Desafio Back-end em C# - ANDREW NAVARRO REIS E SILVA

Da Proposta

1. Descrição do Desafio

- Sistema Bancário
- Há 2 tipos de usuários, os comuns e lojistas, ambos têm carteira com dinheiro e realizam transferências entre eles.
- Para ambos tipos de usuário, precisamos do Nome Completo, CPF/CNPJ, e-mail e Senha. CPF/CNPJ e e-mails devem ser únicos no sistema. Sendo assim, seu sistema deve permitir apenas um cadastro com o mesmo CPF ou endereço de e-mail.
- Usuários podem enviar dinheiro (efetuar transferência) para lojistas e também entre usuários.
- Lojistas **só podem receber** transferências, não enviam dinheiro para ninguém.

2. Requisitos

- Validar se o usuário tem saldo antes da transferência.
- A operação de transferência deve ser uma transação (ou seja, revertida em qualquer caso de inconsistência) e o dinheiro deve voltar para a carteira do usuário que envia.
- No recebimento de pagamento, o usuário ou lojista precisa receber notificação (envio de email) enviada por um serviço de terceiro e eventualmente este serviço pode estar indisponível/instável. Por usar qualquer serviço de envio de email, como por exemplo https://www.mocklab.io/docs/mock-rest-api/
- Este serviço deve ser RESTFul.

3. Diferencial

- Uso de Docker
- Testes de integração
- Testes unitários
- Uso de Design Patterns
- Documentação
- Proposta de melhoria na arquitetura

4. Avaliação

- Apresentar sua API funcionando
- Documentação
- Se for para vaga sênior, foque bastante no **desenho de arquitetura**
- Código limpo e organizado (nomenclatura, etc)
- Conhecimento de padrões (PSRs, design patterns, SOLID)
- Ser consistente e saber argumentar suas escolhas
- Apresentar soluções que domina
- Modelagem de Dados
- Manutenibilidade do Código
- Tratamento de erros
- Cuidado com itens de segurança
- Arquitetura (estruturar o pensamento antes de escrever)
- Carinho em desacoplar componentes (outras camadas, service, repository)

Do desenvolvimento

Banco de dados

MSSQL utilizado em máquina com servidor local, utilizando banco de dados criado com nome ADCDB (Curto para App De Casa Database). Usuário criado para acessar o banco (Local): [redacted] com senha: [redacted]. O login remoto através do Docker foi testado com usuário *sa* e senha Banco01!, o qual estava operacional conforme arquitetura.

Tabelas a serem utilizadas:

- **Pessoa**
  - ID, CPF, E-mail, Senha, Nome, Saldo
- **Loja**
  - ID, CNPJ, E-mail, Senha, Nome, Saldo

Melhorias na Arquitetura do Banco de dados

O Banco está planejado apenas para guardar informações atuais, impossibilitando busca de histórico para transações passadas. Modificar

a arquitetura para permitir o armazenamento de histórico seria uma grande melhoria no sistema. Além disso, em um sistema real se usaria uma senha criptografada para armazenagem no banco. Código para geração dos dados está anexado junto a este arquivo, onde

Representação Gráfica

Modelagem

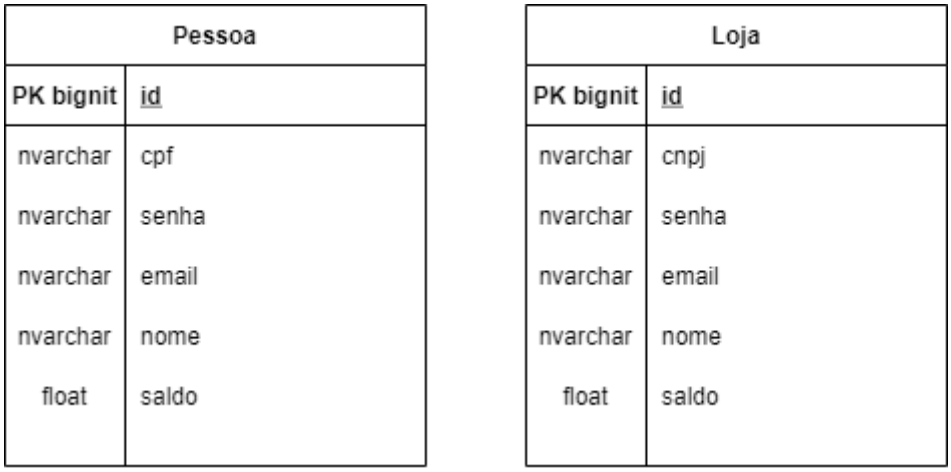


Figura 1, elaborado pelo autor.

