

## Descrição do projeto

Projeto com a proposta de apresentar uma abordagem mais ampla de conhecimentos para montagem de uma solução completa em aspnet core.

A solução desenvolvida segue o conceito conhecido como "Clean Architecture", utilizado em sistemas com arquiteturas modernas, permitindo maior facilidade para manutenção e expansão, e melhor entendimento do código, com divisão clara de responsabilidades para cada uma das camadas que compõem a solução.

A solução está dividida nas seguintes camadas/projetos:

- **Projeto Domain e Projeto Application** - Contém as regras de negócio
- **Projeto Infrastructure** - Contém a persistência
- **Projeto CrossCutting** - Contém classes referenciadas pelos demais projetos
- **Projeto User Interface** - Contém a apresentação/Api
- **Projeto Consumer** – Contém a classe consumer da mensageria
- **Projeto Unit Test** – Contém classes de teste unitário

O projeto implementa os seguintes patterns:

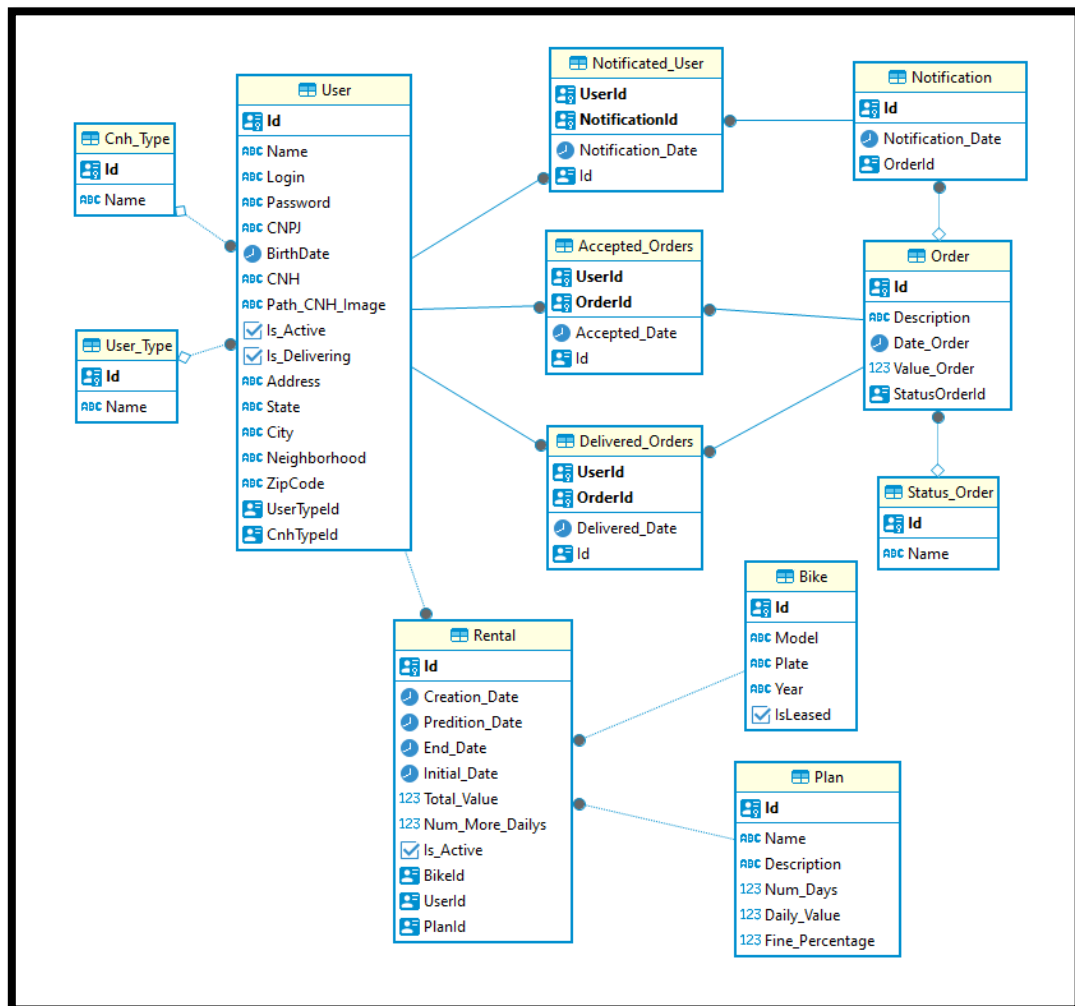
- Clean Architecture
- n-Tier
- Unit Of Work
- Repository
- Class Mapping
- ORM

O projeto utiliza mensageria por meio do **RabbitMQ** em conjunto com a biblioteca de abstração **MassTransit**.

