Source Code: AI Generated Report

Generated on: 2026-02-07 14:02:16

```
from fastapi import FastAPI, Depends, HTTPException, Query, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from pydantic import BaseModel

from sqlalchemy import create_engine, Column, Integer, String, Float, ForeignKey

from sqlalchemy.orm import sessionmaker, DeclarativeBase, relationship

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import Session

from datetime import datetime, timedelta

import jwt

from fastapi.testclient import TestClient


DATABASE_URL = "sqlite:///:memory:"


engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)


class Base(DeclarativeBase):

    pass


class User(Base):

    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)

    username = Column(String, unique=True, index=True)
    hashed_password = Column(String)

    role = Column(String)


class Transaction(Base):

    __tablename__ = "transactions"

    id = Column(Integer, primary_key=True, index=True)

    amount = Column(Float)

    category = Column(String)

    description = Column(String)

    user_id = Column(Integer, ForeignKey("users.id"))

    user = relationship("User")


Base.metadata.create_all(bind=engine)


class UserCreate(BaseModel):

    username: str

    password: str

    role: str = "user"


class UserInDB(UserCreate):

    hashed_password: str


class TransactionCreate(BaseModel):

    amount: float

    category: str

    description: str
```

```
class Token(BaseModel):

    access_token: str

    token_type: str



class TokenData(BaseModel):

    username: str



app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")



SECRET_KEY = "mysecretkey"

ALGORITHM = "HS256"

ACCESS_TOKEN_EXPIRE_MINUTES = 30



def create_access_token(data: dict, expires_delta: timedelta = None):

    to_encode = data.copy()

    if expires_delta:

        expire = datetime.utcnow() + expires_delta

    else:

        expire = datetime.utcnow() + timedelta(minutes=15)

    to_encode.update({"exp": expire})

    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

    return encoded_jwt



def get_user(db: Session, username: str):
    return db.query(User).filter(User.username == username).first()


def verify_password(plain_password, hashed_password):
    return plain_password == hashed_password


def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
    except jwt.PyJWTError:
        raise credentials_exception
    return token_data


@app.post("/token", response_model=Token)
def login(form_data: OAuth2PasswordRequestForm = Depends()):
    db = SessionLocal()
    user = get_user(db, form_data.username)
    if not user or not verify_password(form_data.password, user.hashed_password):
```

```python
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}


@app.post("/users/", response_model=UserInDB)
def create_user(user: UserCreate):
    db = SessionLocal()
    hashed_password = user.password  # In a real app, hash the password
    db_user = User(username=user.username, hashed_password=hashed_password,
role=user.role)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user


@app.post("/transactions/", response_model=TransactionCreate)
def create_transaction(transaction: TransactionCreate, current_user: TokenData =
Depends(get_current_user)):
    db = SessionLocal()
    db_transaction = Transaction(**transaction.dict(), user_id=current_user.username)
    db.add(db_transaction)
    db.commit()
    db.refresh(db_transaction)
    return db_transaction


@app.get("/transactions/", response_model=list[TransactionCreate])
def read_transactions(skip: int = 0, limit: int = 10, current_user: TokenData =
Depends(get_current_user)):
    db = SessionLocal()
            transactions = db.query(Transaction).filter(Transaction.user_id ==
current_user.username).offset(skip).limit(limit).all()
    return transactions


client = TestClient(app)


def test_create_user():
        response = client.post("/users/", json={"username": "testuser", "password":
"testpass", "role": "user"})
    if response.status_code != 200:
        print(response.status_code, response.text)
    assert response.status_code == 200


def test_login():
        response = client.post("/token", data={"username": "testuser", "password":
"testpass"})
```

```
    if response.status_code != 200:
        print(response.status_code, response.text)
    assert response.status_code == 200


def test_create_transaction():
    token_response = client.post("/token", data={"username": "testuser", "password": "testpass"})
    token = token_response.json().get("access_token")
    response = client.post("/transactions/", json={"amount": 100.0, "category": "Food", "description": "Lunch"}, headers={"Authorization": f"Bearer {token}"})
    if response.status_code != 200:
        print(response.status_code, response.text)
    assert response.status_code == 200


test_create_user()
test_login()
test_create_transaction()
```