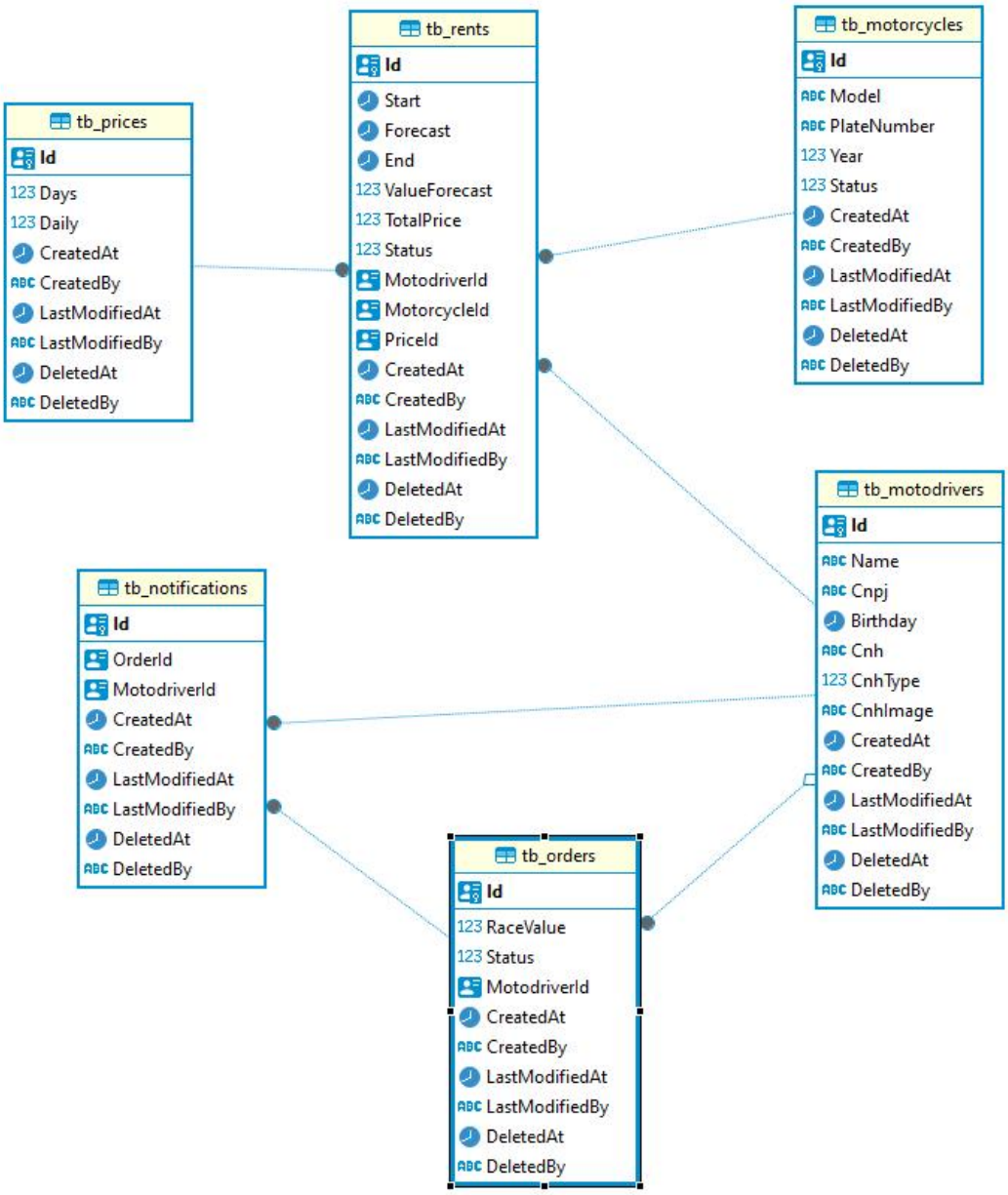
**NOTAS TÉCNICAS**

Preferi escrever esse documento a parte que servirá de apoio ao entendimento do software com maior nível de detalhamento em cada ponto, em cada caso de uso do sistema. Dessa forma, pretendo aprofundar mais, cada decisão tomada e ir respondendo, um a um, cada requisito proposto.