Para o banco de dados, foi utilizado o **Postgres versão 16.2**, implementando a tecnologia ORM por meio do **Entity Framework Core**, para abstração do banco de dados.

Além disso, por meio do EF Core, foi adotado a **abordagem Code First** para modelagem de banco de dados.

A seguir, a modelagem proposta do banco de dados, de acordo com possíveis funcionalidades imaginadas para esse sistema fictício de gestão de locações de moto.



Para dar suporte a execução da solução, é preciso ter instalado o **Docker** localmente. Pelo Docker, serão criados **3 containers** que darão suporte ao sistema. **Um container do banco de dados, outro do Broker de mensageria RabbitMQ e outro com gerenciador de cache distribuído.**

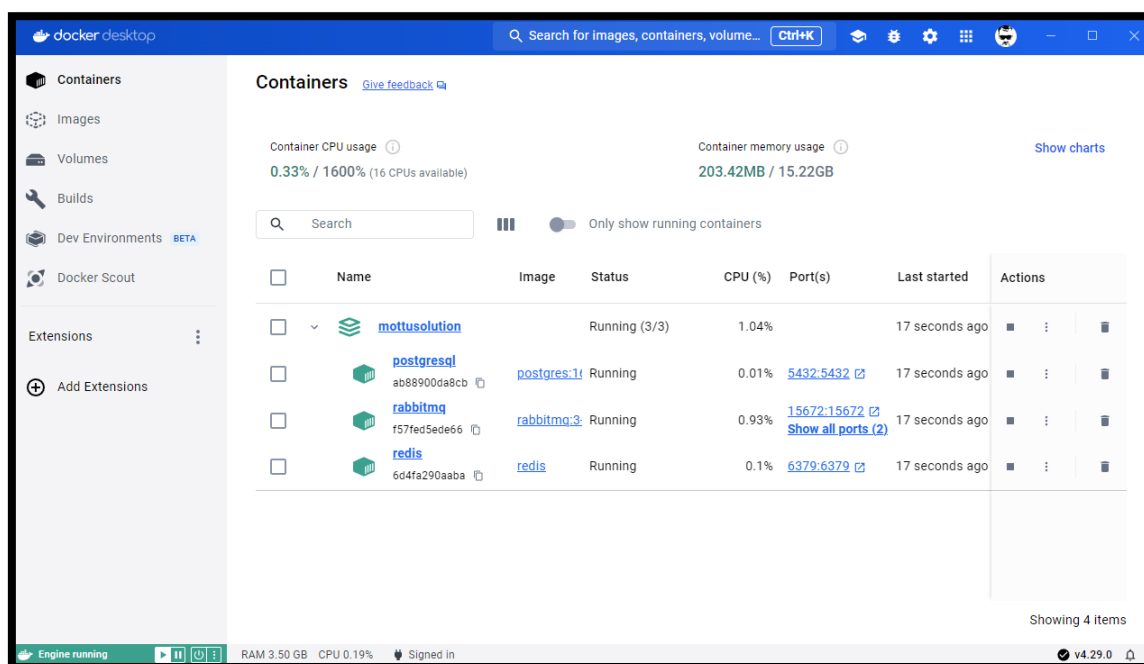
Com a presente solução, se propõe mostrar uma meríade de tecnologias sendo utilizadas num projeto aspNet core, de forma a contemplar boas práticas e a adoção de convenções e padrões de mercado, aplicados ao desenvolvimento de sistemas.

## Imagens e containers do Docker

Na raiz do sistema, no nível da solution, existe um arquivo docker-compose.yml. Abrir um powershell nessa pasta e executar o comando abaixo:

