



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

Introduction.....	1
FastAPI	2
Database	3
Authentication.....	7
Respiratory Pattern.....	9
Routers	10
Testing	12
Conclusion	18

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](#).

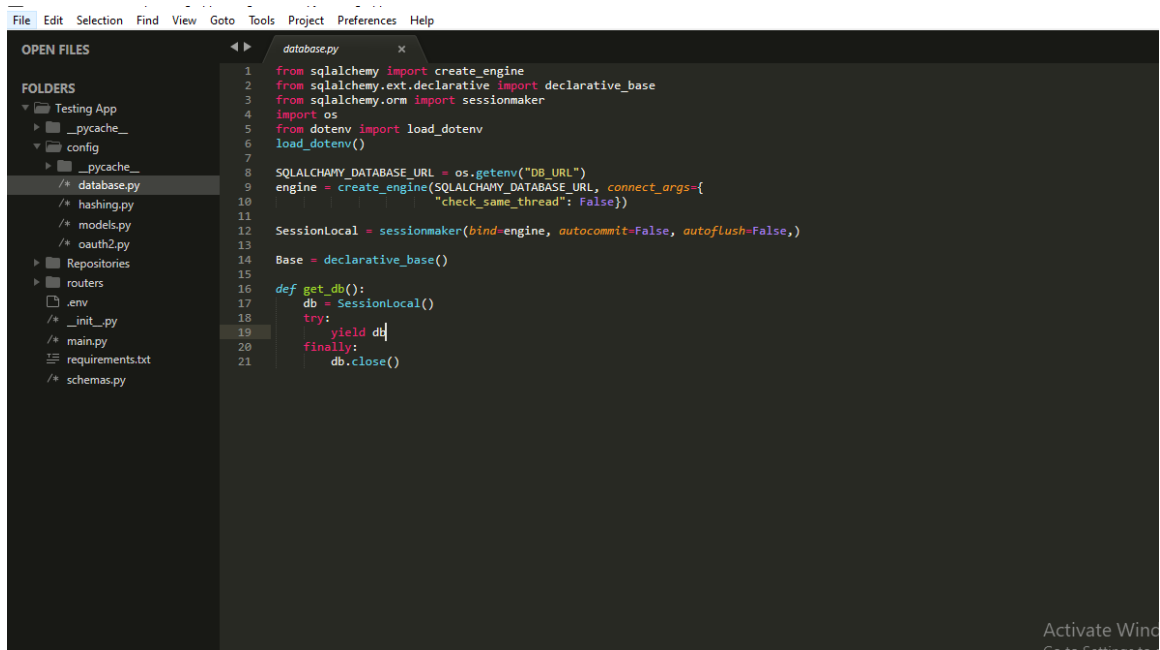
In this project we used the best practices and design patterns of FastAPI although it would take more time to implement, the code structure will get more complex, but it is efficient for scalability as the project will get bigger.

```
C:\Users\adam\Desktop\Testing App
(venv) λ ls
__init__.py  __pycache__/  App.db  config/  main.py  Repositories/  requirements.txt  routers/  schemas.py

C:\Users\adam\Desktop\Testing App
(venv) λ |
      home 174 I know that we can use os.walk() to list all sub-directories of all files in a directory. However, I
      public would like to list the full directory tree content:
      Questions
      Tags
      Users
      Companies
      collections
```