Docker-Compose

O Docker foi utilizado durante o desenvolvimento, mas devido a um erro desconhecido e extrema lentidão devido à baixa quantidade de memória RAM da minha máquina foi abandonado pois a migração de dados não estava executando de forma correta, necessário algum ajuste.

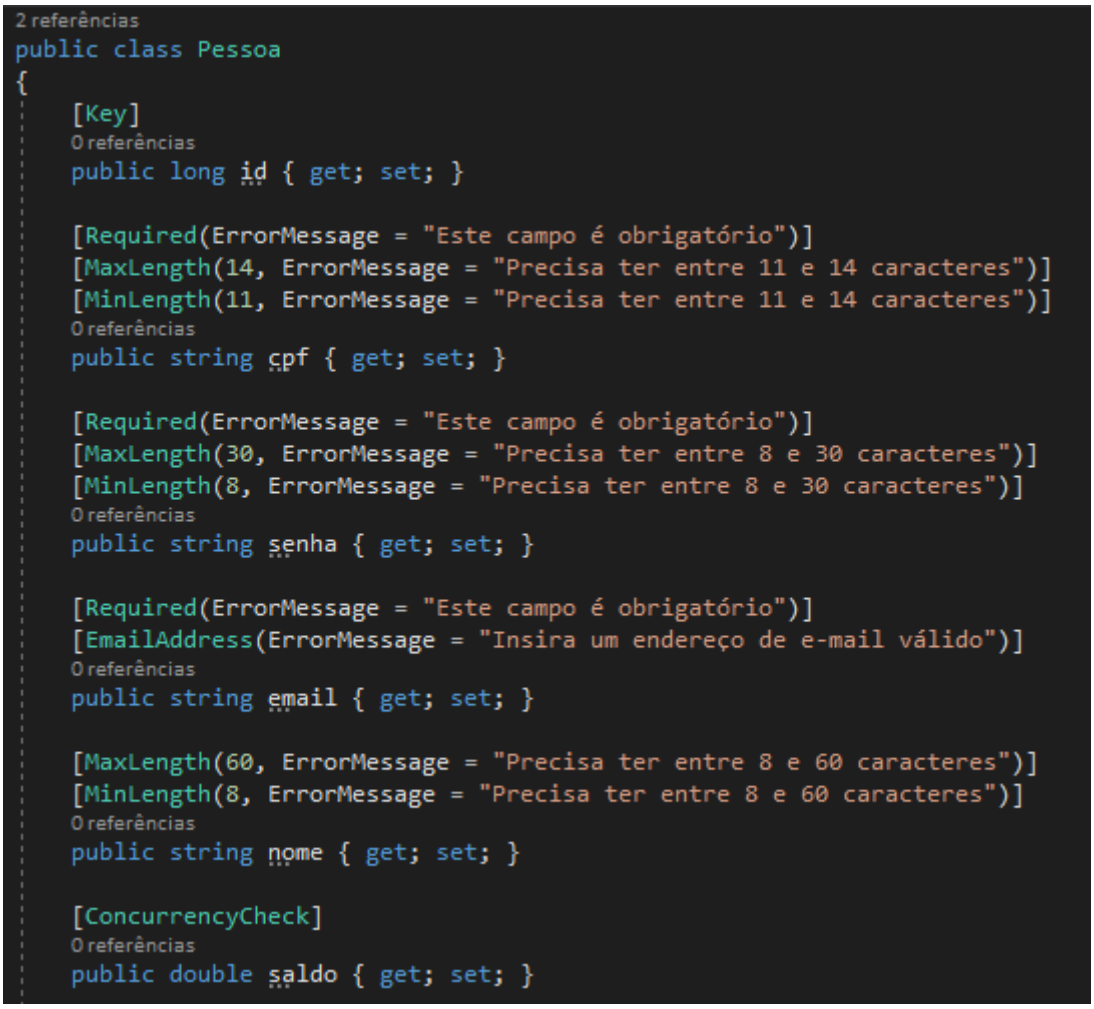
Os containers e a comunicação estão funcionando normalmente se o banco for populado manualmente.

Código Gerado das Classes

Modelos

Por ser um sistema simples, as classes geradas são bem similares ao banco de dados, facilitando a utilização do *EntityFramework* aplicado com *DbContext*. A utilização de anotações da biblioteca `DataAnnotations` permite o uso de validação ao receber classes para persistir no banco, garantindo maior coerência e consistência das informações salvas. As classes seguirão este padrão, sendo que a classe de `Loja` terá como diferença o CNPJ no lugar do CPF.

