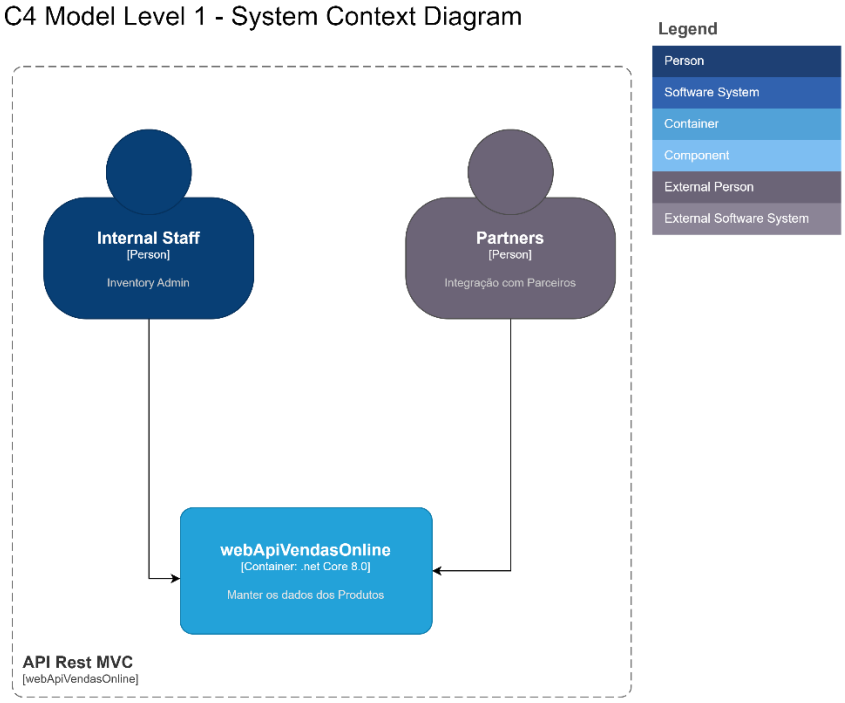
Relatório do Desafio

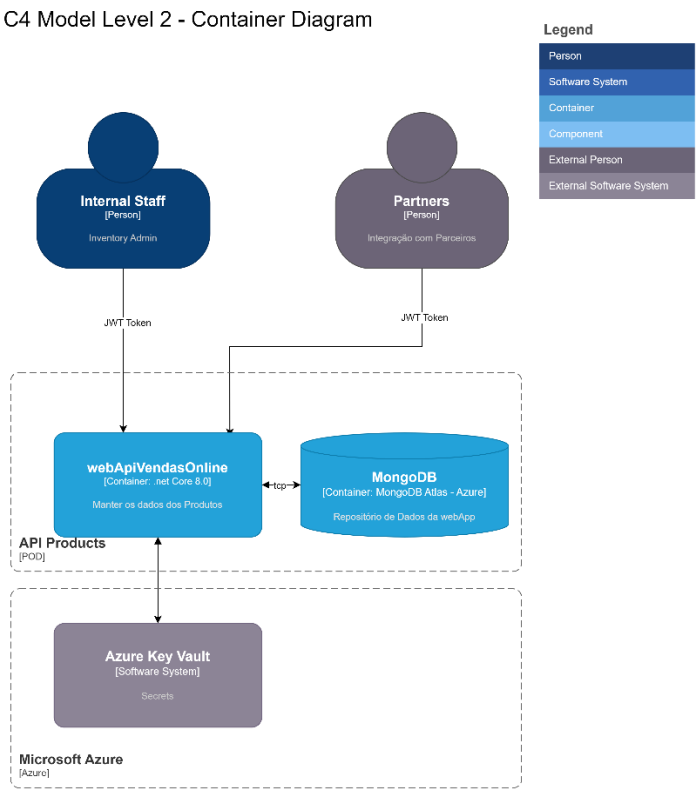
Nome: Renan Evangelista Pereira

1. Desenho arquitetural do software:

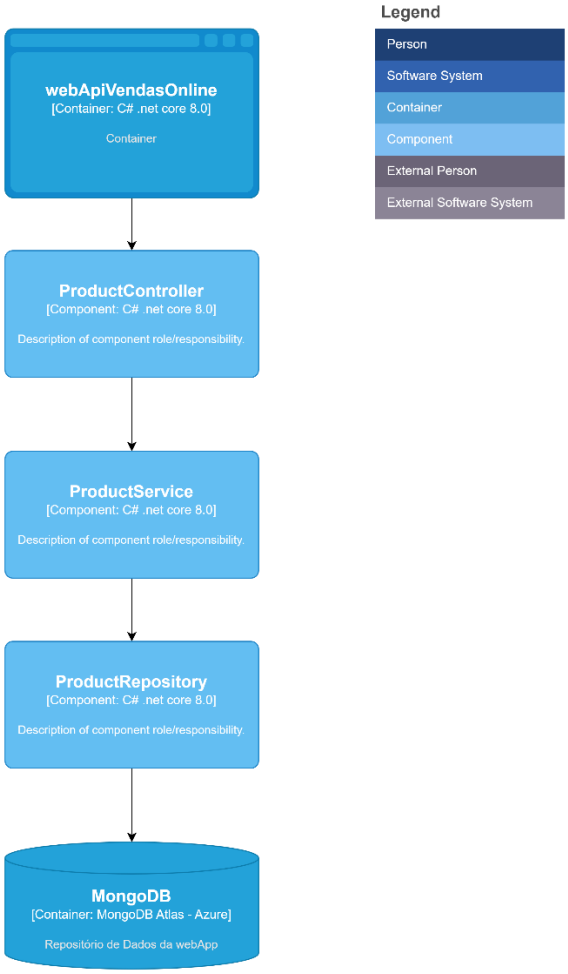
a. Modelo C4 – Level 1:



b. Modelo C4 – Level 2:

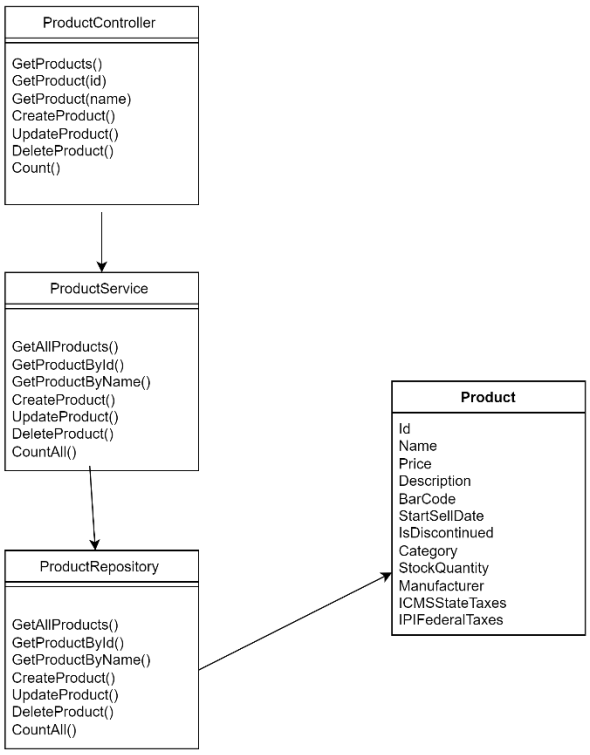


c. Modelo C4 – Level 3:



d. Modelo C4 – Level 4:

C4 Model Level 4 - Code Diagram



2. Estrutura de pastas e explicação dos componentes:

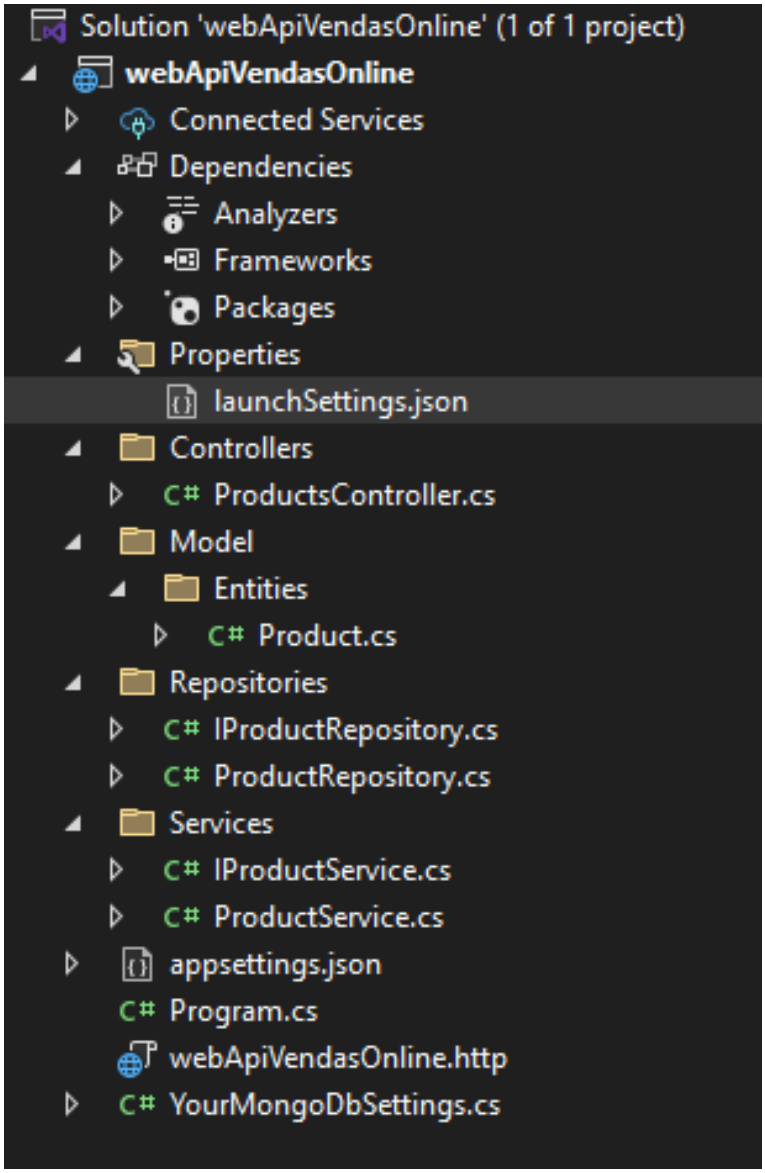
A estrutura de pastas de do projeto webApiVendasOnline foi organizada de forma a facilitar a manutenção do código, obedecendo uma estrutura lógica e obedecendo a padrões utilizados em projetos MVC. Segue o detalhamento da estrutura:

1. **Controllers:** A pasta Controllers em um projeto C# MVC .NET Core é essencial, pois contém classes que processam solicitações dos usuários e retornam respostas. Cada controlador herda da classe base Controller e possui métodos de ação que manipulam requisições HTTP para renderizar as respostas**.** Nesse projeto temos o seguinte arquivo:
    - ProductsController.cs
2. **Models:** A pasta Models em um projeto C# MVC .NET Core desempenha um papel essencial, abrigando as classes que representam os dados e a lógica de negócios do aplicativo. Essas classes são incumbidas de armazenar, manipular e validar os dados do sistema. Adicionalmente, os modelos frequentemente estabelecem interações com o banco de dados, especialmente quando se utilizam frameworks ORM, como o Entity Framework Core. Nesse projeto temos o seguinte arquivo:
    - Entities/Product.cs: Define a classe Product, que representa um produto no sistema de vendas online, com propriedades como Id, Name, Price, Description, etc.