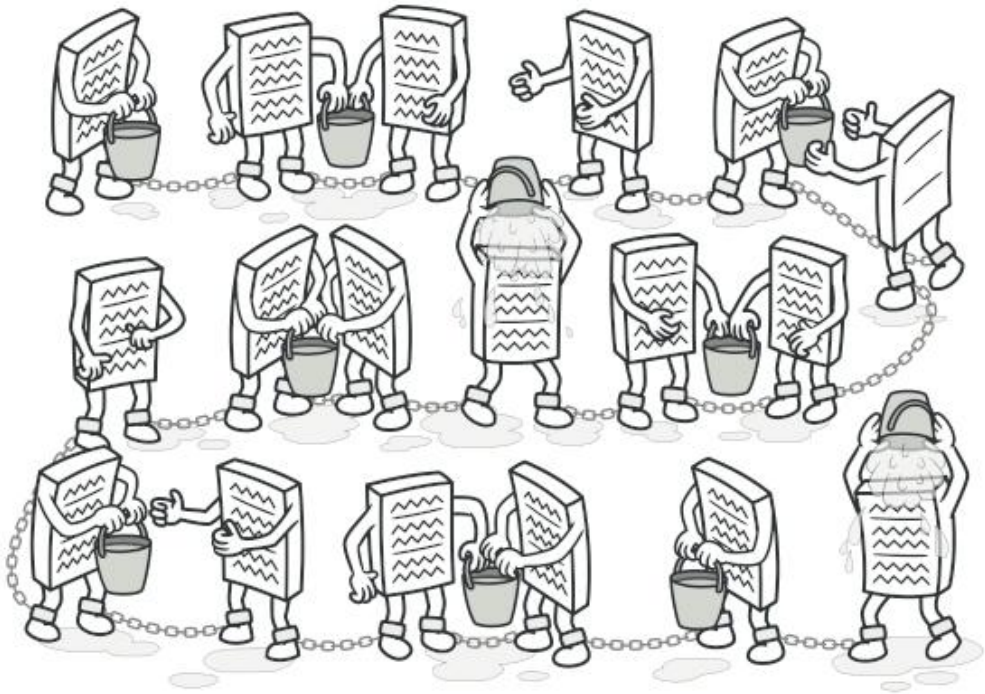
Queria começar por responder porque decidi usar a combinação DAPPER + EF; trata-se de uma preferência pessoal, apenas. Eu gosto da facilidade que o EF me entrega para modelar as entidades e relacionamentos, mas já prefiro o DAPPER para escrita dos repositórios e são vários os motivos, por experiências que já vivi ao longo do tempo; primeiro porque o DAPPER performa melhor que o EF) e mais: mesmo que não tem facilidade com "linq" e com programação, ou mesmo nem sabe programar, mas sabe escrever uma consulta SQL pode entrar na solução e dar manutenção num momento mais crítico. Contundo, o histórico de alterações no banco de dados; criação de novas tabelas e chaves, índices, inclusão de novos campos... enfim, a possibilidade das migrations automáticas me encantam.

**DIAGRAMA DO BANCO DE DADOS**



**PADRÃO DE PROJETO**

Usei um dos padrões que mais uso no dia a dia, o “Chain of Responsability”. Trata-se de um padrão de projeto comportamental, que nos permite passar pedidos numa corrente de “handlers”. Ao receber um request, cada handler decide se processa o pedido ou passa a diante, para o próximo “handler” na corrente.



Ele nos oferece muitas vantagens na sua adoção. Nos facilita novas implementações, modificações, incremento de novas features “handlers” e com isso temos classes geralmente muito curtas e objetivas, que atendem um único propósito, uma única intenção. É o primeiro princípio do SOLID (princípio da responsabilidade única). Quando houver necessidade de incrementar uma nova regra de negócio no seu caso de uso, você cria uma nova “handler” e posiciona essa nova classe no local mais adequado da esteira do caso de uso.

Como já disse, são classes bem simples -- *geralmente nem criam “scroll”* -- de fácil entendimento e manutenção e simples de serem testadas. Daí, o que geralmente **EU** faço, especialmente quando usamos o CQRS (é o nosso caso) eu excluo o “Handler” principal (Mediator) do teste unitário e da cobertura de teste; porque nunca haverá regra de negócio ali; ele apenas recebe a requisição, põe na fila de execução (“Chain of responsability”) e aguarda o resultado processado; no sucesso ou falha.