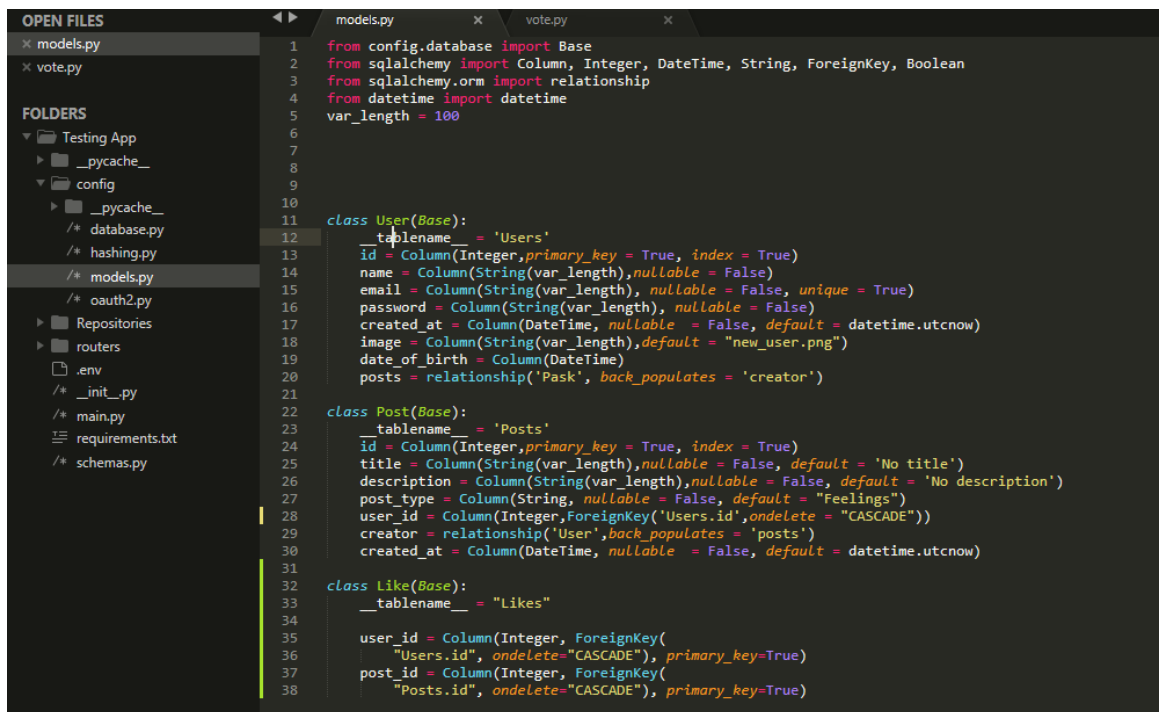
Database

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.

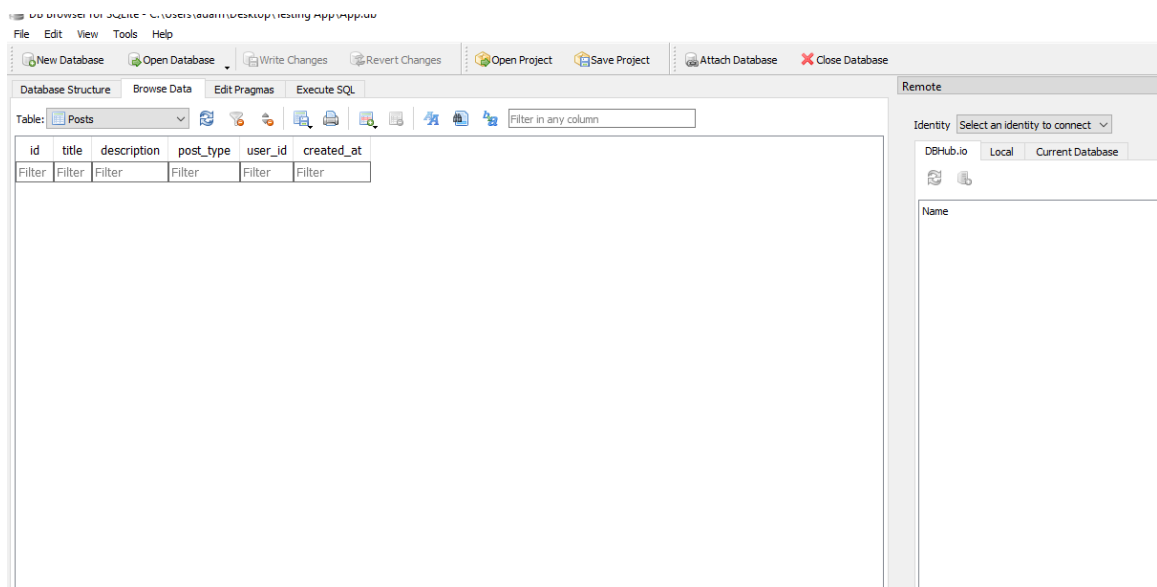
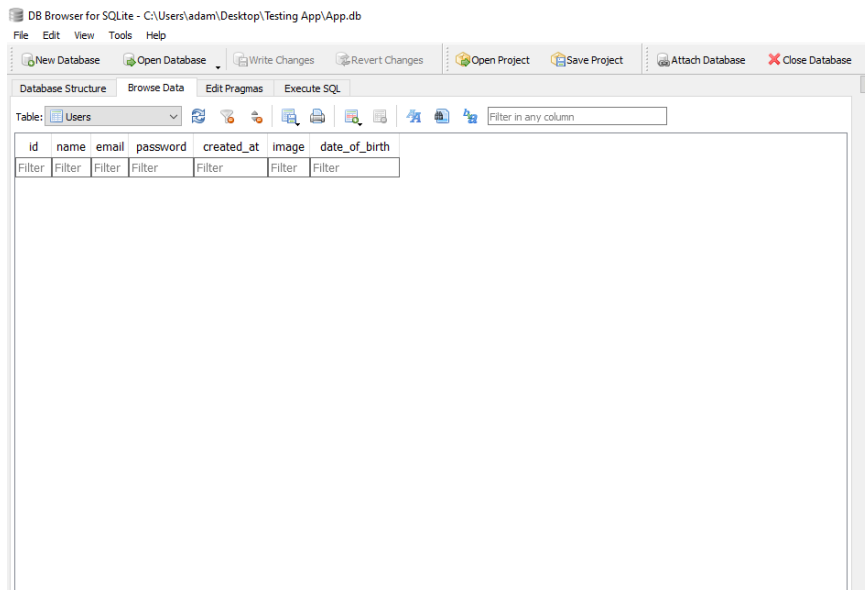