### *Docker-compose up -d*

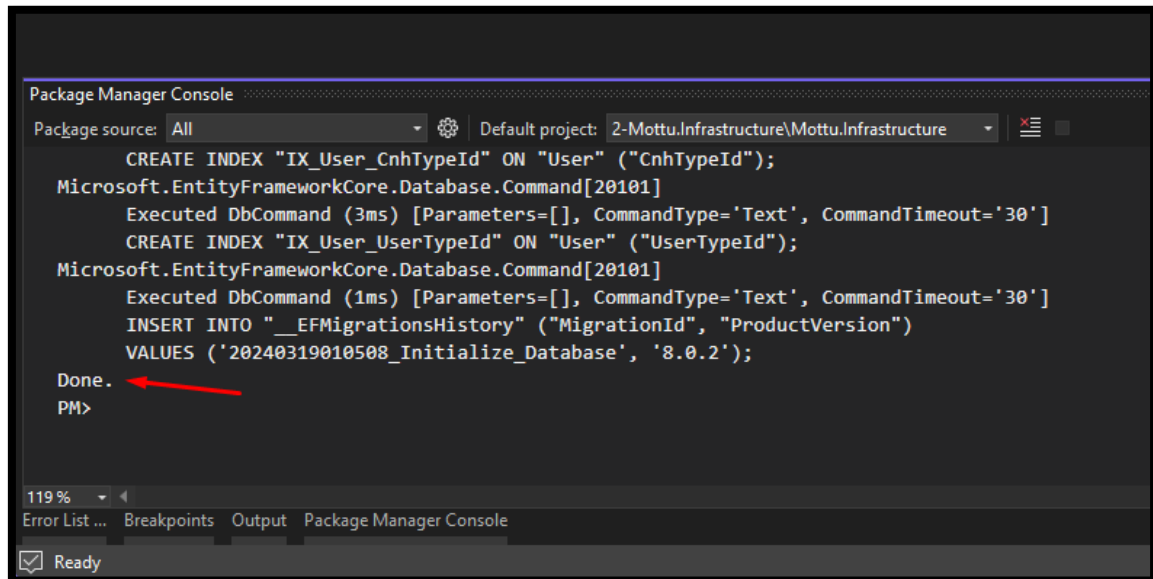
Uma vez executado com sucesso, os containers a seguir serão apresentados, já em execução.



A seguir, deve-se abrir a solution do projeto, abrir uma **Package Manager Console**, selecionar o projeto **Mottu.Infrastrucure** na dropdown Default Project, e rodar o comando abaixo que a migration e inicialização do banco seja executada:

### *Update-database*

A mensagem **Done** indicará que a criação do banco de dados ocorreu com sucesso.



```
Package Manager Console
Package source: All Default project: 2-Mottu.Infrastructure\Mottu.Infrastructure
CREATE INDEX "IX_User_CnhTypeId" ON "User" ("CnhTypeId");
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
CREATE INDEX "IX_User_UserId" ON "User" ("UserId");
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
INSERT INTO "__EFMigrationsHistory" ("MigrationId", "ProductVersion")
VALUES ('20240319010508_Initialize_Database', '8.0.2');
Done.
PM>
```

Utilize um gerenciador de banco de dados para abrir o modelo e visualizar o banco criado. A sugestão é usar o **Dbeaver**.



Para conectar o banco, informe os dados conforme apresentados no print abaixo. O usuário é **postgres**, a senha **12345** e o nome do banco **MottuDB**.

Conectar a um banco de dados

## Configurações de conexão

Configuração de conexão para PostgreSQL

Principal PostgreSQL Driver properties SSH SSL + Network configurations...

Servidor

Conecte usando: ☒ Host ☐ URL

URL:

Host:  Porta:

Banco de dados:  ☐ Exibir todos os bancos de dados

Autenticação

Autenticação:

Nome de usuário:

Senha:  ☒ Salvar senha

Advanced

Role da sessão:  Cliente local:

[Você pode usar variáveis nos parâmetros de conexão.](#) [Detalhes da conexão \(nome, tipo, ...\)](#)

Nome do driver: PostgreSQL [Configurações de driver](#) [Driver license](#)

[Testar conexão ...](#) [< Voltar](#) [Avançar >](#) [Concluir](#) [Cancelar](#)

MottuDB - localhost:5432

- Bancos de dados
  - MottuDB
    - Schemas
      - public
        - Tabelas
 

> Accepted_Orders	16K
> Bike	24K
> Cnh_Type	24K
> Delivered_Orders	16K
> Notificated_User	16K
> Notification	16K
> Order	24K
> Plan	32K
> Rental	32K
> Status_Order	24K
> User	32K
> User_Type	24K
> __EFMigrationsHistory	24K
        - Foreign Tables