Em cada etapa, o objeto da requisição recebe novos atributos que passa para frente. Já a etapa seguinte checa se ocorreu algum erro anterior (se sim, nem segue a esteira) e devolve o resultado para o cliente, senão resolve o que é de sua responsabilidade, volta a incrementar algo novo no objeto e salta pro próximo “handler”... e esse processo vai até o fim de cada caso de uso.

Segue exemplo abaixo do  caso de uso **RentUseCase**.

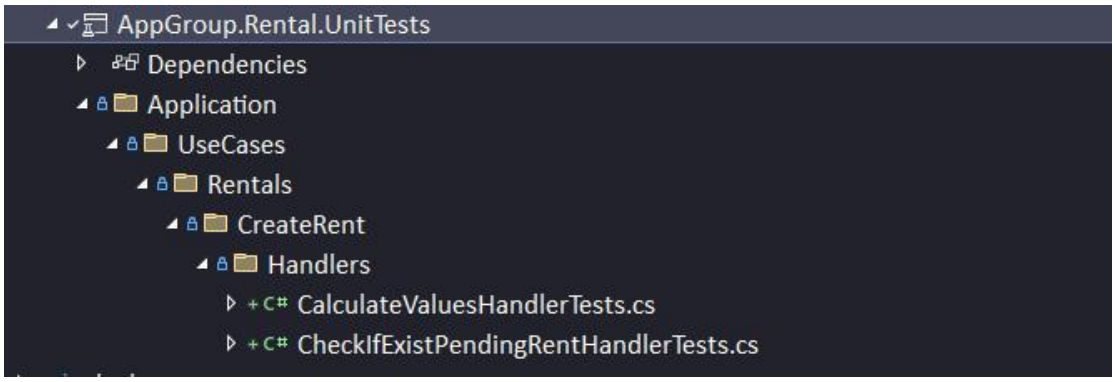
```
var h1 = new CheckIfExistPendingRentHandler(_rentRepository);
var h2 = new GetDataDriverHandler(_motodriversRepository);
var h3 = new GetDataPriceHandler(_motorcyclesRepository);
var h4 = new GetDataMotoHandler(_motorcyclesRepository);
var h5 = new CalculateValuesHandler();
var h6 = new CreateProposalHandler(_rentRepository);

h1.SetSuccessor(h2);
h2.SetSuccessor(h3);
h3.SetSuccessor(h4);
h4.SetSuccessor(h5);
h5.SetSuccessor(h6);

await h1.Process(request);
```

Note como fica fácil o entendimento do caso de uso, (mesmo para quem não programa) quais etapas são necessárias para entregar aquele resultado; e como também fica fácil incrementar uma nova regra de negócio; deve ser feito pequeno ajuste na sequencia de execução e colocar para testar. O “handler” principal só tem o papel de receber a requisição e responder ao usuário o que foi produzido.

**TESTES REQUERIDOS**

Quanto aos testes, eu optei por apenas demonstrar como geralmente organizo meus testes, especialmente os testes unitários, que organizo exatamente como a estrutura da camada de aplicação. Como o tempo foi curto, optei por apenas demonstrar. Priorizei no entendimento do negócio e na estruturação do projeto e até tomei a liberdade de incluir novas regras que não estavam previstas no desafio, por exemplo: o entregador não poder devolver a moto com pedido de entrega em aberto, ou ainda, (também não previsto) assim que o pedido é gerado todos os entregadores serão notificados, porém serão muitos pedidos e muitos entregadores, então que as entregas sejam atendidas das mais antigas para as mais recentes. O entregador não vai poder escolher o pedido; ele checa se há pedidos pendentes, se sim, será do mais antigo ao mais novo; com critério na data de criação do pedido.

Quanto a organização dos testes:




Segue a mesma organização de pastas da aplicação; fica fácil de encontrar; ninguém se perde. Quanto aos teste de integração é importante ressaltar, que primeiro deve-se levantar o docker-compose (via terminal), aguardar os conteiners ficarem ativos e só a partir daí vocês podem abrir o Visual Studio e executar os testes de integração. Combinado? Contudo, devo confessar que não sou amante de testes de integração, prefiro outros tipos de testes: por exemplo, o teste de carga, que nos alerta se algo pode ser melhorado, alguma rotina, alguma query no banco pode ser refatorada, algum processo precise ser revisto, enfim. Algo que nos alerte realmente; algo que aponte se estamos com o melhor código, indo pelo melhor caminho ou se devemos rever alguma coisa para performar melhor.