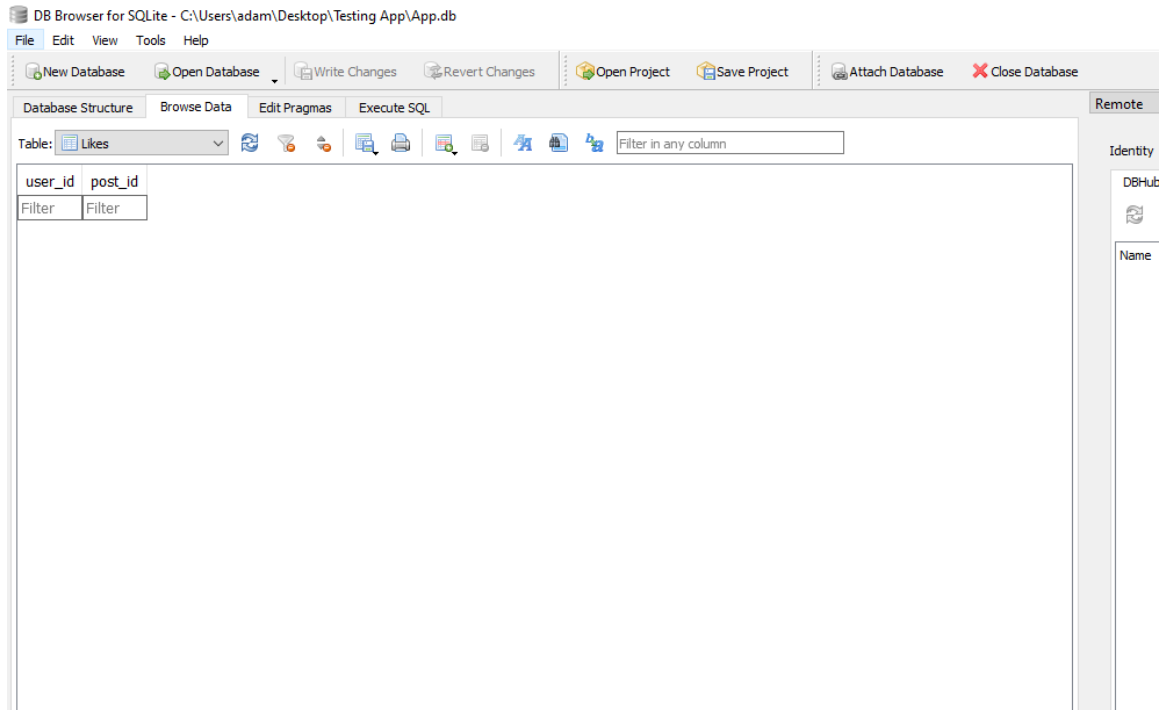
```
1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from sqlalchemy.orm import sessionmaker
4 import os
5 from dotenv import load_dotenv
6 load_dotenv()
7
8 SQLALCHEMY_DATABASE_URL = os.getenv("DB_URL")
9 engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={
10     "check_same_thread": False})
11
12 SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False,)
13
14 Base = declarative_base()
15
16 def get_db():
17     db = SessionLocal()
18     try:
19         yield db
20     finally:
21         db.close()
```

