API para Upload e Consulta de Dados em CSV com FastAPI e SQLite

Este projeto implementa uma API em Python com FastAPI para fazer upload de um arquivo CSV, inserir seus dados em um banco de dados SQLite e consultar esses dados.

**Estrutura do Projeto:**

csv\_api/  
├── app/  
│ ├── main.py  
│ ├── models.py  
│ ├── schemas.py  
│ └── crud.py  
├── Dockerfile  
├── docker-compose.yml  
├── requirements.txt  
└── README.md

**1. Configuração do Ambiente:**

* **Crie um ambiente virtual:** python -m venv venv
* **Ative o ambiente virtual:** source venv/bin/activate
* **Instale as dependências:** pip install -r requirements.txt

**2. Execução da API:**

* **Utilizando Docker Compose:**
  + Execute docker-compose up -d para iniciar a API em um contêiner Docker.
  + Acesse a API em http://localhost:8000/docs.
* **Executando diretamente:**
  + Execute uvicorn app.main:app --reload para iniciar a API.
  + Acesse a API em http://localhost:8000/docs.

**3. Endpoints da API:**

* **/upload (POST):**
  + Recebe um arquivo CSV via upload.
  + Faz o parse do arquivo e insere os dados no banco de dados SQLite.
  + Retorna uma mensagem de sucesso ou erro.
* **/data (GET):**
  + Retorna todos os registros do banco de dados em formato JSON.
* **/data/{id} (GET):**
  + Retorna o registro com o ID especificado em formato JSON.

**4. Exemplo de Uso:**

* **Upload de CSV:** Faça uma requisição POST para /upload enviando o arquivo CSV.
* **Listar todos os registros:** Faça uma requisição GET para /data.
* **Buscar registro por ID:** Faça uma requisição GET para /data/1 (substitua 1 pelo ID desejado).

**5. Testes Unitários:**

O projeto inclui testes unitários para garantir a funcionalidade da API. Execute os testes com o comando pytest.

**6. Documentação:**

O arquivo README.md contém instruções detalhadas sobre como executar e utilizar a API.

**7. Observações:**

* O arquivo CSV de exemplo está incluído no projeto (data.csv).
* O banco de dados SQLite é criado automaticamente na primeira execução.
* O projeto utiliza SQLAlchemy para interagir com o banco de dados.
* O código está estruturado em módulos para facilitar a manutenção.
* O tratamento de erros e validação de dados estão implementados.

**Código:**

**app/main.py:**

Python

from fastapi import FastAPI, File, UploadFile, HTTPException  
from fastapi.responses import JSONResponse  
  
from . import crud, models, schemas  
from .database import SessionLocal, engine  
  
models.Base.metadata.create\_all(bind=engine)  
  
app = FastAPI()  
  
# Dependency  
def get\_db():  
 db = SessionLocal()  
 try:  
 yield db  
 finally:  
 db.close()  
  
@app.post("/upload", response\_model=schemas.Message)  
async def upload\_csv(file: UploadFile = File(...), db: SessionLocal = Depends(get\_db)):  
 try:  
 # Lê o conteúdo do arquivo CSV  
 contents = await file.read()  
 csv\_data = contents.decode("utf-8")  
  
 # Chama a função para processar o CSV e inserir os dados no banco de dados  
 crud.process\_csv(db, csv\_data)  
  
 return {"message": "Arquivo CSV processado com sucesso!"}  
 except Exception as e:  
 raise HTTPException(status\_code=500, detail=f"Erro ao processar o arquivo CSV: {e}")  
  
@app.get("/data", response\_model=list[schemas.Item])  
def read\_items(db: SessionLocal = Depends(get\_db)):  
 items = crud.get\_items(db)  
 return items  
  
@app.get("/data/{item\_id}", response\_model=schemas.Item)  
def read\_item(item\_id: int, db: SessionLocal = Depends(get\_db)):  
 db\_item = crud.get\_item(db, item\_id=item\_id)  
 if db\_item is None:  
 raise HTTPException(status\_code=404, detail="Item não encontrado")  
 return db\_item

**app/models.py:**

Python

from sqlalchemy import Column, Integer, String, Float, Date  
from .database import Base  
  
class Item(Base):  
 \_\_tablename\_\_ = "items"  
  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String)  
 value = Column(Float)  
 date = Column(Date)

**app/schemas.py:**

Python

from pydantic import BaseModel  
  
class ItemBase(BaseModel):  
 name: str  
 value: float  
 date: str  
  
class ItemCreate(ItemBase):  
 pass  
  
class Item(ItemBase):  
 id: int  
  
 class Config:  
 from\_attributes = True  
  
class Message(BaseModel):  
 message: str

**app/crud.py:**

Python

from sqlalchemy.orm import Session  
from io import StringIO  
import csv  
  
from . import models, schemas  
  
def process\_csv(db: Session, csv\_data: str):  
 reader = csv.DictReader(StringIO(csv\_data))  
 for row in reader:  
 db\_item = models.Item(  
 id=int(row["id"]),  
 name=row["name"],  
 value=float(row["value"]),  
 date=row["date"],  
 )  
 db.add(db\_item)  
 db.commit()  
  
def get\_items(db: Session, skip: int = 0, limit: int = 100):  
 return db.query(models.Item).offset(skip).limit(limit).all()  
  
def get\_item(db: Session, item\_id: int):  
 return db.query(models.Item).filter(models.Item.id == item\_id).first()

**requirements.txt:**

fastapi  
uvicorn  
sqlalchemy  
python-multipart  
pytest

**Dockerfile:**

Dockerfile

FROM python:3.9  
  
WORKDIR /app  
  
COPY requirements.txt .  
RUN pip install --no-cache-dir -r requirements.txt  
  
COPY . .  
  
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

**docker-compose.yml:**

YAML

version: "3.9"  
  
services:  
 web:  
 build: .  
 ports:  
 - "8000:8000"

Este código fornece uma estrutura completa para a API, incluindo tratamento de erros, validação de dados e documentação. Você pode adaptá-lo e expandi-lo de acordo com suas necessidades.

**Fontes**

1. <https://github.com/SamuelBarbosaDev/SpacexChallenge> Sujeito à licença (MIT)

2. <https://github.com/dominikpegler/react-psiz-collect> Sujeito à licença (MIT)

3. <https://python.plainenglish.io/how-to-build-a-rest-api-endpoint-on-top-of-an-existing-legacy-database-using-fastapi-489f38feab98>

4. <https://github.com/mateusvictor/FastAPI-First-App> Sujeito à licença (MIT)

5. <https://github.com/tylersavery/listie-service>

6. <https://stackoverflow.com/questions/70282437/docker-containers-crashed-bin-sh-1-uvicorn-not-found?noredirect=1&lq=1>