Requirement Document for a Social Media App

by

Adam Jardali-202101194

Ihab Awad-202101195

Sara Ezzedine-202000684

A REPORT

Submitted to Dr. Ibrahim Al Bitar in partial fulfillment of the requirements for the course Software Engineering

Phase 3

April 28th,2023

Table of Contents

This page is intentionally left blank.

Introduction

Our social media app is designed to provide a seamless user experience for social media enthusiasts. Using the fastapi framework and PostgreSQL for deployment, along with SQLite for testing, we aim to create a fast and reliable platform that users can rely on.

To ensure maximum security and privacy for our users, we have implemented JWT (Json Web Tokens) for authentication and authorization. This will enable users to securely login, create profiles, and access the features of the app.

Initially, we faced some challenges related to database management and performance, which prompted us to shift from our original database solution to PostgreSQL. With PostgreSQL, we were able to improve the overall speed and reliability of the app.

In this document, we will showcase three use cases of our software, including user profile creation and sharing posts, and likes viewing. With these features, our app will provide a comprehensive social media experience for users, making it the go-to platform for all their social needs.

FastAPI

We decided to use FastAPI because: it is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints.

The key features are:

- **Fast**: Very high performance, on par with **NodeJS** and **Go** (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- **Fast to code**: Increase the speed to develop features by about 200% to 300%. *
- **Fewer bugs**: Reduce about 40% of human (developer) induced errors. *
- **Intuitive**: Great editor support. Completion everywhere. Less time debugging.
- **Easy**: Designed to be easy to use and learn. Less time reading docs.
- **Short**: Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- **Robust**: Get production-ready code. With automatic interactive documentation.

Standards-based: Based on (and fully compatible with) the open standards for APIs: OpenAPI (previously known as Swagger) and JSON Schema.

In this project we used the best practices and design patterns of FastAPI although it would take more time to implement, the code structure will get more complex, but it is efficient for scalability as the project will get bigger.

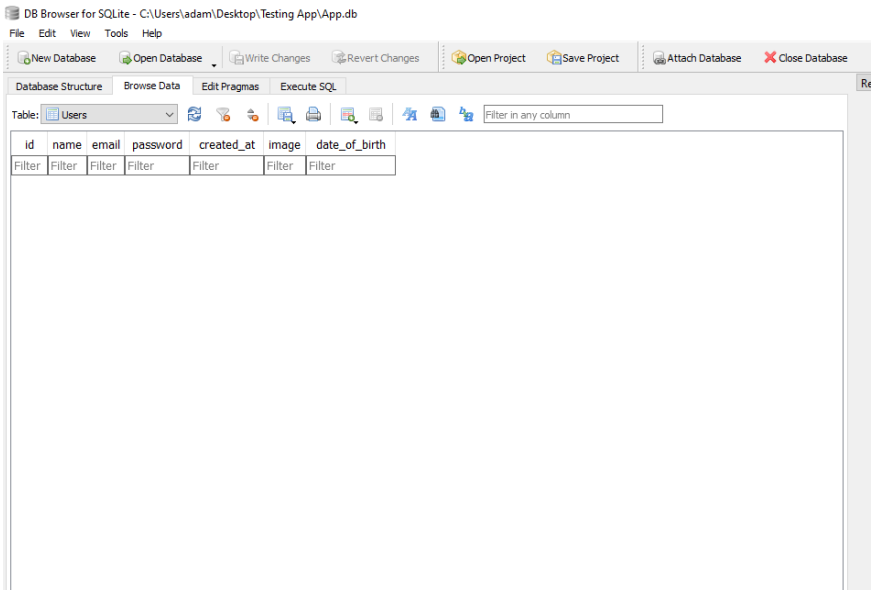
Database

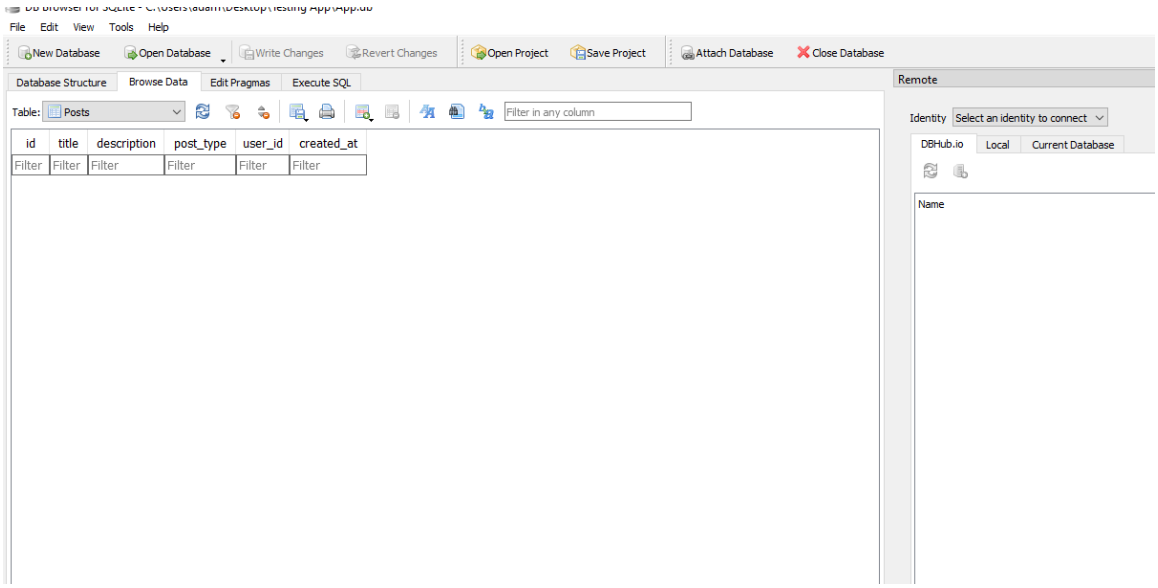
For the database we decided to use PostgreSQL for deployment and SQLite for testing. In this picture we are setting the database. As you can see, we are using Environment variables so would keep the database info secret as we publish the project.