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

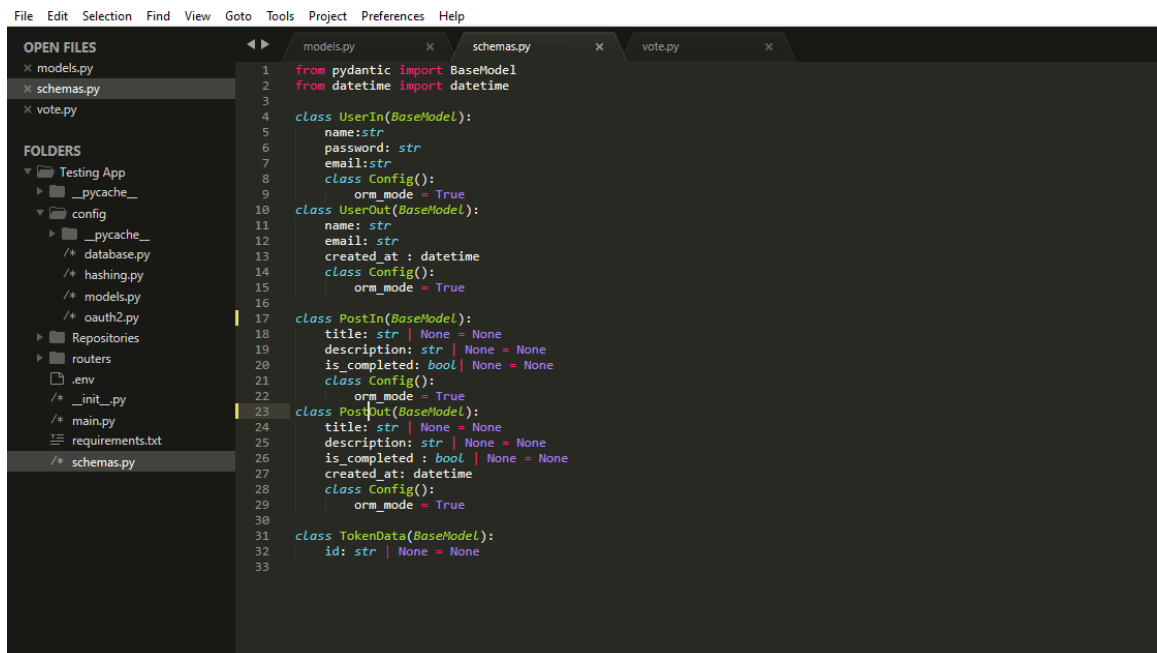


```
1 from config.database import Base
2 from sqlalchemy import Column, Integer, DateTime, String, ForeignKey, Boolean
3 from sqlalchemy.orm import relationship
4 from datetime import datetime
5 var_length = 100
6
7
8
9
10
11 class User(Base):
12     __tablename__ = 'Users'
13     id = Column(Integer, primary_key = True, index = True)
14     name = Column(String(var_length), nullable = False)
15     email = Column(String(var_length), nullable = False, unique = True)
16     password = Column(String(var_length), nullable = False)
17     created_at = Column(DateTime, nullable = False, default = datetime.utcnow)
18     image = Column(String(var_length), default = "new_user.png")
19     date_of_birth = Column(DateTime)
20     posts = relationship('Post', back_populates = 'creator')
21
22 class Post(Base):
23     __tablename__ = 'Posts'
24     id = Column(Integer, primary_key = True, index = True)
25     title = Column(String(var_length), nullable = False, default = 'No title')
26     description = Column(String(var_length), nullable = False, default = 'No description')
27     post_type = Column(String, nullable = False, default = "Feelings")
28     user_id = Column(Integer, ForeignKey('Users.id', ondelete = "CASCADE"))
29     creator = relationship('User', back_populates = 'posts')
30     created_at = Column(DateTime, nullable = False, default = datetime.utcnow)
31
32 class Like(Base):
33     __tablename__ = 'Likes'
34
35     user_id = Column(Integer, ForeignKey(
36         "Users.id", ondelete="CASCADE"), primary_key=True)
37     post_id = Column(Integer, ForeignKey(
38         "Posts.id", ondelete="CASCADE"), primary_key=True)
```





