Projeto 1 - Python Puro

Acesse o material diretamente pelo Notion

```
https://grizzly-amaranthus-f6a.notion.site/Projeto-1-Python-Puro-1936cf8ea89f804ca3b8cec76092d9de?pvs=4
```

▼ Criação do ambiente

Primeiro devemos criar o ambiente virtual:

```
# Criar
    # Linux
        python3 -m venv venv
    # Windows
        python -m venv venv
```

Após a criação do venv vamos ativa-lo:

```
#Ativar
    # Linux
        source venv/bin/activate
    # Windows
        venv\Scripts\Activate

# Caso algum comando retorne um erro de permissão execute o código e tente novamente:

Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

Agora vamos fazer a instalação das bibliotecas necessárias:

```
pip install sqlmodel
```

▼ Criação do banco

Em models.py crie uma classe para representar a tabela responsável pelas contas no banco:

```
from sqlmodel import Field, SQLModel, create_engine
from enum import Enum

class Bancos(Enum):
    NUBANK = 'Nubank'
    SANTANDER = 'Santander'
    INTER = 'Inter'

class Status(Enum):
    ATIVO = 'Ativo'
    INATIVO = 'Inativo'

class Conta(SQLModel, table=True):
    id: int = Field(primary_key=True)
    banco: Bancos = Field(default=Bancos.NUBANK)
    status: Status = Field(default=Status.ATIVO)
    valor: float
```

Agora adicione o código para efetivar a criação da tabela no DB:

```
sqlite_file_name = "database.db"
sqlite_url = f"sqlite:///{sqlite_file_name}"

engine = create_engine(sqlite_url, echo=False)

def create_db_and_tables():
    SQLModel.metadata.create_all(engine)
```

```
if __name__ == "__main__":
    create_db_and_tables()
```

▼ Manipulação da conta

Em view.py crie uma função para que o usuário possa criar contas dentro da aplicação.

```
from models import Conta, engine
from sqlmodel import Session, select

def criar_conta(conta: Conta):
    with Session(engine) as session:
        statement = select(Conta).where(Conta.banco==conta.banco)
        results = session.exec(statement).all()
        if results:
            print("Já existe uma conta nesse banco!")
            return
        session.add(conta)
        session.commit()
        return conta
```

Agora vamos para função responsável em listar todas as contas:

```
def listar_contas():
    with Session(engine) as session:
        statement = select(Conta)
        results = session.exec(statement).all()
    return results
```

Vamos desenvolver a função responsável por desativar uma determinada conta:

```
def desativar_conta(id):
    with Session(engine) as session:
        statement = select(Conta).where(Conta.id==id)
        conta = session.exec(statement).first()
        if conta.valor > 0:
            raise ValueError('Essa conta ainda possui saldo, não é possível desativar.')
        conta.status = Status.INATIVO
        session.commit()
```

E para finalizar a manipulação de contas, vamos desenvolver uma lógica para permitir que o usuário possa transferir o saldo entre contas.

```
def transferir_saldo(id_conta_saida, id_conta_entrada, valor):
    with Session(engine) as session:
        statement = select(Conta).where(Conta.id==id_conta_saida)
        conta_saida = session.exec(statement).first()
        if conta_saida.valor < valor:
            raise ValueError('Saldo insuficiente')
        statement = select(Conta).where(Conta.id==id_conta_entrada)
        conta_entrada = session.exec(statement).first()

        conta_saida.valor -= valor
        conta_entrada.valor += valor
        session.commit()
```

▼ Movimentações financeiras

Nessa primeira etapa desenvolver a funcionalidade do usuário realizar SAÍDAS e ENTRADAS de uma conta.

Para que isso seja possível, vamos criar uma tabela no banco para armazenar o histórico de movimentações

```
class Tipos(Enum):
    ENTRADA = 'Entrada'
    SAIDA = 'Saida'

class Historico(SQLModel, table=True):
    id: int = Field(primary_key=True)
```

```
    conta_id: int = Field(foreign_key="conta.id")
    conta: Conta = Relationship()
    tipo: Tipos = Field(default=Tipos.ENTRADA)
    valor: float
    data: date
```

Com as tabelas criadas, em views.py vamos desenvolver a lógica para movimentações financeiras:

```
def movimentar_dinheiro(historico: Historico):
    with Session(engine) as session:
        statement = select(Conta).where(Conta.id==historico.conta_id)
        conta = session.exec(statement).first()
        if historico.tipo == Tipos.ENTRADA:
            conta.valor += historico.valor
        else:
            if conta.valor < historico.valor:
                raise ValueError("Saldo insuficiente")
            conta.valor -= historico.valor

        session.add(historico)
        session.commit()
        return historico
```

Crie a feature onde o usuário pode verificar o saldo somado de todas as contas:

```
def total_contas():
    with Session(engine) as session:
        statement = select(Conta)
        contas = session.exec(statement).all()

    total = 0
    for conta in contas:
        total += conta.valor

    return float(total)
```

▼ Análise de dados

Nessa etapa vamos desenvolver funcionalidades que permite o usuário analisar seus dados financeiros para tomar melhores decisões.

Primeiro, crie a view responsável por filtrar as movimentações financeiras dentro de um período específico:

```
def buscar_historicos_entre_datas(data_inicio: date, data_fim: date):
    with Session(engine) as session:
        statement = select(Historico).where(
            Historico.data >= data_inicio,
            Historico.data <= data_fim
        )
        resultados = session.exec(statement).all()
        return resultados
```

Para finalizar as views, vamos para etapa de criação de gráficos e para isso precisaremos instalar a biblioteca MATPLOTLIB:

```
pip install matplotlib
```

Com ela instalada, podemos desenvolver um gráfico que mostra o total de dinheiro em cada conta para o usuário:

```
def criar_grafico_por_conta():
    with Session(engine) as session:
        statement = select(Conta).where(Conta.status==Status.ATIVO)
        contas = session.exec(statement).all()
        bancos = [i.banco.value for i in contas]
        total = [i.valor for i in contas]
        import matplotlib.pyplot as plt
        plt.bar(bancos, total)
        plt.show()
```

▼ Interface (Terminal)

Crie agora uma interface via terminal para permitir que o usuário interaja com a aplicação:

```
from models import *
from view import *

class UI:
    def start(self):
        while True:
            print('''
            [1] → Criar conta
            [2] → Desativar conta
            [3] → Transferir dinheiro
            [4] → Movimentar dinheiro
            [5] → Total contas
            [6] → Filtrar histórico
            [7] → Gráfico
                ''')

            choice = int(input('Escolha uma opção: '))

            if choice == 1:
                self._criar_conta()
            elif choice == 2:
                self._desativar_conta()
            elif choice == 3:
                self._transferir_saldo()
            elif choice == 4:
                self._movimentar_dinheiro()
            elif choice == 5:
                self._total_contas()
            elif choice == 6:
                self._filtrar_movimentacoes()
            elif choice == 7:
                self._criar_grafico()
            else:
                break

    def _criar_conta(self):
        print('Digite o nome de algum dos bancos abaixo:')
        for banco in Bancos:
            print(f'---{banco.value}---')

        banco = input().title()
        valor = float(input('Digite o valor atual disponível na conta: '))

        conta = Conta(banco=Bancos(banco), valor=valor)
        criar_conta(conta)

    def _desativar_conta(self):
        print('Escolha a conta que deseja desativar.')
        for i in listar_contas():
            if i.valor == 0:
                print(f'{i.id} → {i.banco.value} → R$ {i.valor}')

        id_conta = int(input())

        try:
            desativar_conta(id_conta)
            print('Conta desativada com sucesso.')
        except ValueError:
            print('Essa conta ainda possui saldo, faça uma transferência')

    def _transferir_saldo(self):
        print('Escolha a conta retirar o dinheiro.')
        for i in listar_contas():
            print(f'{i.id} → {i.banco.value} → R$ {i.valor}')

        conta_retirar_id = int(input())

        print('Escolha a conta para enviar dinheiro.')
        for i in listar_contas():
            if i.id != conta_retirar_id:
```

```
            print(f'{i.id} → {i.banco.value} → R$ {i.valor}')

        conta_enviar_id = int(input())

        valor = float(input('Digite o valor para transferir: '))
        transferir_saldo(conta_retirar_id, conta_enviar_id, valor)

    def _movimentar_dinheiro(self):
        print('Escolha a conta.')
        for i in listar_contas():
            print(f'{i.id} → {i.banco.value} → R$ {i.valor}')

        conta_id = int(input())

        valor = float(input('Digite o valor movimentado: '))

        print('Selecione o tipo da movimentação')
        for tipo in Tipos:
            print(f'---{tipo.value}---')

        tipo = input().title()
        historico = Historico(conta_id=conta_id, tipo=Tipos(tipo), valor=valor, data=date.today())
        movimentar_dinheiro(historico)

    def _total_contas(self):
        print(f'R$ {total_contas()}')

    def _filtrar_movimentacoes(self):
        data_inicio = input('Digite a data de início: ')
        data_fim = input('Digite a data final: ')

        data_inicio = datetime.strptime(data_inicio, '%d/%m/%Y').date()
        data_fim = datetime.strptime(data_fim, '%d/%m/%Y').date()

        for i in buscar_historicos_entre_datas(data_inicio, data_fim):
            print(f'{i.valor} - {i.tipo.value}')

    def _criar_grafico(self):
        criar_grafico_por_conta()

UI().start()
```