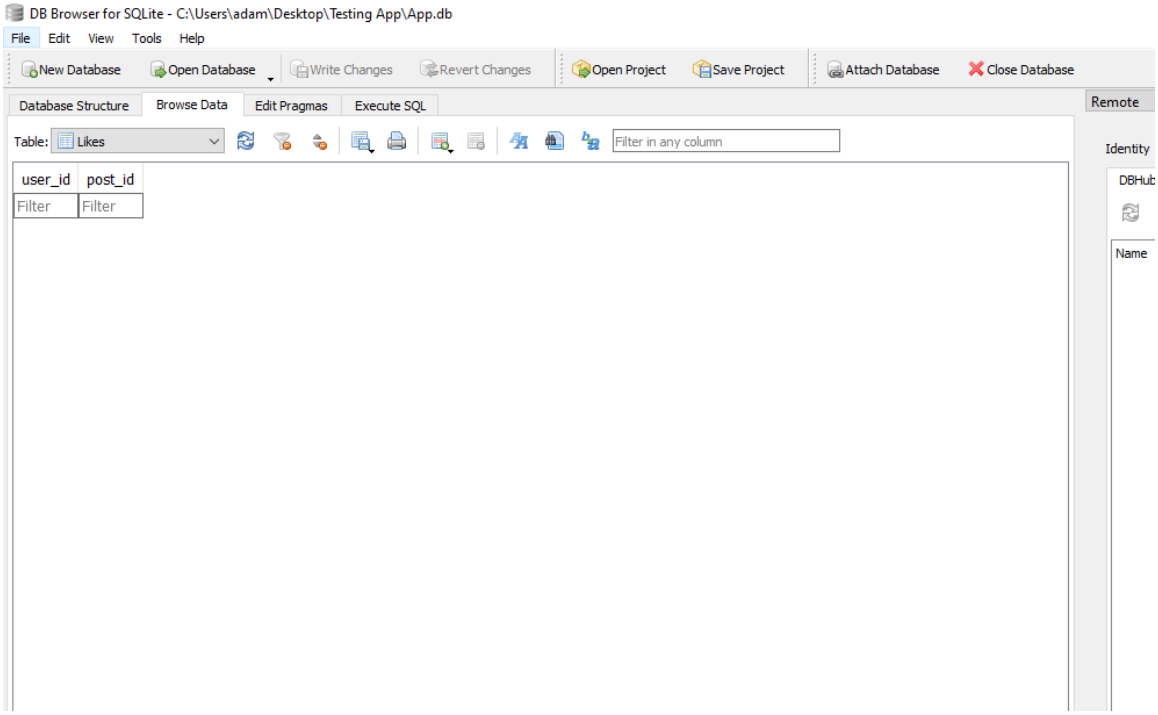
```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
import os
from dotenv import load_dotenv
load_dotenv()

SQLALCHAMY_DATABASE_URL = os.getenv("DB_URL")
engine = create_engine(SQLALCHAMY_DATABASE_URL, connect_args={
                                "check_same_thread": False})

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

Base = declarative_base()

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

We decided to create 3 entities for this demo which are: User, Post, and Like.

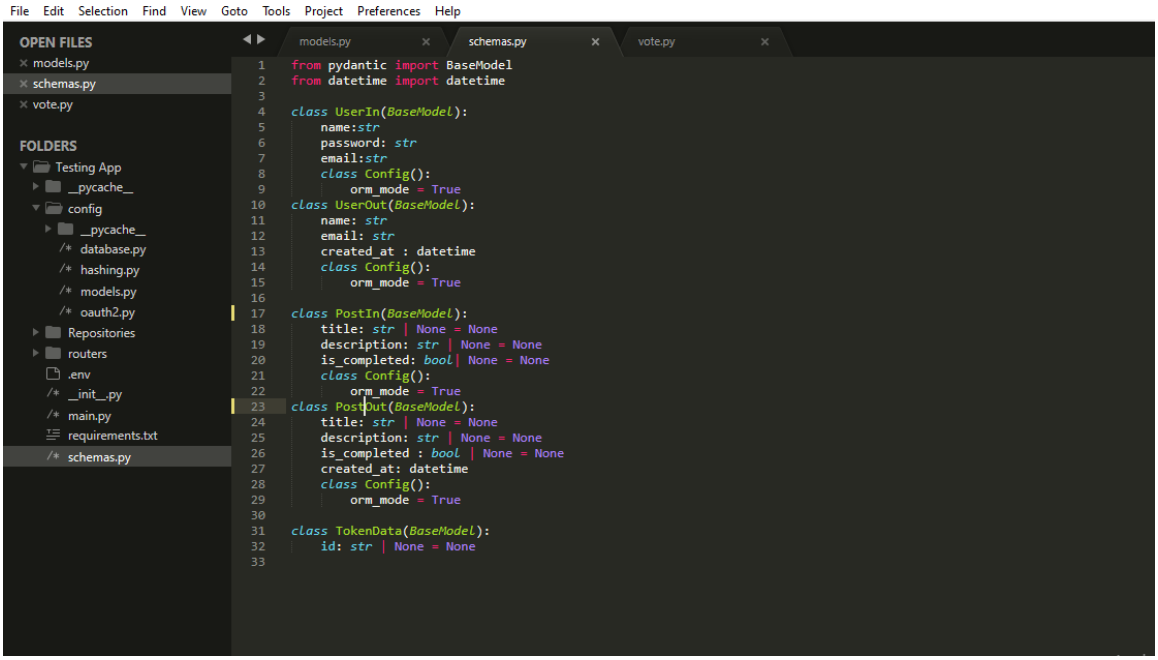
```python
from config.database import Base
from sqlalchemy import Column, Integer, DateTime, String, ForeignKey, Boolean
from sqlalchemy.orm import relationship
from datetime import datetime
var_length = 100