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.

```
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"

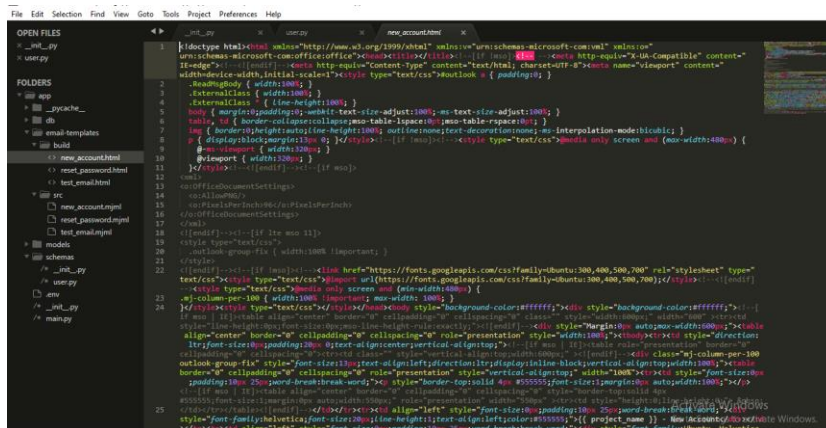
EMAIL_RESET_TOKEN_EXPIRE_HOURS: int = 48
EMAIL_TEMPLATES_DIR: str = "email-templates/build"
EMAILS_ENABLED: bool = False

@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()
```



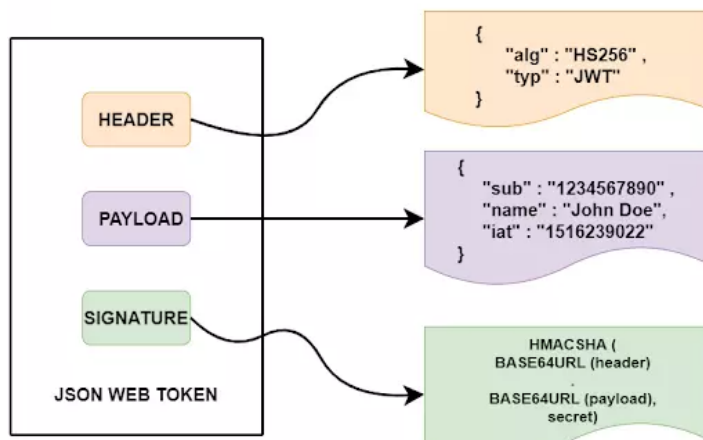
Authentication

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

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



Structure of JSON Web Token (JWT)



```
File Edit Selection Find View Goto Tools Project Preferences Help
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
1 from passlib.context import CryptContext
2 pwd_ctx = CryptContext(schemes=["bcrypt"], deprecated="auto")
3
4
5 class Hash():
6     def bcrypt(password: str):
7         return pwd_ctx.hash(password)
8
9     def verify(hashed_password, plain_password):
10        return pwd_ctx.verify(plain_password, hashed_password)
```