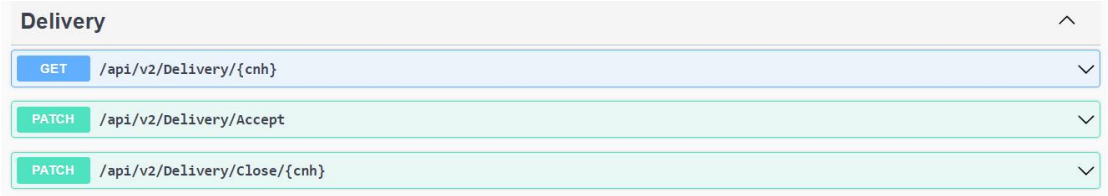
**VERSIONAMENTO DE API's**

Como trata-se de uma POC... OK, a implementação que adotei, para criar duas versões distintas numa mesma API (versão de administrador e outra de usuário) funciona bem e entrega o que o cliente quer, porém numa eventual evolução do projeto, vai nos levar a quebra em duas API's distintas ou mais; dois ou mais micro-serviços, por exemplo: pode-se criar um micro-serviço exclusivo de usuários, outro só para motos, um para contratos, um para entregas, outro para pagamentos, um para notificação... enfim, são infinitas possibilidades. A evolução do projeto vai nos guiar.

**NOTIFICAÇÃO DE USUÁRIOS**

Não ficou 100% claro para mim o que vocês pretendiam aqui, exemplo, enviar um email pro usuário, um SMS, notificar o aplicativo... não entendi, então tomei a liberdade de simplificar o processo ao máximo. No caso de uso de criação de pedido "**CreateOrderUseCase"**; assim que o pedido é gerado e salvo no banco de dados, uma mensagem é disparada automaticamente numa fila do RabbitMQ "create-order"; imediatamente depois, quase que de forma instantânea um consumidor pega esse pedido, volta no banco de dados para filtrar todos os entregadores que atendam os critérios definidos na documentação e gera registros na tabela de notificação, afinal o administrador quer saber quem estava disponível naquele momento para realizar a entrega; o primeiro entregador que bater no endpoint abaixo "GET", vai conferir se há entrega disponível na plataforma ou não, se sim, ele opta por aceitar ou não aquele pedido.

Observe:

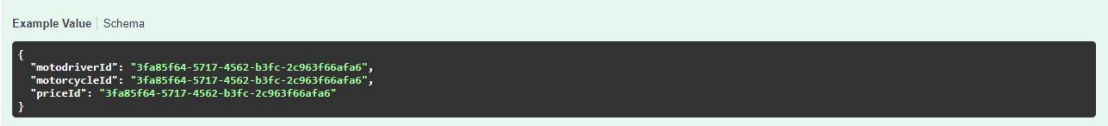



O primeiro endpoint serve para buscar os pedidos pendentes, lembrando dos mais antigos aos mais recentes. Se sim, irá apresentar o ID do pedido. O entregador terá a opção de aceitar o pedido ou não. Se sim, pega aquele ID e vai para o segundo endpoint "Accept". Essa ação fará alteração na tabela de pedidos, atribuindo o ID do entregador ao pedido e o status do pedido para "aceito", além de alterar a data da última atualização daquele pedido. Feito isso, a rotina volta na tabela de notificações para liberar aquele pedido da fila e demais entregadores já não irão mais enxergar aquele pedido aceito.

Assim que o entregador realiza a entrega, ele deve registrar no último endpoint: "Close" para ficar disponível para realizar novas entregas. Nenhuma linha da tabela de notificações será removida, afinal serve de base de relatório e decisões do time de administradores. O entregador não pode pegar mais de um pedido por vez.

**ALUGUÉIS**

O contrato de aluguel é gerado pelo administrador, que irá fornecer as seguintes informações:




O registro do aluguel da moto é gerado com base nos seguintes dados (ID do entregador + ID do veículo + ID da tabela de preços); uma vez gerado, esse contrato vai gerar um novo ID que deve ser aceito no endpoint "Patch".

