Decisões de Projeto

Database

- Foi utilizado o `SQLite` pela praticidade de utilização junto ao `Dapper` e a possibilidade de migração para bancos mais robustos como `SQL Server` no futuro.
- Proposta inicial de uma única tabela dada a pequena quantidade de dados necessária, mas sempre é possível estender o banco usando o campo `id` como chave estrangeira entre as tabelas.
- Campo `ConnectionString` presente no arquivo `appsettings.json` para utilização de qualquer database futura. Para ambiente de `debug` foi fixado um banco `localhost` utilizando `preprocessor directives`

Estrutura da `.sln` (BackEnd)

- O projeto `.NET Core` foi criado usando meu template de API
- A *solution* é separada em 4 *projects*:
  - `Dti.Api.Test`: projeto ASP.NET Core (3.1) contendo os arquivos de inicialização, Controllers, Middlewares e arquivos de orquestração
  - `Dti.Api.Test.Facades`: projeto em .NET Standard (2.1) com toda a camada de regra de negócio/lógica e até injeção de dependências. O projeto ASP.NET "enxerga" apenas essa camada.
    - Originalmente existia um projeto `Dti.Api.Test.Services` que serviria para comunicação com serviços terceiros, que seria enxergue somente por este projeto, mas na ausência da necessidade foi removido.
  - `Dti.Api.Test.Models`: projeto em .NET Standard (2.1) contendo os modelos de objetos a serem utilizados através da solução. Como são essencialmente `DTOs`, não há problemas (em primeira vista) de ser "enxergue" por todos os projetos da solução.
  - `Dti.Api.Test.Tests`: projeto em .NET Core (3.1) contendo testes automatizados para a solução. Nesse primeiro momento foram feitos **testes de integração** entre a API e o banco de dados, com cobertura razoável.
  - Por alto vemos uma certa arquitetura por camadas, onde separamos regra de negócio de camada de interface (e de serviços terceiros, se existirem)
- O projeto já está preparado para *deploy* em `Kubernetes`, necessitando apenas de ajustes finos nos `yaml` de acordo com o ambiente (configurável em CI)
- Toda a API foi documentada usando *Swagger* e a interface está disponível na raiz da API, quando executada.
- Um princípio de *health check* foi implementado junto com uma interface de health check (disponível em `/healthchecks-ui`) utilizando AspNetCore.Diagnostics.HealthChecks.
- A solução também está preparada para utilização de **Sonarqube**, com os pacotes necessários, bem como GUIDs em cada projeto.

FrontEnd

- Foi utilizado o *boilerplate* padrão disponibilizado pelo Facebook, usando `React 16.13`
- Interface simplificada com 4 telas:

- Tela inicial mostrando a listagem de produtos, com botões de edição, exclusão, criação e busca
- Tela de edição, possibilitando editar um ou mais campos do produto selecionado
- Tela de criação, possibilitando incluir produtos
- Tela de busca, possibilitando a busca por produtos *pelo ID*
    - Possível expansão para busca por nome