class User(Base):
    __tablename__ = 'Users'
    id = Column(Integer,primary_key = True, index = True)
    name = Column(String(var_length),nullable = False)
    email = Column(String(var_length), nullable = False, unique = True)
    password = Column(String(var_length), nullable = False)
    created_at = Column(DateTime, nullable  = False, default = datetime.utcnow)
    image = Column(String(var_length),default = "new_user.png")
    date_of_birth = Column(DateTime)
    posts = relationship('Pask', back_populates = 'creator')

class Post(Base):
    __tablename__ = 'Posts'
    id = Column(Integer,primary_key = True, index = True)
    title = Column(String(var_length),nullable = False, default = 'No title')
    description = Column(String(var_length),nullable = False, default = 'No description')
    post_type = Column(String, nullable = False, default = "Feelings")
    user_id = Column(Integer,ForeignKey('Users.id',ondelete = "CASCADE"))
    creator = relationship('User',back_populates = 'posts')
    created_at = Column(DateTime, nullable  = False, default = datetime.utcnow)

class Like(Base):
    __tablename__ = "Likes"

    user_id = Column(Integer, ForeignKey(
        "Users.id", ondelete="CASCADE"), primary_key=True)
    post_id = Column(Integer, ForeignKey(
        "Posts.id", ondelete="CASCADE"), primary_key=True)
```


DB Browser for SQLite - C:\Users\adam\Desktop\Testing App\App.db
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Close Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: Users
Filter in any column
id name email password created_at image date_of_birth
Filter Filter Filter Filter Filter Filter Filter


File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Close Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: Posts
Filter in any column
id title description post_type user_id created_at
Filter Filter Filter Filter Filter Filter
Remote
Identity
Select an identity to connect
DBHub.io
Local
Current Database
Name

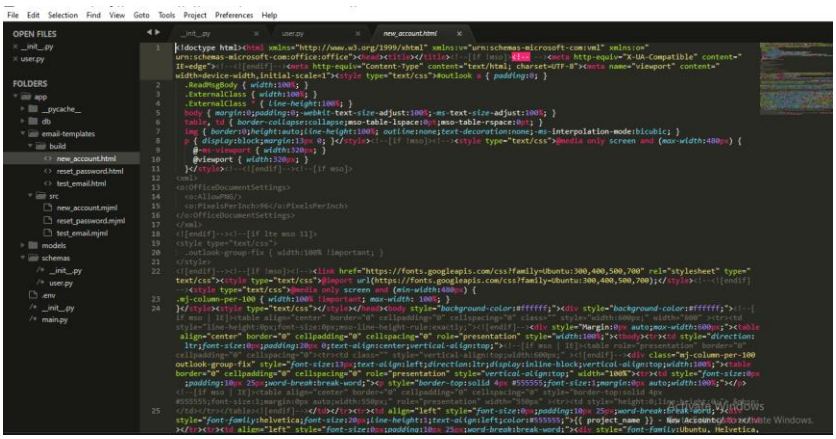


Add schemas folder which is responsible for showing the properties of each entity in different situations. For example, when creating a user, the client will see different properties of his account when he decided to make a post.



Implement the functionality to be able to send the users emails for signing up, password recovery, and changing password by using the SMTP protocol.

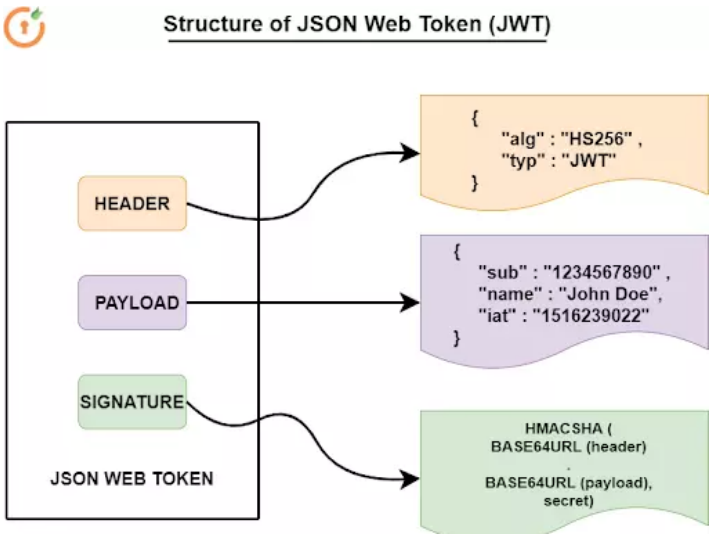
```python
    SMTP_TLS: bool = True
    SMTP_PORT: Optional[int] = 20
    SMTP_HOST: Optional[str] = 21
    SMTP_USER: Optional[str] = 22
    SMTP_PASSWORD: Optional[str] = 123456
    EMAILS_FROM_EMAIL: Optional[EmailStr] = "you@hotmail.com"
    EMAILS_FROM_NAME: Optional[str] = "XYZ"