OPEN FILES

models.py

schemas.py

oauth2.py

vote.py

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

```

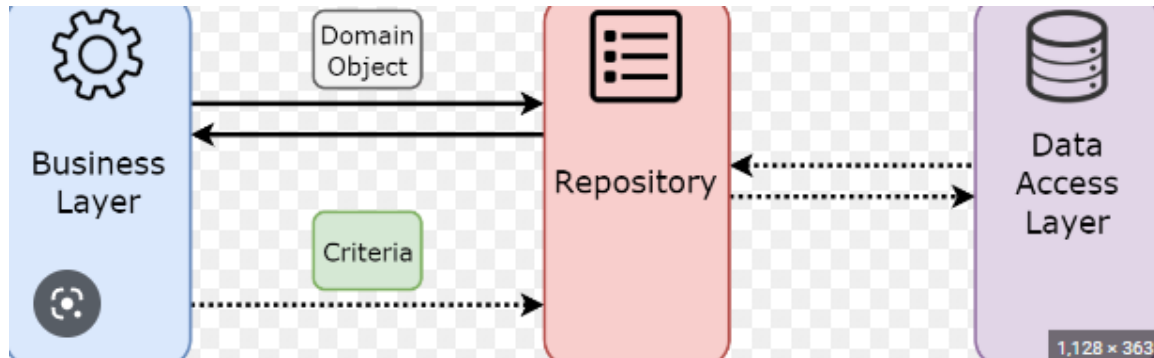
1 from fastapi import Depends, HTTPException, status
2 from fastapi.security import OAuth2PasswordBearer
3 from jose import JWTError, jwt
4 from datetime import datetime, timedelta
5 from config import database, models
6 from sqlalchemy.orm import Session
7 import schemas
8 import os
9 from dotenv import load_dotenv
10 load_dotenv()
11
12 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/login")
13
14 SECRET_KEY = os.getenv("SECRET_KEY")
15 ALGORITHM = os.getenv("ALGORITHM")
16 ACCESS_TOKEN_EXPIRE_MINUTES = os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES")
17 ACCESS_TOKEN_EXPIRE_MINUTES = 30
18
19 get_db = database.get_db
20
21 def create_access_token(data: dict):
22
23     to_encode = data.copy()
24     expire_time = datetime.utcnow() + timedelta(minutes = int(ACCESS_TOKEN_EXPIRE_MINUTES))
25     to_encode.update({"exp": expire_time})
26     token = jwt.encode(to_encode, SECRET_KEY, algorithm = ALGORITHM)
27     return token
28
29 def verify_access_token(token: str, credentials_exception):
30
31     try:
32         payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
33         id: str = payload.get("user_id")
34         if id is None:
35             raise credentials_exception
36         token_data = schemas.TokenData(id=id)
37     except JWTError:
38         raise credentials_exception
39
40     return token_data
41
42
43
44 def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
45     credentials_exception = HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
46                                           detail="Could not validate credentials", headers={"WWW-Authenticate": "Bearer"})
47
48     token = verify_access_token(token, credentials_exception)

```

8

Respiratory Pattern

Use the repository pattern for implementing each entity functionalities.



```
models.py schemas.py User.py vote.py
1 from fastapi import Depends, HTTPException, status, Response
2 from config import models, oauth2, database
3 import schemas
4 from sqlalchemy.orm import Session
5 from config.hashing import Hash
6
7 class User:
8     def register_user(user: schemas.UserIn, db: Session):
9         hashed_password = Hash.bcrypt(user.password)
10        user.password = hashed_password
11        is_unique_email = db.query(models.User).filter(models.User.email == user.email)
12        if is_unique_email.first():
13            raise HTTPException(status_code = 409, detail = f"Email {user.email} is taken by another user")
14
15        new_user = models.User(**user.dict())
16        db.add(new_user)
17        db.commit()
18        db.refresh(new_user)
19        return new_user
20
21    def get_user_by_id(id: int, db: Session):
22        user = db.query(models.User).filter(models.User.id == id).first()
23        if not user:
24            raise HTTPException(status_code = 404, detail = f"User with id {id} is not found")
25        return user
26
27    def delete_user(id: int, db: Session, current_user : int):
28        user = db.query(models.User).filter(models.User.id == id)
29        if not user.first():
30            raise HTTPException(status_code = 404, detail = f"User with id {id} is not found")
31        if user.first().id != current_user.id:
32            raise HTTPException(status_code = 403, detail = f"Not authorized to perform requested action")
33        user.delete(synchronize_session = False)
34        db.commit()
35        return Response(status_code = 204)
36
37    def update_user(id: int, new_user: schemas.UserIn, db: Session, current_user : int):
38        user = db.query(models.User).filter(models.User.id == id)
39        if not user.first():
40            raise HTTPException(status_code = 404, detail = f"User with id {id} is not found")
41        if user.first().id != current_user.id:
42            raise HTTPException(status_code = 403, detail = f"Not authorized to perform requested action")
43        user.update(new_user.dict(), synchronize_session = False)
44        db.commit()
45        return user.first()
46
```