3. **Views**: A pasta Views em um projeto C# MVC .NET Core é incumbida de armazenar as visualizações do aplicativo. Essas visualizações determinam a aparência e a estrutura das páginas que serão apresentadas aos usuários. Elas podem ser desenvolvidas utilizando a sintaxe Razor (ou Blazor), que possibilita a combinação de C# com HTML para a geração de conteúdo dinâmico. Nesse projeto não temos essa pasta pois criamos uma API RESTful que retorna apenas Application/Json nos seus endpoints.
    - Inexistente
4. **Repositories:** A pasta Repositories em um projeto C# MVC .NET Core é empregada para implementar o padrão de repositório, que estabelece uma camada de abstração entre a camada de acesso a dados e a lógica de negócios do aplicativo. Essa metodologia facilita a manutenção e a testabilidade do código, ao isolar a lógica de acesso a dados em classes específicas, denominadas repositórios. Permite, ainda, que o aplicativo utilize diferentes tecnologias de acesso a dados sem a necessidade de modificar a lógica de negócios. Nesse projeto temos o seguinte arquivo:
    - ProductRepository.cs: Contém a implementação do repositório de produtos, responsável por interagir com o banco de dados MongoDB para operações CRUD (Create, Read, Update, Delete) de produtos.
5. **Services:** A pasta Services em um projeto C# MVC .NET Core é essencial para a organização da lógica de negócios da aplicação. Esta pasta é destinada a classes que executam operações que não se enquadram diretamente nos controladores, modelos ou visualizações. As principais funções dessas classes incluem: I) Interações com APIs externas, realização de chamadas a serviços web ou APIs de terceiros; II) Processamento de dados, implementação de lógica complexa de manipulação de dados, que não deve ser realizada nos controladores; III) Serviços de domínio, aplicação de regras de negócios específicas do domínio da aplicação; IV) Gerenciamento de dependências, facilitação da injeção de dependências, promovendo um código mais modular e testável; Essa estruturação contribui para a manutenção da organização do código, além de facilitar a manutenção e os testes da aplicação**.**
    - **ProductService.cs:** O arquivo ProductService.cs contém a implementação da lógica de negócios relacionada aos produtos no sistema de vendas online. A função principal deste arquivo é servir como uma camada intermediária entre o repositório de dados (ProductRepository) e os controladores da API, encapsulando a lógica de negócios e operações específicas que não devem estar diretamente no repositório ou nos controladores.
6. **Interfaces:** A interface serve como um contrato que garante que qualquer classe que a implemente fornecerá as funcionalidades descritas. Nesse projeto não foi criada uma pasta específica para as interfaces, porém é comum em projetos maiores colocá-las em pastas ou “assemblies” separados para utilizar padrões mais complexos que facilitariam a manutenção ou a solução de problemas. Poderia permitir que a aplicação dependa de abstrações em vez de implementações concretas, facilitando a manutenção e a testabilidade do código. Neste projeto temos como exemplo os arquivos abaixo:

- IProductRepository.cs: define a interface para o repositório de produtos, especificando os métodos que devem ser implementados, como GetAllProductsAsync, GetProductByIdAsync, CreateProductAsync, etc.
- IProductService.cs: define a interface que especifica os métodos que a classe de serviço de produtos (ProductService) deve implementar.