    @validator("EMAILS_FROM_NAME")
    def get_project_name(cls, v: Optional[str], values: Dict[str, Any]) -> str:
        # # if not v:
        # #     return values["PROJECT_NAME"]
        # return v
        return "APP"
    EMAIL_RESET_TOKEN_EXPIRE_HOURS: int = 48
    EMAIL_TEMPLATES_DIR: str = "email-templates/build"
    EMAILS_ENABLED: bool = False

    @validator("EMAILS_ENABLED", pre=True)
    def get_emails_enabled(cls, v: bool, values: Dict[str, Any]) -> bool:
        return bool(
            values.get("SMTP_HOST")
            and values.get("SMTP_PORT")
            and values.get("EMAILS_FROM_EMAIL")
        )

    EMAIL_TEST_USER: EmailStr = "test@example.com"  # type: ignore
    FIRST_SUPERUSER: EmailStr = "adamjardali@hotmail.com"
    FIRST_SUPERUSER_PASSWORD: str = "12345"
    USERS_OPEN_REGISTRATION: bool = False

    class Config:
        case_sensitive = True


settings = Settings()
```

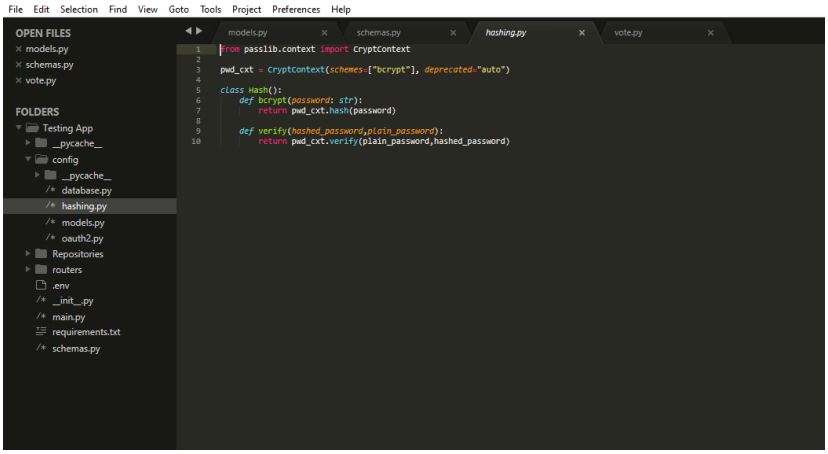


Authentication

Authentication and resource Authorization: we will use JWT (Json Web Tokens) for authentication and resource authorization. There are many steps needed to implement it.

a. Implement hashing algorithms as in the best practices, the stored password in the database should be hashed.
b. Use oauth2 methods. Create methods for creating a token and verifying it.
c. Apply functional dependency





OPEN FILES
- models.py
- schemas.py
- vote.py

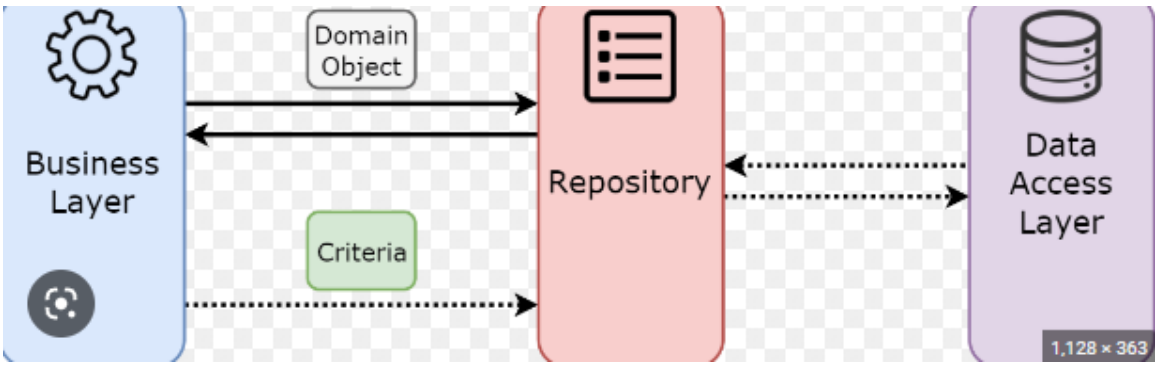
FOLDERS
- Testing App
  - __pycache__
  - config
    - __pycache__
    - database.py
    - hashing.py
    - models.py
    - oauth2.py
  - Repositories
  - routers
  - .env
  - __init__.py
  - main.py
  - requirements.txt
  - schemas.py

models.py | schemas.py | oauth2.py | vote.py

```python
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from datetime import datetime, timedelta
from config import database,models
from sqlalchemy.orm import Session
import schemas
import os
from dotenv import load_dotenv
load_dotenv()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

SECRET_KEY = os.getenv('SECRET_KEY')
ALGORITHM = os.getenv('ALGORITHM')
ACCESS_TOKEN_EXPIRE_MINUTES = os.getenv('ACCESS_TOKEN_EXPIRE_MINUTES')
ACCESS_TOKEN_EXPIRE_MINUTES = 30

get_db = database.get_db

def create_access_token(data: dict):

    to_encode = data.copy()
    expire_time = datetime.utcnow() + timedelta(minutes = int(ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({'exp':expire_time})
    token  = jwt.encode(to_encode, SECRET_KEY, algorithm = ALGORITHM)
    return token

def verify_access_token(token: str, credentials_exception):

    try:

        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        id: str = payload.get("user_id")
        if id is None:
            raise credentials_exception
        token_data = schemas.TokenData(id=id)
    except JWTError:
        raise credentials_exception

    return token_data


def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
    credentials_exception = HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
                                          detail=f"Could not validate credentials", headers={"WWW-Authenticate": "Bearer"})

    token = verify_access_token(token, credentials_exception)
```

Respiratory Pattern

Use the repository pattern for implementing each entity functionalities.



models.py | schemas.py | User.py | vote.py