Routers

Implement the routers of each entity.

```
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)
```

```
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'}
```

```

4 from sqlalchemy.orm import Session
5 from typing import List
6 from repositories.task import Task
7
8 router = APIRouter(
9     prefix = '/posts',
10     tags = ['Posts']
11 )
12
13 get_db = database.get_db
14
15 @router.post("/{user_id}", response_model = schemas.TaskOut)
16 def create_new_post(user_id: int, task: schemas.TaskIn, db: Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
17     return Task.create_new_task(user_id, task, db, current_user)
18
19
20 @router.get("/{user_id}/tasks", response_model = List[schemas.TaskOut])
21 def get_all_posts(user_id: int, db: Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
22     return Task.get_all_tasks(user_id, db, current_user)
23
24 @router.get("/{id}/sortedposts", response_model = List[schemas.TaskOut])
25 def get_sorted_posts(id: int, db: Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
26     return Task.get_sorted_tasks(id, db, current_user)
27
28 @router.get("/{id}", response_model = schemas.TaskOut)
29 def get_post_by_id(id: int, db: Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
30     return Task.get_task_by_id(id, db, current_user)
31
32 @router.get("/{id}/completed", response_model = List[schemas.TaskOut])
33 def get_favorite_posts(id: int, db: Session = Depends(get_db), current_user: int = Depends(oauth2.get_current_user)):
34     return Task.get_completed_tasks(id, db, current_user)
35
36 @router.get("/{id}/pagintation", response_model = List[schemas.TaskOut])
37 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)):
38     return Task.get_custom_tasks(id, skip, limit, db, current_user)
39
40 @router.put("/update/{user_id}/{task_id}", response_model = schemas.TaskOut)
41 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)):
42     return Task.update_task(user_id, task_id, db, current_user)
43
44
45 @router.delete("/update/{user_id}/{task_id}")

```

Testing

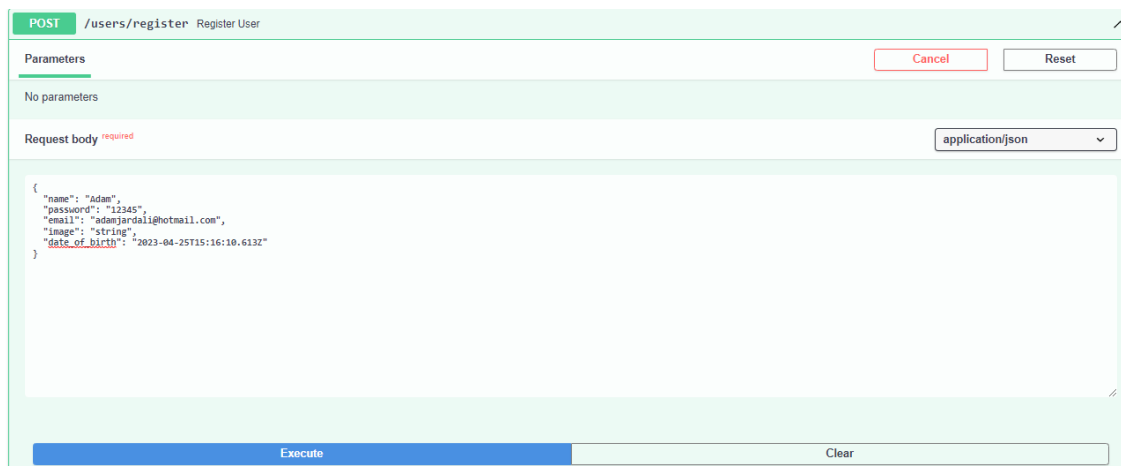
Now it is time for testing. No need to use postman as FastAPI provides a UI for that purpose.

