CORE-BANKING-SYSTEM

Projeto consiste em basicamente em simular aspectos de uma conta bancária.
Dentre os endpoints expostos, destaca-se 2 (dois) principais:

•**SALDO**: Permite que seja verificado o saldo bancário de um determinado usuário.

•**TRANSAÇÃO**: Possibilita que seja efetuada operações de crédito (depósito de valores) ou de débito (retirada de valores) em uma conta corrente específica.

Embora não fosse fundamental, alguns outros endpoints também foram desenvolvidos no intuito de auxiliar nos testes e cenários:

•Listar todas as contas.

•Listar uma conta específica pelo seu identificador.

•Inserir uma nova conta para uma determinada agência.

Spring WebFlux

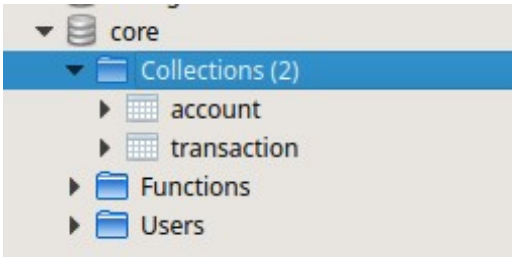
Este projeto foi desenvolvido em Java (versão 11), utilizando Spring 5 e framework funcional/reativo WebFlux.
Optou-se por esta abordagem devido a questões como concorrência e conflitos transacionais.

Banco de Dados

Neste projeto foi utilizado o banco de dados não-relacional MongoDB.
Ao subir a aplicação as estrutura de Collections/Documents necessárias são automaticamente criadas.

O banco de dados foi utilizado para armazenar o saldo bancário das contas, além de manter um histórico das transações efetuadas. isto é efetuado nas respectivas

collections a seguir: *account* e *transactions*.



Portas Utilizadas

Esta aplicação utiliza as portas 8082 (core-banking-system) e 27017 (mongoDB).
Caso a aplicação seja rodada localmente, certifique-se que estas portas estejam liberadas ou, caso preferir, altere-as no arquivo *application.properties* do projeto.

Dummy Data / Dados para testes

Ao inicializar a aplicação, são inseridas 15 (quinze) contas de usuários para fins de simulações e testes.
A inserção dos dados pode ser vista na seguinte
classe: *br.com.bank.core.data.AccountDummyData*
A collection no qual estes dados ficam armazenados é: *Account*

Docker / Inicialização

A aplicação funciona em qualquer sistema operacional (incluindo Linux e MacOS).
Foi utilizado Docker, justamente para garantir tal possibilidade, pois desta forma automaticamente a aplicação indica seus requisitos e os obtêm.
Os arquivos *Dockerfile* e *docker-compose.yml*, ambos localizados na raiz do projeto, detalham melhor isto.

Por fim, abaixo seguem alguns comandos para inicialização da aplicação e montagem de imagens do docker.

- Iniciando aplicação:
    - Por linha de comando, na pasta raiz do projeto, digite:
      docker-compose up
- Parando aplicação:
    - Por linha de comando, na pasta raiz do projeto, digite:
      docker-compose stop
- Criando uma imagem da aplicação:
    - Por linha de comando, na pasta raiz do projeto, digite:
      docker build -t core-banking-system:latest .

Endpoint para Visualizar Saldo de Uma Determinada Conta

- GET Request:
  http://localhost:8082/account/{ACCOUNT}/branch/{BRANCH}/balance
- Success Response Example:
  No exemplo abaixo, a cointa 44758-1, da agência 0001, possui 204 reais de saldo. O ID da conta é o "5d231a66a7b11b00012dc077".


```
{
    "meta": {
        "processName": "/account/44758-1/branch/0001/balance",
        "status": "success"
    },
    "data": {
        "id": "5d231a66a7b11b00012dc077",
        "branchNumber": "0001",
        "accountNumber": "44758-1",
        "balance": 284
    }
}
```

Endpoint para Realizar Uma Transação de Crédito ou Débito

•POST Request:
http://localhost:8082/transaction
•Payload Example:
Neste exemplo, enviaremos um débito (DEBIT) de 3 reais para a agência 0001 e conta 44758-1.
Para enviar um crédito, basta informar CREDIT no transactionType.

```
{
        "account": {
        "branchNumber": "0001",
        "accountNumber": "44758-1"
        },
        "transactionType" : "DEBIT",
        "amount": 3
}
```

•Success Response Example:
Neste exemplo foi retornada a confirmação da transação de ID 5d2385c6a7b11b00012dc07c, a qual foi debitado o montante de 3 reais da conta de ID 5d231a66a7b11b00012dc077 e, por conseguinte, seu novo saldo ficou em 281 reais.

```
{
    "meta": {
        "processName": "/transaction",
        "status": "success"
    },
    "data": {
        "id": "5d2385c6a7b11b00012dc07c",
        "account": {
            "id": "5d231a66a7b11b00012dc077",
            "branchNumber": "0001",
            "accountNumber": "44758-1",
```

```
                "balance": 281
            },
            "transactionType": "DEBIT",
            "amount": 3
        }
}
```


- Error Response Example - Insufficient Funds (Only possible for debit transactions):


```
{
    "meta": {
            "processName": "/transaction",
            "status": "error"
    },
    "error": {
            "msg": "Transaction not effetivated - Insufficient funds",
            "codigo": "ERR-4001",
            "origem": "WS-CORE-BANKING-SYSTEM",
            "tipo": "TEC",
            "subtipo": "TRANSACTION"
    }
}
```


- Error Response Example - Account Not Found:


```
{
    "meta": {
            "processName": "/transaction",
            "status": "error"
    },
    "error": {
            "msg": "Account not found",
            "codigo": "ERR-2001",
            "origem": "WS-CORE-BANKING-SYSTEM",
            "tipo": "TEC",
            "subtipo": "VALIDATION"
    }
```

```
}
```

Arquitetura Futura - Kafka

Considerando o projeto apresentado, na minha concepção o ideal seria aliá-lo a utilização do Kafka, pois desta forma, teríamos um controle maior sobre o processo transacional (operações de crédito e débito).

As requisições transacionais poderiam ser enviadas para um evento no Kafka, o qual o core-banking-system tería um consumer para receber a transação e imediatamente processá-la.

O retorno de sucesso e/ou falha do processamento da requisição também poderia ser disponibilizado em outra fila do Kafka, para posterior consumo por parte do seu requisitante inicial ou demais aplicações e fluxos necessários, como por exemplo rotinas contábeis, rotinas de documentações legais, entre outros.

Tal cenário daria uma segurança maior inclusive no processo de deploy do microserviço do core-banking-system, visto que o Kafka possui ponteiro para o último registro processado. Em caso de falha no microserviço, retornar o ponteiro e recontruir cenários de falhas sería mais fácil.