```python
from fastapi import  Depends, HTTPException, status, Response
from config import models,oauth2, database
import schemas
from sqlalchemy.orm import Session
from config.hashing import Hash

class User:
    def register_user(user:schemas.UserIn, db:Session):
        hashed_password = Hash.bcrypt(user.password)
        user.password = hashed_password
        is_unique_email = db.query(models.User).filter(models.User.email == user.email)
        if is_unique_email.first():
            raise HTTPException(status_code =  409, detail = f"Email {user.email} is taken by another user")

        new_user = models.User(**user.dict())
        db.add(new_user)
        db.commit()
        db.refresh(new_user)
        return new_user

    def get_user_by_id(id: int, db: Session):
        user = db.query(models.User).filter(models.User.id == id).first()
        if not user:
            raise HTTPException(status_code = 404, detail =  f"User with id {id} is not found")
        return user

    def delete_user(id:int, db:Session, current_user : int):
        user = db.query(models.User).filter(models.User.id == id)
        if not user.first():
            raise HTTPException(status_code = 404, detail = f"User with id {id} is not found")
        if user.first().id != current_user.id:
            raise HTTPException(status_code = 403, detail = f"Not authorized to perform requested action")
        user.delete(synchronize_session = False)
        db.commit()
        return Response(status_code = 204)

    def update_user(id: int, new_user: schemas.UserIn,db:Session, current_user : int):
        user = db.query(models.User).filter(models.User.id == id)
        if not user.first():
            raise HTTPException(status_code = 404, detail = f"User with id {id} is not found")
        if user.first().id != current_user.id:
            raise HTTPException(status_code = 403, detail = f"Not authorized to perform requested action")
        user.update(new_user.dict(),synchronize_session = False)
        db.commit()
        return user.first()
```

Routers

Implement the routers of each entity.

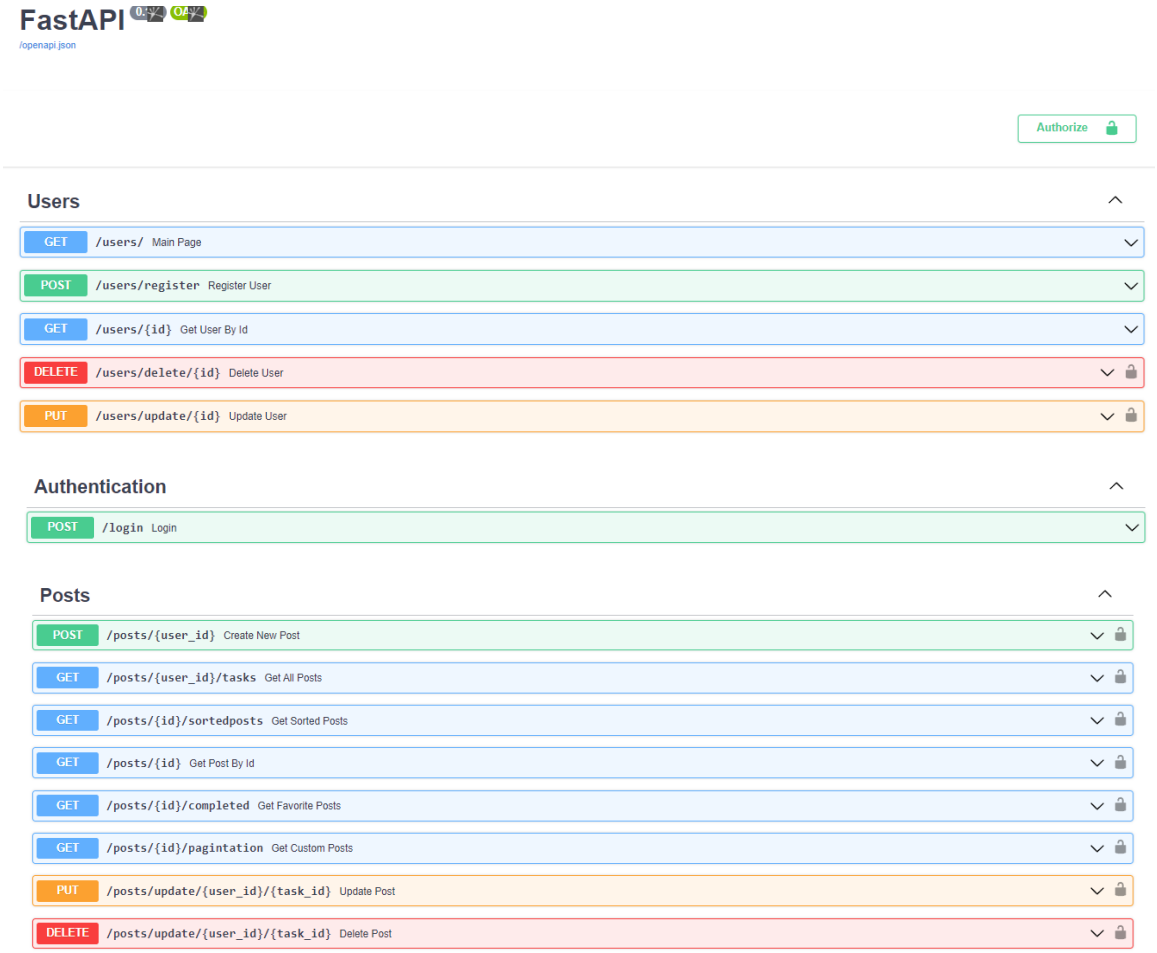
```python
from fastapi import APIRouter, Depends, HTTPException, status, Response, Request
from config import models,oauth2, database
from sqlalchemy.orm import Session

import schemas
from config.hashing import Hash
from Repositories.User import User

router = APIRouter(
    prefix = '/users',
    tags = ['Users'],
    )
get_db = database.get_db