## Descrição das Entidades do Banco

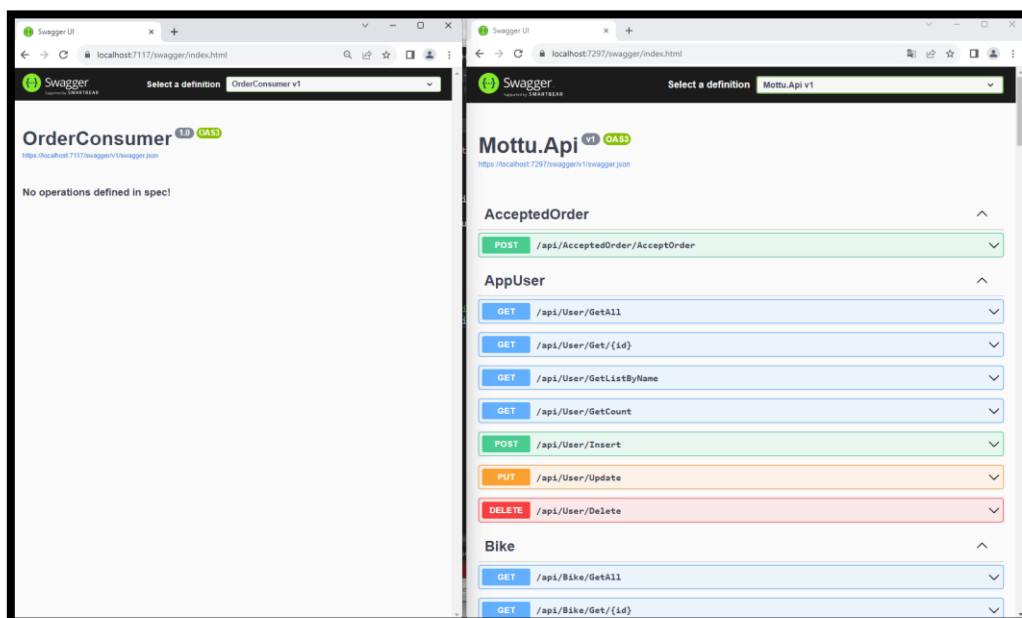
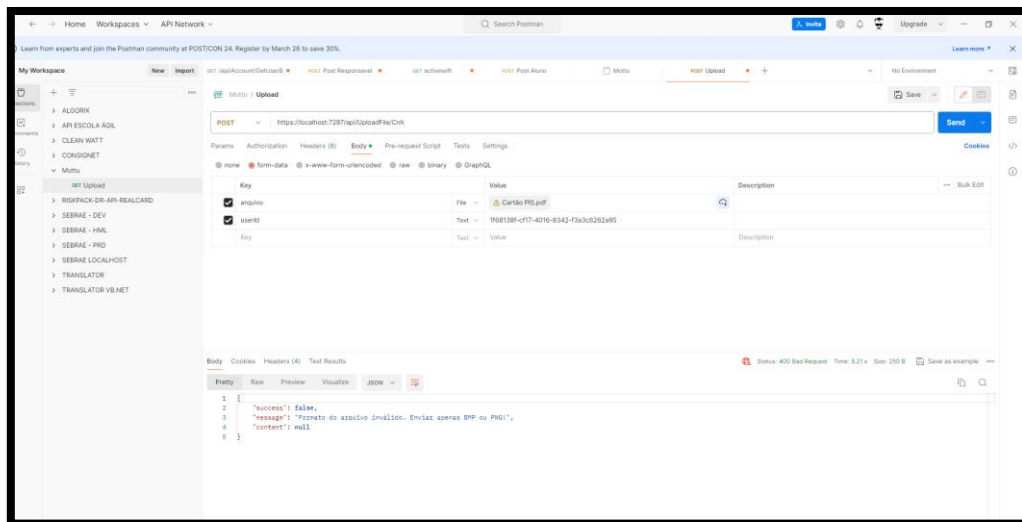
- Accepted\_Orders (Pedidos Aceitos)
- Bike (Moto)
- Cnh\_Type (Tipo de Cnh)
- Delivered\_Orders (Pedidos Entregues)
- Notificated\_User (Usuários Notificados)
- Notification (Notificações)
- Order (Pedidos)
- Plan (Planos)
- Rental (Locações)
- Status\_Order (Status dos Pedidos)
- User (Usuários)
- User\_Type (Tipo de Usuário)

## Como testar

Para testar a aplicação, deve por dois projetos pra iniciarem simultaneamente. Um é o projeto **Mottu.Api** e o outro é o projeto **OrderConsumer**, de forma a permitir ver o processo de mensageria sendo executado.

Uma vez em execução, deverão abrir duas ou três abas do navegador, sendo que duas (abaixo) são as que estão envolvidas no teste do sistema com o uso do Broker de mensageria.

Para realizar o teste de upload de imagem, é necessário utilizar um programa que faça a requisição do serviço passando a imagem e a chave do usuário. Utilize o Postman que atende todos os requisitos pra realizar a requisição e fazer todos os testes.



## Observação:

Ao executar a solução com dois projetos ativos, algumas vezes podem ocorrer das abas dos dois serviços se sobreporem e a aba com os endpoints da aplicação não aparecer.