Só depois disso o entregador estará apto para receber as notificações e realizar as entregas. O administrador pode consultar os dados da proposta e encerrar o contrato, desde que não haja pendência de entrega daquele entregador; e multas serão geradas (de acordo com a documentação se este entregar antes ou depois do prazo)

**REGRAS IMPORTANTES**

- Quanto aos administradores;
  - No momento da geração do pedido, os administradores poderão gerar pedidos (independentemente se há ou não, entregadores disponíveis naquele momento), contudo, a rotina só irá gerar notificações automáticas no caso de HAVER entregadores disponíveis (e esse processo ocorre em lote), o sistema lista todos os entregadores disponíveis (que atendam os critérios definidos por regra de negócio) e gera todas as notificações de uma vez; do contrário, caso não haja entregadores, a rotina não tem como gerar;
  - O papel da notificação é criar um relatório para saber quais entregadores estavam disponíveis para aceitar os pedidos (no momento de sua geração); se não há entregadores, essa notificação perde o sentido e será gerada noutro momento. E porque fiz assim? Porque novos entregadores estarão sendo cadastrados a todo instante e esse novo entregador ainda pode (e deve) receber os pedidos que os administradores estarão gerando; esse é o CORE do negócio;
  - Esse novo entregador (uma vez apto para receber pedidos), irá conferir no endpoint GET da controller (**Delivery - v2**) se há pedidos pendentes de entrega, bastando informar sua **CNH**, se sim, o sistema gera ali a notificação, informando o ID do pedido; ele logo deve pegar esse **ID** e aceitar a entrega, impedindo que outro entregador pegue o entrega;
- Quanto aos entregadores;
  - Os entregadores só poderão aceitar apenas um único pedido por vez; o sistema confere antes se há pedidos pendentes vinculados a sua CNH; se sim, o entregador recebe uma mensagem de erro, informando que há pedidos pendentes; sendo assim, torna-se obrigatório o registro da conclusão da entrega;
  - Disponibilizamos um enpoint para o entregador buscar detalhes do pedido aceito (ele não precisa anotar o ID do pedido em nenhum lugar, que seria uma loucura); para isso basta informar o número de sua **CNH** no endpoint ("/api/v2/Delivery/Details/{cnh}") e ele terá, por exemplo: o ID do pedido, valor, quando foi criado, onde deve ser entregue...
  - Os entregadores não podem encerrar um contrato de aluguel com pendências de entrega; caso o entregador tente encerrar o contrato da moto, estando este com entrega pendente, ele receberá uma mensagem de erro informando.
  - Os entregadores só poderão ter um único contrato de aluguel ativo por vez; caso ele tente gerar um novo contrato, estando com um contrato ativo, o sistema devolve uma mensagem de erro informando.
  - O entregador terá autonomia para criar seu próprio cadastro, contudo ele deve informar um **CNPJ** e um **CNH** válidos; lembrando que apenas os entregadores que possuam a categoria "A" será capaz de receber pedidos de entrega; ou seja, o sistema é capaz de registrar algueisl para qualquer pessoa, mas apenas aqueles com a categoria "A" da **CNH** poderão receber os pedidos de entrega;
  - No momento da geração do contrato de aluguel, depois que o usuário fornecer todos os dados obrigatórios (definidos em regra de negócio) o sistema gera um registro na tabela de alugueis, porém com status **OPEN** (trata-se inicialmente de uma proposta) que **DEVE** ser

aceita para valer como contrato ativo; do contrário (caso ele não aceite o contrato), outro usuário poderá selecionar a moto que o primeiro usuário escolheu (e não deu aceite); aquela moto ainda estará disponível para aluguel para outro usuário;

**PADRÃO REST**

Nesse ponto gostaria de mencionar que usei os verbos adequados para cada "endpoint", assim como seus respectivos StatusCode, desse jeito, decidi não detalhar a fundo cada recurso da API, porque seguem normais internacionais:

- Verbos:
  - POST - cria um novo recurso, seu retorno deve ser o 201 (no caso de sucesso);
  - PATCH - faz pequenos ajustes no recurso, pequenas atualizações;
  - UPDATE - Não fizemos uso desse verbo na solução;
  - GET - Serve para as pesquisas;
  - DELETE - Há um endpoint com esse verbo (mas segue as regras definidas no requisito);

**AGRADECIMENTOS**

Gente foi uma correria imensa!!!! Mas vocês não imaginam como foi gostoso e desafiador construir esse projeto para vocês, coloco-me 100% disponível para conversar e tirar dúvidas. Estou na torcida de ir trabalhar com vocês, de verdade!

Grande abraço.