Atividade Prática: Backend de um Sistema de Gerenciamento de Tarefas Simples (Task Manager)

Objetivo:

Desenvolver o backend de um sistema simples de gerenciamento de tarefas (Task Manager) utilizando Python com FastAPI, ORM (SQLAlchemy), Postgres (Docker) e a partir de uma solicitação de cliente fictícia.

Duração: 4 horas (foco no backend)

Solicitação do Cliente (Texto Base):

"Prezados,

Necessitamos de um sistema simples para gerenciar as tarefas da nossa equipe. Atualmente, utilizamos planilhas e e-mails, o que se tornou ineficiente. O sistema deve permitir que um **usuário** (identificado por um nome e e-mail únicos) possa criar, atribuir, atualizar e deletar **tarefas**. Cada tarefa deve ter um título, descrição, data de vencimento e status (pendente, em andamento, concluída).

O sistema deve garantir que cada tarefa seja atribuída a um único usuário. Além disso, gostaríamos de ter a possibilidade de filtrar as tarefas por usuário, status e data de vencimento.

É fundamental que o sistema seja rápido e responsivo, mesmo com um grande número de tarefas. A segurança dos dados é crucial, e o acesso deve ser restrito a usuários autenticados. Queremos também que o sistema seja fácil de manter e escalar no futuro.

Agradecemos a atenção e aguardamos ansiosamente a entrega do sistema."

Etapas da Atividade:

1. **Análise da Solicitação (30 minutos):**
    - **Dividir os alunos em grupos.**
    - Cada grupo deve ler atentamente a solicitação do cliente.
    - Cada grupo deve identificar e documentar:
        - **Casos de Uso:** (Ex: Criar Tarefa, Atribuir Tarefa, Atualizar Tarefa, Listar Tarefas, Deletar Tarefa)
        - **Requisitos Funcionais:** (Ex: Usuários podem criar tarefas com título, descrição, data de vencimento e status; Tarefas podem ser filtradas por usuário, status e data de vencimento)
        - **Requisitos Não-Funcionais:** (Ex: Rápido e responsivo, seguro, fácil de manter e escalar)
        - **Atores:** (Usuário)
        - **Regras de Negócio:** (Ex: Cada tarefa deve ser atribuída a um único usuário; E-mails de usuários devem ser únicos.)
    - Cada grupo compartilha suas descobertas com a turma. Discutir e consolidar as informações.
2. **Configuração do Ambiente (30 minutos):**
    - **Docker e Postgres:**
        - Fornecer um arquivo `docker-compose.yml` pré-configurado com o Postgres.
        - Orientar os alunos a executar o comando `docker-compose up -d` para iniciar o container.
    - **Projeto Python (FastAPI):**
        - Criar um novo projeto Python.
        - Instalar as dependências: `pip install fastapi uvicorn sqlalchemy psycopg2-binary python-dotenv`
        - Criar o arquivo `.env` com as configurações do banco de dados (ex: `DATABASE_URL=postgresql://user:password@localhost:5432/database`).
3. **Modelagem do Banco de Dados (30 minutos):**
    - Definir os modelos do banco de dados usando SQLAlchemy.
    - Criar as tabelas `users` e `tasks`.
    - Exemplo (simplificado):

```
 from sqlalchemy import create_engine, Column, Integer, String, DateTime, Enum, ForeignKey
from sqlalchemy.orm import declarative_base, sessionmaker, relationship
from datetime import datetime
import enum

Base = declarative_base()

class StatusEnum(str, enum.Enum):
    PENDENTE = "pendente"
    EM_ANDAMENTO = "em_andamento"
    CONCLUIDA = "concluida"

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    email = Column(String, unique=True, nullable=False)
    tasks = relationship("Task", back_populates="owner")

class Task(Base):
    __tablename__ = "tasks"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, nullable=False)
    description = Column(String)
    due_date = Column(DateTime)
    status = Column(Enum(StatusEnum), default=StatusEnum.PENDENTE)
    owner_id = Column(Integer, ForeignKey("users.id"))
    owner = relationship("User", back_populates="tasks")


# Configuração do banco de dados (substitua com suas configurações)
DATABASE_URL = "postgresql://user:password@localhost:5432/database"
engine = create_engine(DATABASE_URL)
Base.metadata.create_all(engine)

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

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

4. **Implementação das Rotas da API (2 horas):**
    - Criar as rotas para:
        - **Criar Usuário (POST /users):** Validação dos dados (e-mail único).
        - **Criar Tarefa (POST /tasks):** Atribuição da tarefa a um usuário existente.
        - **Listar Tarefas (GET /tasks):** Filtragem por usuário, status e data de vencimento (opcional).
        - **Atualizar Tarefa (PUT /tasks/{task_id}):** Alterar título, descrição, data de vencimento e status.
        - **Deletar Tarefa (DELETE /tasks/{task_id}).**
    - Utilizar o `SessionLocal` para criar uma sessão do banco de dados em cada rota.
    - Exemplo (simplificado):

```
 from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from . import models, schemas
from .database import get_db

app = FastAPI()

@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = db.query(models.User).filter(models.User.email == user.email).first()
    if db_user:
        raise HTTPException(status_code=400, detail="Email already registered")
    db_user = models.User(**user.dict())
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user

@app.post("/tasks/", response_model=schemas.Task)
def create_task(task: schemas.TaskCreate, db: Session = Depends(get_db)):
    db_task = models.Task(**task.dict())
    db.add(db_task)
    db.commit()
    db.refresh(db_task)
    return db_task

@app.get("/tasks/", response_model=list[schemas.Task])
def read_tasks(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    tasks = db.query(models.Task).offset(skip).limit(limit).all()
    return tasks
```

5. **Testes e Ajustes (30 minutos):**
    - Testar as rotas da API utilizando ferramentas como `curl`, `Postman` ou a interface SwaggerUI do FastAPI.
    - Realizar ajustes e correções de erros.

Recursos Adicionais:

- **Documentação FastAPI:** https://fastapi.tiangolo.com/
- **Documentação SQLAlchemy:** https://www.sqlalchemy.org/
- **Documentação Docker:** https://docs.docker.com/

Avaliação:

A avaliação deve considerar:

- A extração correta dos casos de uso, requisitos, atores e regras de negócio.
- A modelagem adequada do banco de dados.
- A implementação correta das rotas da API.
- A qualidade do código (legibilidade, organização, tratamento de erros).
- O funcionamento geral do sistema.

Observações:

- Esta atividade é um guia. Adapte-a de acordo com o nível de seus alunos e o tempo disponível.
- Incentive os alunos a pesquisar e explorar as bibliotecas e frameworks utilizados.
- Priorize a compreensão dos conceitos sobre a implementação perfeita.
- Forneça exemplos de código e templates para auxiliar os alunos.
- Encoraje a colaboração e o compartilhamento de conhecimento entre os alunos.

Schemas:

Não se esqueça de criar os schemas para validação dos dados:

```
 from typing import Optional

from pydantic import BaseModel


class UserBase(BaseModel):
    name: str
    email: str


class UserCreate(UserBase):
    pass


class User(UserBase):
    id: int

    class Config:
        orm_mode = True


class TaskBase(BaseModel):
    title: str
    description: Optional[str] = None


class TaskCreate(TaskBase):
    pass


class Task(TaskBase):
    id: int
    owner_id: int

    class Config:
        orm_mode = True
```

Esta atividade prática ajudará seus alunos a consolidar o conhecimento sobre o processo de desenvolvimento de software, desde a análise da solicitação do cliente até a implementação do backend utilizando tecnologias modernas e relevantes para o mercado de trabalho. Boa sorte!