Observar o carregamento das abas para caso isso ocorra, dar um back no navegador, para ter acesso a aba da direita (acima na imagem).

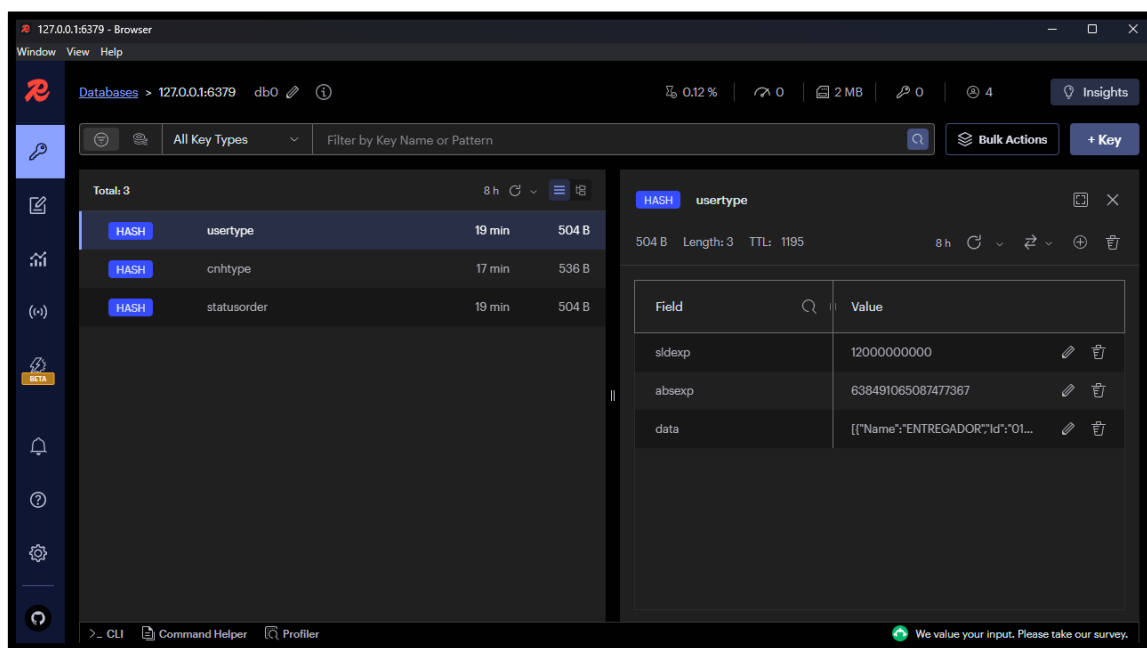
Outro ponto que pode ocorrer ao executar o projeto, é da aba de endpoints não aparecer. Basta dar um clean solution e rebuild solution que resolverá o problema.

## Redis

Para observar os caches gerados no Redis, aconselha-se baixar a ferramenta Redis Insight através do link [RedisInsight - The Best Redis GUI](#).

O Redis Insight é uma ferramenta gráfica que facilita a consulta e o acompanhamento dos repositórios/caches criados no Redis, conforme pode ser observado na imagem abaixo.

No lado esquerdo, pode-se ver os 3 caches criados para armazenamento dos **UserTypes**, **CnhTypes** e **StatusOrders**. E seleccionando o cache, temos acesso ao conteúdo do que foi armazenado.





## Diferenciais

O projeto contempla como diferenciais:

- Uso de ORM por meio do EF Core
- Uso de containers Docker com o controle de criação dos containers por meio de um documento Docker Compose. O sistema faz uso de 3 containers, sendo um do banco de dados (PostGreeSql), outro de mensageria (RabbitMQ) e o último, para sistema de cache distribuído (Redis)
- Adoção de design patterns
- Documentação, além do swagger e de comentários ao longo do código
- Tratamentos de erros em diversos pontos do sistema, buscando a maior estabilidade ao projeto
- Testes unitários através do Xunit.Net
- Solução dividida em layers, adotando arquitetura limpa ou Clean Architecture, como é comumente conhecida
- Modelagem de dados por meio de classes e geração da estrutura de banco pela abordagem Code-First com uso de migrations
- Geração de logs em alguns pontos na classe Consumer que atende ao Broker de mensageria RabbitMQ
- Adoção de convenções de codificação, adotadas pelo mercado de desenvolvimento
- Todo o código desenvolvido em inglês, mas comentários e mensagens do sistema ainda permanecendo em português

## Download da solução

O projeto se encontra versionado no endereço [olivertech/MottuSolution \(github.com\)](https://github.com/olivertech/MottuSolution)