Codice Back-End – Progetto Revolut Web App

requirements.txt

```
fastapi
uvicorn
pydantic
```

models.py

```
from pydantic import BaseModel
from typing import List
from datetime import datetime


class User(BaseModel):
    id: int
    name: str
    email: str


class Account(BaseModel):
    id: int
    user_id: int
    currency: str
    balance: float


class Transaction(BaseModel):
    id: int
    account_id: int
    type: str  # "DEBIT" o "CREDIT"
    amount: float
    currency: str
    description: str
    created_at: datetime


class AccountResponse(BaseModel):
    id: int
    currency: str
```

```
    balance: float


class TransactionResponse(BaseModel):
    id: int
    type: str
    amount: float
    currency: str
    description: str
    created_at: datetime


class TransferRequest(BaseModel):
    source_account_id: int
    target_account_id: int
    amount: float


class TransferResponse(BaseModel):
    source_account_id: int
    target_account_id: int
    amount: float
    exchange_rate: float
    credited_amount: float
    new_source_balance: float
    new_target_balance: float
```

database.py

```
from datetime import datetime
from typing import Dict, List
from .models import User, Account, Transaction

# In un sistema reale queste informazioni risiederebbero in un database.
# Qui vengono simulate in memoria per semplicità.

users: Dict[int, User] = {
    1: User(id=1, name="Mario Rossi", email="mario.rossi@example.com")
}

accounts_by_user: Dict[int, List[Account]] = {
```

```
    1: [
        Account(id=101, user_id=1, currency="EUR", balance=1000.0),
        Account(id=102, user_id=1, currency="USD", balance=500.0),
    ]
}

# Indicizziamo i conti anche per id, per accesso diretto
accounts_by_id: Dict[int, Account] = {
    acc.id: acc
    for user_accounts in accounts_by_user.values()
    for acc in user_accounts
}

transactions_by_account: Dict[int, List[Transaction]] = {
    101: [
        Transaction(
            id=1,
            account_id=101,
            type="CREDIT",
            amount=1000.0,
            currency="EUR",
            description="Deposito iniziale",
            created_at=datetime.now(),
        )
    ],
    102: [
        Transaction(
            id=2,
            account_id=102,
            type="CREDIT",
            amount=500.0,
            currency="USD",
            description="Deposito iniziale",
            created_at=datetime.now(),
        )
    ],
}
```

services.py

```
from datetime import datetime
from .models import Account, Transaction, TransferRequest, TransferResponse
from .database import accounts_by_id, transactions_by_account


class TransferService:
    @staticmethod
    def get_exchange_rate(source_currency: str, target_currency: str) -> float:
        """
        Tassi di cambio simulati.
        In un contesto reale verrebbero presi da un servizio esterno.
        """
        if source_currency == target_currency:
            return 1.0

        # Esempio semplificato: EUR <-> USD
        if source_currency == "EUR" and target_currency == "USD":
            return 1.1
        if source_currency == "USD" and target_currency == "EUR":
            return 0.9

        # Default (nessuna coppia specificata): tasso 1:1
        return 1.0

    @staticmethod
    def execute_transfer(payload: TransferRequest) -> TransferResponse:
        # Recuperiamo i conti
        source = accounts_by_id.get(payload.source_account_id)
        target = accounts_by_id.get(payload.target_account_id)

        if source is None or target is None:
            raise ValueError("Conto sorgente o destinazione inesistente")

        if payload.amount <= 0:
            raise ValueError("L'importo deve essere positivo")

        if source.balance < payload.amount:
            raise ValueError("Fondi insufficienti sul conto sorgente")
```

```
# Calcolo del tasso di cambio
rate = TransferService.get_exchange_rate(source.currency, target.currency)
credited_amount = payload.amount * rate

# Aggiornamento dei saldi
source.balance -= payload.amount
target.balance += credited_amount

# Aggiornamento delle transazioni
now = datetime.now()

debit_tx = Transaction(
    id=len(transactions_by_account.get(source.id, [])) + 1,
    account_id=source.id,
    type="DEBIT",
    amount=payload.amount,
    currency=source.currency,
    description=f"Trasferimento verso conto {target.id}",
    created_at=now,
)

credit_tx = Transaction(
    id=len(transactions_by_account.get(target.id, [])) + 1,
    account_id=target.id,
    type="CREDIT",
    amount=credited_amount,
    currency=target.currency,
    description=f"Trasferimento da conto {source.id}",
    created_at=now,
)

transactions_by_account.setdefault(source.id, []).append(debit_tx)
transactions_by_account.setdefault(target.id, []).append(credit_tx)

return TransferResponse(
    source_account_id=source.id,
    target_account_id=target.id,
    amount=payload.amount,
    exchange_rate=rate,
    credited_amount=credited_amount,
```

```
            new_source_balance=source.balance,
            new_target_balance=target.balance,
        )
```

main.py

```
from fastapi import FastAPI, HTTPException
from typing import List
from .models import (
    AccountResponse,
    TransactionResponse,
    TransferRequest,
    TransferResponse,
)
from .database import users, accounts_by_user, transactions_by_account
from .services import TransferService

app = FastAPI(
    title="Revolut-Style API",
    description="Prototipo di API per dashboard multivaluta ispirata a Revolut",
    version="1.0.0",
)


@app.get("/api/users/{user_id}/accounts", response_model=List[AccountResponse])
def get_accounts(user_id: int):
    if user_id not in users:
        raise HTTPException(status_code=404, detail="Utente non trovato")
    accounts = accounts_by_user.get(user_id, [])
    return [AccountResponse(id=a.id, currency=a.currency, balance=a.balance) for a in
accounts]


@app.get(
    "/api/accounts/{account_id}/transactions",
    response_model=List[TransactionResponse],
)
def get_transactions(account_id: int):
    if account_id not in transactions_by_account:
        raise HTTPException(status_code=404, detail="Conto non trovato")
    txs = transactions_by_account[account_id]
```

```
    return [
        TransactionResponse(
            id=t.id,
            type=t.type,
            amount=t.amount,
            currency=t.currency,
            description=t.description,
            created_at=t.created_at,
        )
        for t in txs
    ]


@app.post("/api/transfers", response_model=TransferResponse, status_code=201)
def transfer(request: TransferRequest):
    try:
        result = TransferService.execute_transfer(request)
        return result
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
```