FastAPI 
/openapi.json



The image shows the FastAPI Swagger UI interface. At the top right, there is an "Authorize" button with a lock icon. The interface is divided into three main sections: "Users", "Authentication", and "Posts". Each section contains a list of endpoints with their respective HTTP methods, URLs, and descriptions. The "Users" section includes endpoints for GET /users/ (Main Page), POST /users/register (Register User), GET /users/{id} (Get User By Id), DELETE /users/delete/{id} (Delete User), and PUT /users/update/{id} (Update User). The "Authentication" section includes a POST /login (Login) endpoint. The "Posts" section includes endpoints for POST /posts/{user_id} (Create New Post), GET /posts/{user_id}/tasks (Get All Posts), GET /posts/{id}/sortedposts (Get Sorted Posts), GET /posts/{id} (Get Post By Id), GET /posts/{id}/completed (Get Favorite Posts), GET /posts/{id}/pagintation (Get Custom Posts), PUT /posts/update/{user_id}/{task_id} (Update Post), and DELETE /posts/update/{user_id}/{task_id} (Delete Post). Each endpoint is represented by a colored bar with a dropdown arrow on the right.

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



The image shows the FastAPI Swagger UI interface for the POST /users/register endpoint. The endpoint is highlighted in green. Below the endpoint name, there are "Parameters" and "Request body" sections. The "Parameters" section shows "No parameters". The "Request body" section is marked as "required" and has a dropdown menu set to "application/json". The request body is a JSON object with the following fields: "name" (string), "password" (string), "email" (string), "image" (string), and "date_of_birth" (string). The "Execute" button is highlighted in blue, and the "Clear" button is in a light gray box.

Curl

```
curl -X 'POST' \
  'http://localhost:8000/users/register' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Adam",
    "password": "12345",
    "email": "adamjardali@hotmail.com",
    "image": "string",
    "date_of_birth": "2023-04-25T15:16:10.613Z"
  }'
```

Request URL

http://localhost:8000/users/register

Server response

Code Details

201

Response body

```
{
  "name": "Adam",
  "email": "adamjardali@hotmail.com",
  "created_at": "2023-04-25T15:18:52.284163",
  "image": "string",
  "date_of_birth": "2023-04-25T15:16:10.613000"
}
```

Response headers

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

	id	name	email	password	created_at
Filter	Filter	Filter	Filter	Filter	Filter
1	1	Adam	adamjardali@hotmail.com	\$2b\$12\$4LNF41gFVGth/...	2023-04-25 15:18:52.284163

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

Available authorizations

x

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:

password:

Client credentials location:

Authorization header ▾

client_id:

client_secret:

Authorize

Close

Auth Error Error: Forbidden

Authorize

Close

Available authorizations

x

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

X

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

Authorized

Flow: password

password: *****

```
client_secret: *****
```

Close

DELETE

/users/delete/{id} Delete User

Parameters

Cancel

Name	Description
<div><div>id required</div><div>integer (path)</div></div>	<div><div>2</div></div>

Execute

Clear

Responses

Curl

```
curl -X "DELETE" \
  "http://localhost:8000/users/delete/2" \
  -H "accept: application/json" \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp1VC99.eyJ1IjoiYm92Z2ktYXNjaW99E20018WzgWJm99.g8R8g9Z7nZn2q81s9T20Wk6sMEI8p18krm8Ttn8RjM"
```

Request URL

```
http://localhost:8000/users/delete/2
```

Response body

Download

```
content-length: 55
content-type: application/json
date: Tue, 25 Apr 2023 15:26:35 GMT
server: uvicorn
```

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

POST /posts/{user_id} Create New Post

Parameters Cancel

Name	Description
user_id * required integer (path)	<input type="text" value="1"/>

Request body **required** application/json

```
{  
  "title": "string",  
  "description": "string",  
  "is_completed": true  
}
```

Code Details

200

Response body

```
{  
  "title": "Post3",  
  "description": "string",  
}
```

Database Structure browse Data Edit Pragmas Execute SQL

Table: Posts

	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.