Classe de Pessoa em C#



```
2 referências
public class Pessoa
{
    [Key]
    0 referências
    public long id { get; set; }

    [Required(ErrorMessage = "Este campo é obrigatório")]
    [MaxLength(14, ErrorMessage = "Precisa ter entre 11 e 14 caracteres")]
    [MinLength(11, ErrorMessage = "Precisa ter entre 11 e 14 caracteres")]
    0 referências
    public string cpf { get; set; }

    [Required(ErrorMessage = "Este campo é obrigatório")]
    [MaxLength(30, ErrorMessage = "Precisa ter entre 8 e 30 caracteres")]
    [MinLength(8, ErrorMessage = "Precisa ter entre 8 e 30 caracteres")]
    0 referências
    public string senha { get; set; }

    [Required(ErrorMessage = "Este campo é obrigatório")]
    [EmailAddress(ErrorMessage = "Insira um endereço de e-mail válido")]
    0 referências
    public string email { get; set; }

    [MaxLength(60, ErrorMessage = "Precisa ter entre 8 e 60 caracteres")]
    [MinLength(8, ErrorMessage = "Precisa ter entre 8 e 60 caracteres")]
    0 referências
    public string nome { get; set; }

    [ConcurrencyCheck]
    0 referências
    public double saldo { get; set; }
```

Figura 2, elaborado pelo autor.

Data

Gerado automaticamente pelo *Visual Studio*, Pacote onde está localizado o `EntityFramework`, utilizado neste projeto através do *DbContext* e chamado de `DesafioDeCasaContext`.

Repositórios

Utilização de um Repositório genérico para controle das funções CRUD padrão (Adicionar, Remover, Buscar) já presentes no *DbContext* para eliminação de código repetido.

Uso da classe genérica com as classes tipadas para implementação dos métodos genéricos e da implementação de métodos específicos por classe, necessários especificar.

Serviços

Classes de service foram feitas para utilizar as funções da classe de repositórios, que possuem os métodos de acesso ao banco de dados, de modo a encapsular as chamadas desses métodos e deixar a classe controladora sem conexão direta com a implementação dos serviços.

Controladores

Criados para utilizar das classes de *service* de acordo com sua classe, permitindo o acesso à implementação do *EntityFramework* para a API de acordo com as rotas implementadas.

Melhorias no Controladores

Visando melhorar o desempenho e a usabilidade da API, uma melhoria a ser efetuada na arquitetura seria a utilização de métodos e atributos assíncronos, o qual não entendo bem o suficiente para aplicar nesta demo, apesar de conhecer os conceitos.

Gerenciador de E-mail

O Gerenciador de e-mail escolhido foi através do framework nativo de e-mail do asp.net. Feito através do servidor do Gmail com um e-mail novo criado e escolhido para ser o protótipo desta funcionalidade. E-mail: noreply.adecasa@gmail.com. É feito a autorização no servidor da google, que passa a autorizar o envio remoto de e-mail pela API.

Caso de Uso

Existem 2 controladores, o de Pessoas e o de Lojas, sendo assim os comandos disponíveis são:

- http://Localhost:44355/
    - Pessoas (GET)
        - Recupera a lista de todas as pessoas
    - Pessoas (POST)
        - Salvar uma entidade de Pessoa no banco

- Pessoas/{id} (DELETE)
  - Remove uma entidade do banco
- Pessoas/Atualizar/{id} (PUT)
  - Atualiza uma entidade do banco
- Pessoas/{idPagador}/PagarPessoa/{idRecebedor}/{valor} (GET)
  - Paga um valor a uma pessoa
- Pessoas/{idPagador}/PagarLoja/{idRecebedor}/{valor} (GET)
  - Paga um valor a uma loja
- Pessoas/{id} (GET)
  - Recupera uma pessoa específica
- Lojas (GET)
  - Recupera a lista de todas as pessoas
- Lojas (POST)
  - Salvar uma entidade de Pessoa no banco
- Lojas/{id} (DELETE)
  - Remove uma entidade do banco
- Lojas/Atualizar/{id} (PUT)
  - Atualiza uma entidade do banco
- Lojas/{idPagador}/PagarPessoa/{idRecebedor}/{valor} (GET)
  - Informa que esta ação não está disponível
- Lojas/{idPagador}/PagarLoja/{idRecebedor}/{valor} (GET)
  - Informa que esta ação não está disponível
- Lojas/{id} (GET)
  - Recupera uma pessoa específica

Desenvolvido por Andrew Silva (andrewnrs@outlook.com).

12/10/2021 – São Luís MA.