@router.get('/')
def main_page():
    return {'detail':'This is the main page'}

@router.post('/register', response_model = schemas.UserOut,status_code= 201)
def register_user(user:schemas.UserIn, db:Session = Depends(get_db)):
    return User.register_user(user,db)

@router.get('/{id}', response_model = schemas.UserOut)
def get_user_by_id(id: int, db: Session = Depends(get_db),):
    return User.get_user_by_id(id,db)

@router.delete("/delete/{id}")
def delete_user(id:int, db:Session = Depends(get_db), current_user : int = Depends(oauth2.get_current_user)):
    return User.delete_user(id,db,current_user)

@router.put('/update/{id}', response_model = schemas.UserOut)
def update_user(id: int, new_user: schemas.UserIn,db:Session = Depends(get_db), current_user : int = Depends(oauth2.get_current_user)):
    return User.update_user(id,new_user,db,current_user)
```

```python
from fastapi import APIRouter, Depends, status, HTTPException, Response
from fastapi.security.oauth2 import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from config import database , models, hashing, oauth2

get_db = database.get_db

router = APIRouter(
        tags = ['Authentication']
    )

@router.post('/login')
def login(user_credentials : OAuth2PasswordRequestForm = Depends(), db :Session = Depends(get_db)):
    user = db.query(models.User).filter(user_credentials.username == models.User.email).first()

    if not user:
        raise HTTPException(status_code = 403, detail = "Invalid credentials")

    if  not hashing.Hash.verify(user.password,user_credentials.password):
        raise HTTPException(status_code = 403, detail = "Invalid credentials")

    access_token = oauth2.create_access_token({'user_id':user.id})
    return {'access_token':access_token, 'token_type':'bearer'}
```

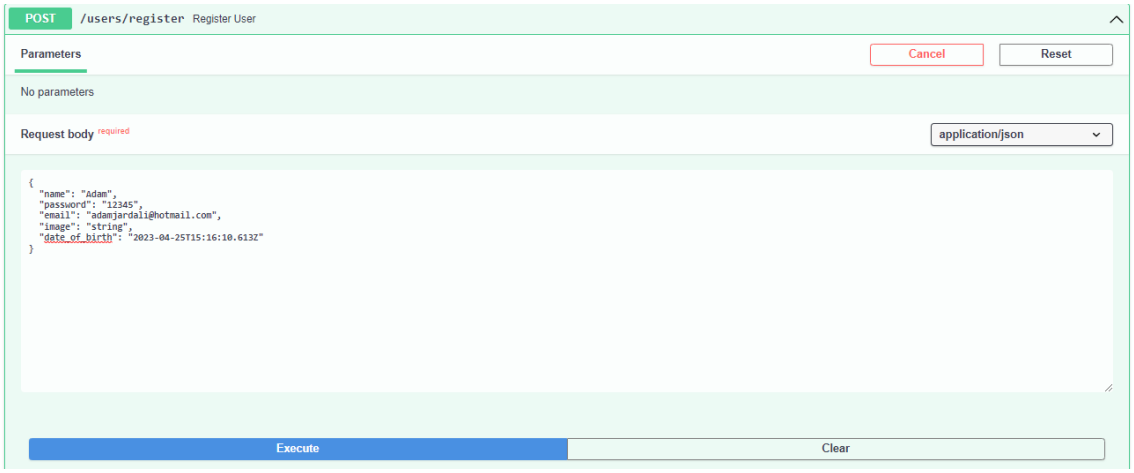
```python
from sqlalchemy.orm import Session
from typing import List
from Repositories.Task import  Task

router = APIRouter(
    prefix = '/posts',
    tags = ['Posts']
    )

get_db = database.get_db

@router.post('/{user_id}',response_model = schemas.TaskOut)
def create_new_post(user_id:int,task: schemas.TaskIn, db:Session = Depends(get_db), current_user : int = Depends(oauth2.get_current_user)):
    return Task.create_new_task(user_id,task,db,current_user)


@router.get('/{user_id}/tasks', response_model = List[schemas.TaskOut])
def get_all_posts(user_id:int,db:Session = Depends(get_db),current_user : int = Depends(oauth2.get_current_user)):
    return Task.get_all_tasks(user_id,db,current_user)

@router.get('/{id}/sortedposts', response_model = List[schemas.TaskOut])
def get_sorted_posts(id:int,db:Session = Depends(get_db), current_user : int = Depends(oauth2.get_current_user),):
    return Task.get_sorted_tasks(id,db,current_user)

@router.get('/{id}', response_model = schemas.TaskOut)
def get_post_by_id(id:int, db:Session = Depends(get_db), current_user : int = Depends(oauth2.get_current_user)):
    return Task.get_task_by_id(id,db,current_user)

@router.get('/{id}/completed', response_model = List[schemas.TaskOut])
def get_favorite_posts(id:int, db:Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
    return Task.get_completed_tasks(id,db,current_user)

@router.get('/{id}/pagintation', response_model = List[schemas.TaskOut])
def get_custom_posts(id : int,skip: int = 0, limit : int = 10, db:Session = Depends(get_db), current_user : int = Depends(oauth2.get_current_user)):
    return Task.get_custom_tasks(id,skip,limit,db,current_user)

