status | Para o Cep informado vai ser atribuído “Approved” caso contrário “Pending” |
|  | state | Estado do cliente |
|  | city | Cidade do cliente |
|  | district | Bairro do cliente |
|  | address | Endereço do cliente |
|  | code | Cep do cliente |

Na **Tabela 3**, Mostro todas as colunas, tipo, tamanho e o significado de cada coluna para cada tabela.

Tabela : Tabelas, colunas, tipo, tamanho e descrições de cada campo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tabela** | **Coluna** | **Tipo** | **Tamanho Máximo** | **Descrição** |
| TB\_CUSTUMER | id | uniqueidentifier | 16 | Guid gerado na aplicação. |
| TB\_CUSTUMER | name | varchar | 60 | Nome do cliente |
| TB\_CUSTUMER | tax\_id | varchar | 25 | Cpf ou Cnpj validado na aplicação. |
| TB\_CUSTUMER | password | varchar | 30 | Senha mínimo caracteres de 8, 1 letra Maiúscula, 1 Minúscula, Caractere Especial. Validado na aplicação |
| TB\_CUSTUMER | phone\_number | varchar | 30 | Telefone do cliente. |
| TB\_CUSTUMER | created\_at | datetime | 8 | Data de criação da conta. |
| TB\_STATUS\_ADDRESS | id | uniqueidentifier | 16 | Guid gerado na aplicação. |
| TB\_STATUS\_ADDRESS | status | varchar | 10 | Para o Cep informado vai ser atribuído “Approved” caso contrário “Pending” |
| TB\_ADDRESS | id | uniqueidentifier | 16 | Guid gerado na aplicação. |
| TB\_ADDRESS | id\_custumer | uniqueidentifier | 16 | Chave estrangeira que referência a TB\_CUSTUMER. |
| TB\_ADDRESS | id\_status\_address | uniqueidentifier | 16 | Chave estrangeira que referência a TB\_STATUS\_ADDRESS |
| TB\_ADDRESS | state | varchar | 4 | Estado do cliente |
| TB\_ADDRESS | city | varchar | 200 | Cidade do cliente |
| TB\_ADDRESS | district | varchar | 200 | Bairro do cliente |
| TB\_ADDRESS | address | varchar | 200 | Endereço do cliente |
| TB\_ADDRESS | code | varchar | 200 | Cep do cliente |

Em **Script 1**, está o passo a passo para criação do banco de dados, tabelas e view. Ressaltando que esse Script está na pasta **3\_Scripts**.





Script : Script utilizado para persistir dados da Api.RegisterCustomer.