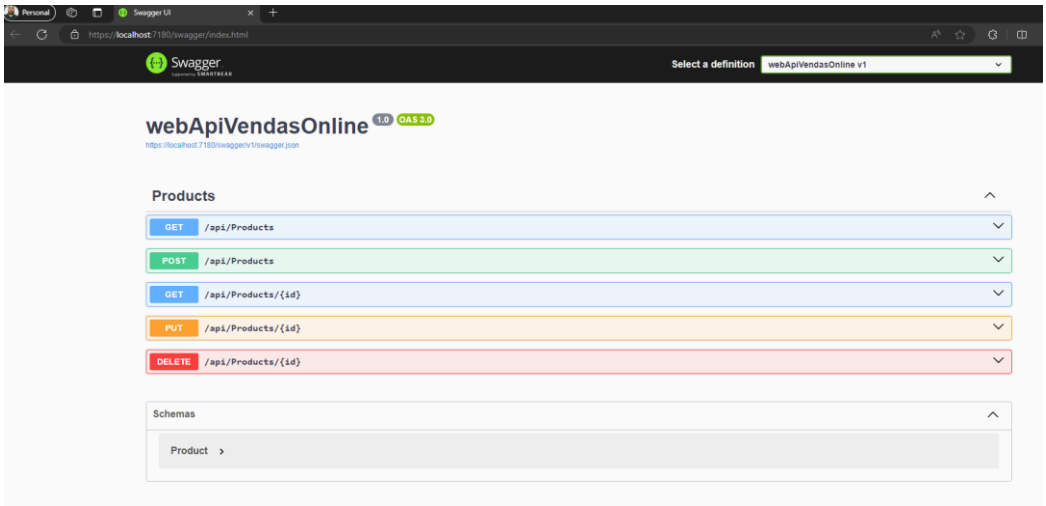
7. **Settings:**
   - YourMongoDbSettings.cs: Define a classe YourMongoDbSettings, que contém as configurações necessárias para conectar ao banco de dados MongoDB, como ConnectionString e DatabaseName.
   - appsettings.json: esse é um arquivo de configuração usado em aplicações .NET para armazenar configurações de forma estruturada e em formato JSON. Ele é comumente utilizado para definir configurações de conexão com banco de dados, chaves de API, configurações de logging, e outras configurações específicas da aplicação. Nesse projeto foram acrescentadas a string de conexão para o MongoDB, rodando na plataforma Atlas, no Azure.
8. **Program.cs:**
   - Arquivo principal que configura e inicia a aplicação. Contém a configuração dos serviços e a construção do aplicativo.
9. Captura da estrutura de pastas do projeto webApiVendasOnline, na solução webApiVendasOnline:

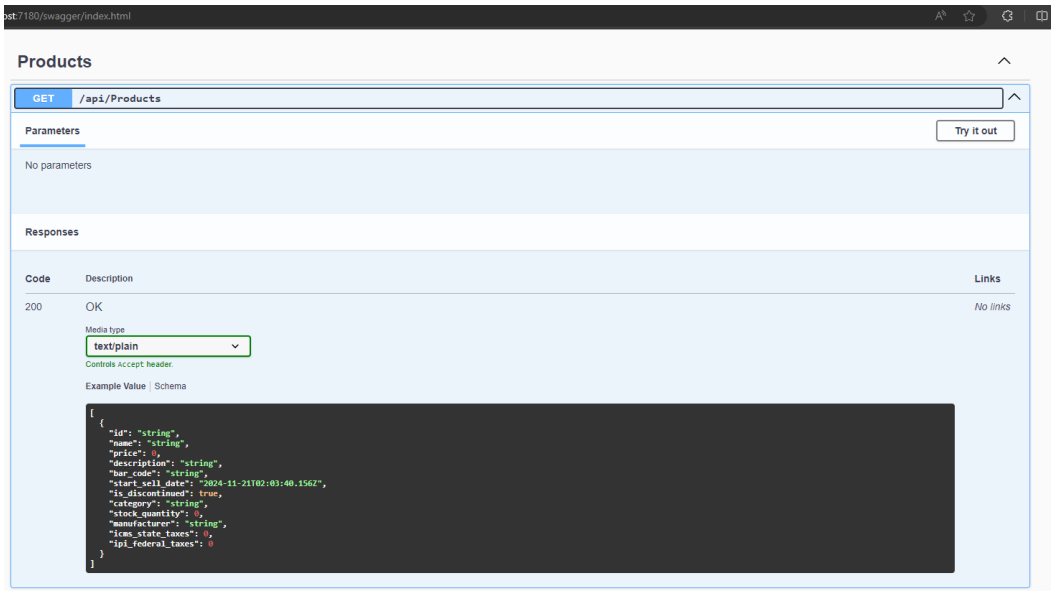


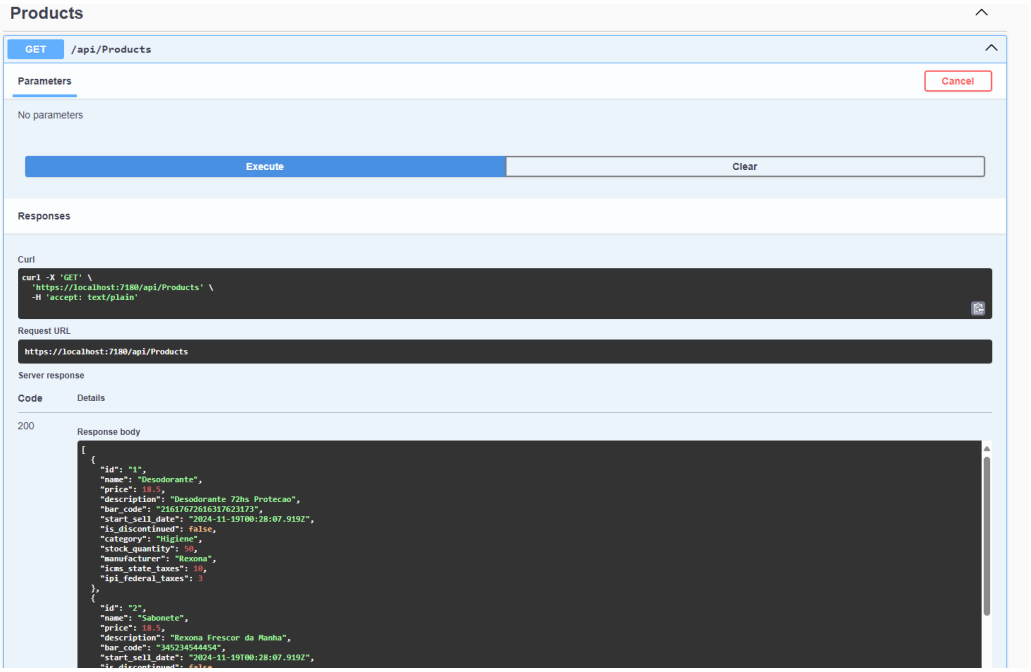
10. Repositório do Código:

    https://github.com/RenanEvangelistaPereira/desafioArquiteturaSoftwareXP

11. Swagger:







3. Avaliação da Implementação do Projeto:

Para avaliar o projeto em relação aos princípios SOLID, vamos revisar o código fornecido e identificar possíveis violações dos princípios SOLID:

1. **Princípio da Responsabilidade Única (SRP)**:
    - Cada classe deve ter apenas um motivo para mudar.
    - **Avaliação**:
        - A classe Product adere ao SRP, pois representa apenas a entidade do produto.

        - A classe ProductService é responsável pela lógica de negócios e validação, o que é apropriado.
        - A classe ProductsController lida com solicitações e respostas HTTP, o que é apropriado.

2. **Princípio Aberto/Fechado (OCP)**:
    - As classes devem estar abertas para extensão, mas fechadas para modificação.
    - **Avaliação**:
        - A classe ProductService pode ser estendida adicionando novos métodos sem modificar os existentes.
        - A classe ProductRepository pode ser estendida adicionando novos métodos sem modificar os existentes.
        - A classe ProductsController pode ser estendida adicionando novos endpoints sem modificar os existentes.
3. **Princípio da Substituição de Liskov (LSP)**:
    - Subtipos devem ser substituíveis por seus tipos base.
    - **Avaliação**:
        - A classe ProductService implementa a interface IProductService, e qualquer classe que implemente essa interface deve ser substituível.
        - A classe ProductRepository implementa a interface IProductRepository, e qualquer classe que implemente essa interface deve ser substituível.
4. **Princípio da Segregação de Interfaces (ISP)**:
    - Os clientes não devem ser forçados a depender de interfaces que não utilizam.
    - **Avaliação**:
        - As interfaces IProductService e IProductRepository são bem definidas e específicas para suas respectivas responsabilidades, aderindo ao ISP.
5. **Princípio da Inversão de Dependência (DIP)**:
    - Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
    - **Avaliação**:
        - A classe ProductService depende da interface IProductRepository, aderindo ao DIP.
        - A classe ProductsController depende da interface IProductService, aderindo ao DIP.

Minha conclusão é que com base no código disponibilizado, o projeto adere bem aos princípios SOLID. Não há violações significativas dos princípios SOLID. Cada classe tem uma única responsabilidade, o

código está aberto para extensão, as interfaces são bem definidas e as dependências são invertidas de forma apropriada.