@router.put('/update/{user_id}/{task_id}', response_model = schemas.TaskOut)
def update_post(user_id:int, task_id:int, new_task: schemas.TaskIn,db:Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
    return Task.update_task(user_id,task_id,db,current_user)


@router.delete('/update/{user_id}/{task_id}')
```

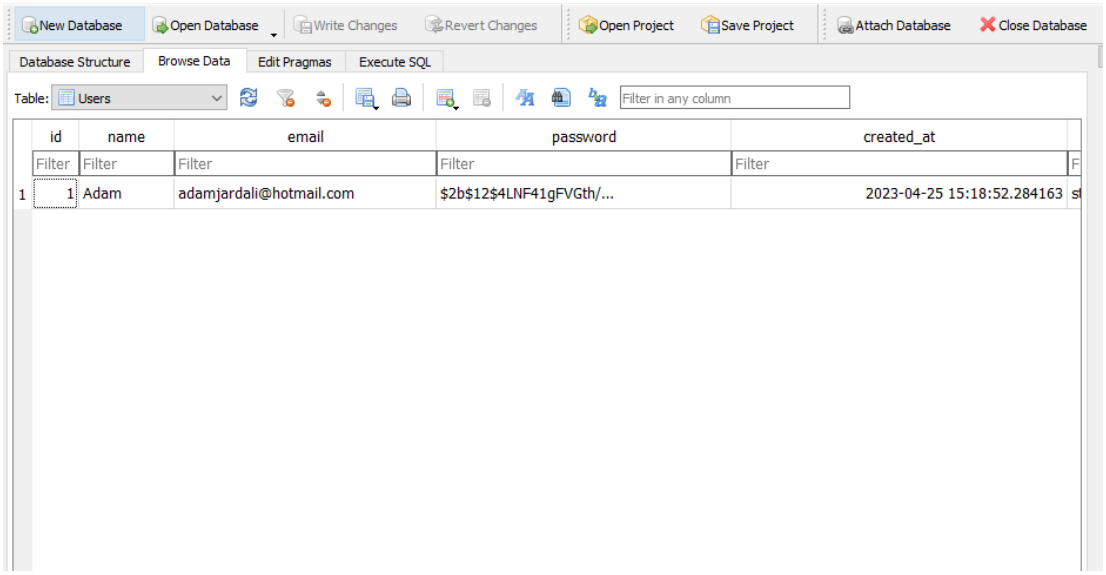
Testing

Now it is time for testing. No need to use postman as FastAPI provides a UI for that purpose.



After testing creating a new user with different scenarios everything worked fine.





Sign in with new user and try wrong inputs to see if it works.

Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.

API requires the following scopes. Select which ones you want to grant to Swagger UI.

OAuth2PasswordBearer (OAuth2, password)

Token URL: `login`

Flow: `password`

username:

adamjardali@hotmail.com

password:

•

Client credentials location:

Authorization header

client_id:

client_secret:

Authorize Close

`Auth Error  Error: Forbidden`

Authorize Close

Available authorizations

×

OAuth2PasswordBearer (OAuth2, password)

Token URL: `login`

Flow: `password`

username:

adamjardali@hotmail.com

password:

•••••

Client credentials location:

Authorization header

client_id:

client_secret:

```
Auth Error  Error: Forbidden
```

Authorize Close

Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.

API requires the following scopes. Select which ones you want to grant to Swagger UI.

OAuth2PasswordBearer (OAuth2, password)

Authorized

Token URL: `login`

Flow: `password`

username: `adamjardali@hotmail.com`

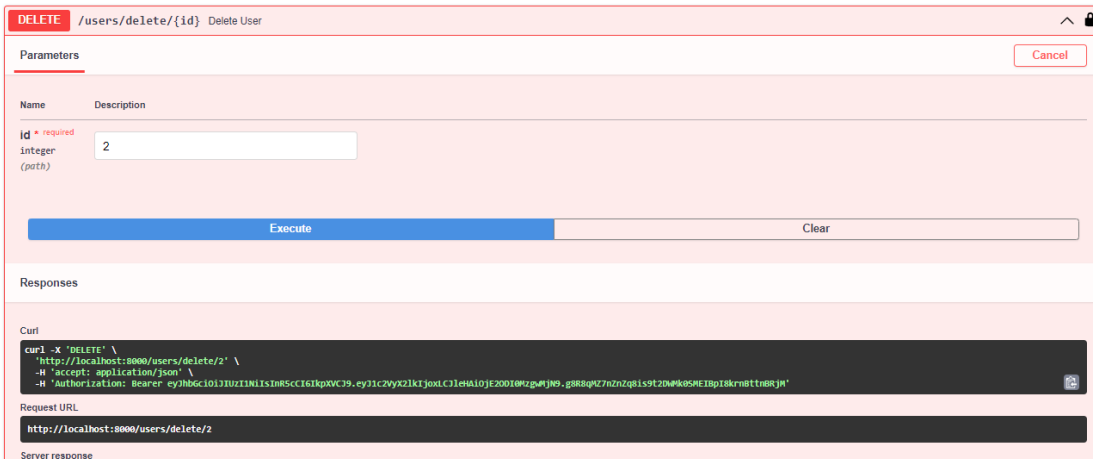
password: `******`

Client credentials location: `basic`

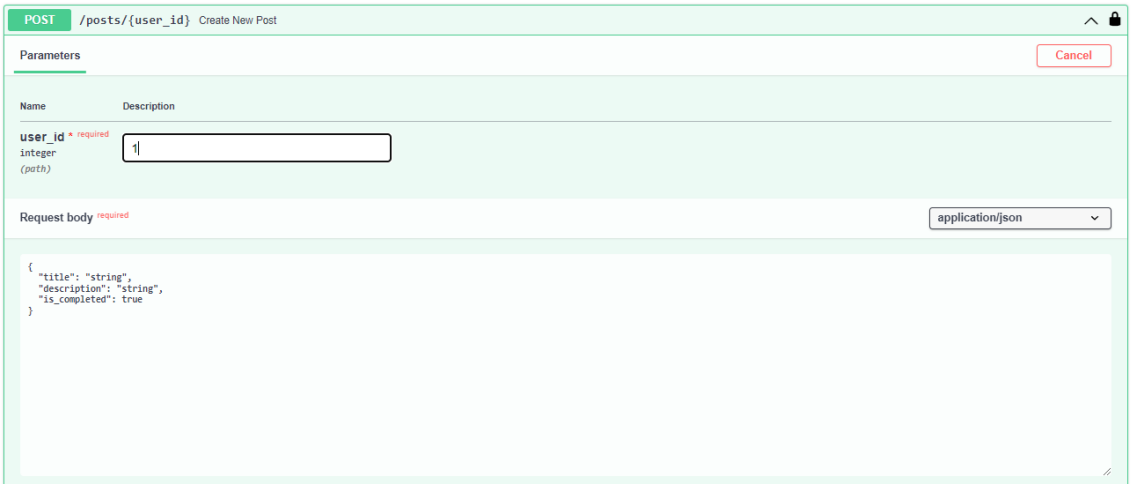
client_secret: `******`

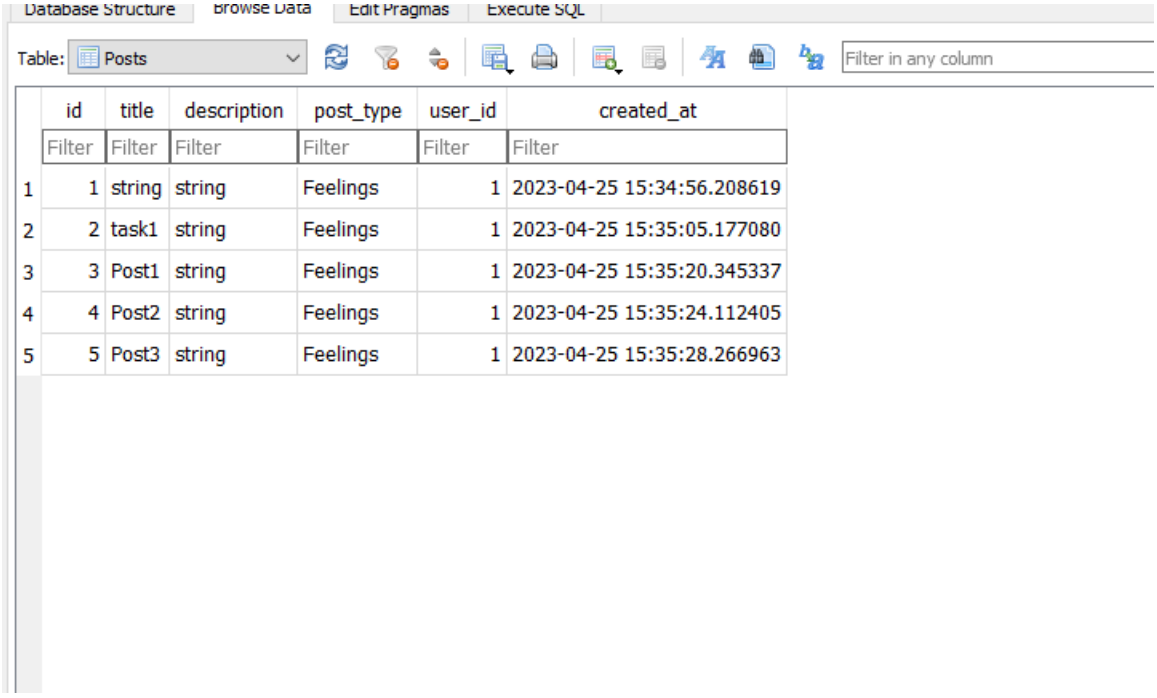
Logout | Close

Test the functionality to delete an account that you don't have the privileges to do that.



Test creating new posts and then list all of them. You should be authenticated to do that.





	id	title	description	post_type	user_id	created_at
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	string	string	Feelings	1	2023-04-25 15:34:56.208619
2	2	task1	string	Feelings	1	2023-04-25 15:35:05.177080
3	3	Post1	string	Feelings	1	2023-04-25 15:35:20.345337
4	4	Post2	string	Feelings	1	2023-04-25 15:35:24.112405
5	5	Post3	string	Feelings	1	2023-04-25 15:35:28.266963

Conclusion

As the world becomes increasingly digital, it is imperative that education keeps pace with these changes. Our mobile app aims to revolutionize the way we approach education by providing a platform that enhances the learning experience for students and teachers alike.

Using cutting-edge technology and innovative ideas, our app is designed to provide a seamless and intuitive experience for users. By integrating multimedia content, interactive exercises, and real-time feedback, we aim to create a platform that is engaging and effective.

We understand that learning can be challenging, and that's why we have put great emphasis on the user experience. With a clean and intuitive interface, users can easily navigate the app and access the features they need to enhance their learning experience.

In addition to providing a great user experience, we have also conducted rigorous testing to ensure that our app meets the highest standards of quality and reliability. By identifying and addressing any issues that arise, we are confident that our app will provide a seamless and effective learning experience for all users.

Overall, our mobile app is poised to revolutionize the way we approach education. With its innovative features, engaging content, and user-centric design, we believe that it has the potential to transform the learning experience for students and teachers alike.