FastAPI

Beginners Handnote

Initial project configuration

- Have a virtual environment
- Create a directory
- Initialize git (`git init`)
- Add `.gitignore` file (user `gitignore.io`)
- Create another directory named ‘backend’
- Create a file named ‘requirements.txt’ inside backend
- Create a file named ‘main.py’ inside backend
- Create another folder inside backend named ‘core’
- Create a file inside ‘core’ named ‘config.py’
- requirements.txt file would contains (`fastapi==0.115.6 uvicorn==0.32.1`)
- Install dependencies.

Initial project configuration (cont'd)

```
├─.gitignore
└─backend/
  ├─core/
  │  └─config.py
  ├─main.py
  └─requirements.txt
```

Hello World

Write some code inside ‘core/config.py’

```
#core/config.py

class Settings:
    PROJECT_NAME:str = "Innovative 🚀"
    PROJECT_VERSION: str = "1.0.0"

settings = Settings()
```

Hello World (cont'd)

Write main.py file

```
#main.py

from fastapi import FastAPI
from core.config import settings

app = FastAPI(title=settings.PROJECT_NAME,version=settings.PROJECT_VERSION)

@app.get("/")
def hello_api():
    return {"msg":"Hello FastAPI"}
```

Hello World (cont'd)

Run the server!

```
uvicorn main:app --reload
```

Now in the browser. Goto this address : http://127.0.0.1:8000/

Or some documentation (swagger), go to this address: http://127.0.0.1:8000/docs

Connecting to Database

Update the requirements.txt

```
fastapi==0.95.1

uvicorn==0.22.0

#new

SQLAlchemy==2.0.13

psycopg2-binary==2.9.6    #windows user use: psycopg2-==2.9.6

python-dotenv==1.0.0
```

Run '`pip install -r requirements.txt`'

Connecting to Database [Environment Variables]

Create a **`.env`** at the ‘backend’ directory. Paste the content below.

```
POSTGRES_USER=dummy
POSTGRES_PASSWORD=secret
POSTGRES_SERVER=localhost
POSTGRES_PORT=5432
POSTGRES_DB=blogdb
```

Connecting to Database [Add Configuration File]

```
# backend/config.py

import os

from dotenv import load_dotenv

from pathlib import Path


env_path = Path('.') / '.env'

load_dotenv(dotenv_path=env_path)
```

From the previous file, continues...

```
class Settings:

    PROJECT_NAME:str = "Blog Board"

    PROJECT_VERSION: str = "1.0.0"


    POSTGRES_USER : str = os.getenv("POSTGRES_USER")

    POSTGRES_PASSWORD = os.getenv("POSTGRES_PASSWORD")

    POSTGRES_SERVER : str = os.getenv("POSTGRES_SERVER","localhost")

    POSTGRES_PORT : str = os.getenv("POSTGRES_PORT",5432) # default postgres port is 5432

    POSTGRES_DB : str = os.getenv("POSTGRES_DB","tdd")

    DATABASE_URL = f"postgresql://{POSTGRES_USER}:{POSTGRES_PASSWORD}@{POSTGRES_SERVER}:{POSTGRES_PORT}/{POSTGRES_DB}"


settings = Settings()
```

Connecting to Database [Session File]

Create a new folder under backend named **db**. Here make a new file ‘session.py’

```
#db/session.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from core.config import settings


SQLALCHEMY_DATABASE_URL = settings.DATABASE_URL
print("Database URL is ",SQLALCHEMY_DATABASE_URL)
engine = create_engine(SQLALCHEMY_DATABASE_URL)

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

Installation!

Add a new package named ‘inflect’ which will help use to convert a singular word to plural. In requirements.txt:

- inflect == 7.5.0

And install requirements.txt

- Create two new files under ‘db/’ named it ‘base_class.py’ & ‘base.py’

Connecting to Database [Base Class]

```
#db/base_class.py
import inflect
from typing import Any
from sqlalchemy.ext.declarative import declared_attr
from sqlalchemy.orm import as_declarative


# Create an inflect engine
p = inflect.engine()

@as_declarative()
class Base:
   id: Any
   __name__: str

   # to generate tablename from classname
   @declared_attr
   def __tablename__(cls) → str:
       # Convert class name to lowercase and pluralize it
       singular_name = cls.__name__.lower()
       plural_name = p.plural(singular_name)
       return plural_name
```

Import models into base.py

This **Base** class of ‘blass_class.py’ file should be imported into base.py

In base.py

```
from db.base_class import Base
```

Database Migration (Initialize alembic)

- Modify ‘requirements.txt’ and add a new package ‘`alembic==1.11.1`’
- Run: `pip install -r requirements.txt`
- Now, come to the directory where main.py is situated & goto terminal , then run a command:
- `alembic init alembic`
- (One file ‘`alembic.ini`’ & one directory `‘alembic/’` will be created)

Database Migration (Configure alembic)

- Go to ‘alembic/env.py’ file

#Import these

```
from core.config import settings
from db.base import Base
```

- Search where is ‘`config = context.config`’ Just the next line , write the code:

```
config.set_main_option("sqlalchemy.url",settings.DATABASE_URL)
```

- Search where is ‘`target_metadata = None`’ remove the line , write the code:

```
target_metadata = Base.metadata
```

Database Migration (Run First Migration)

- Now Write our first migration:

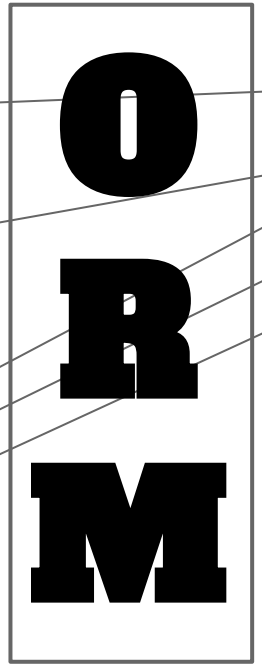
```
alembic revision --autogenerate -m 'initial migration'
```

- A new folder will be created under ‘alembic/’ folder named ‘versions/’
- Here a new migration file would be found
- To confirm this migration by running the below command:

```
alembic upgrade head
```

WHAT IS ORM?

```
class Person:
    def __init__(self , id , first_name ,
last_name, phone):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.phone = phone


Person(1,"Jhon","Connor","+16105551234")
Person(2,"Matt","Makai","+12025555689")
Person(3,"Sarah","Smit","+19735554512")
```



Person

ID	FIRST_NAME	LAST_NAME	PHONE
1	John	Connor	+16105551234
2	Matt	Makai	+12025555689
3	Sarah	Smith	+19735554512
...	...	...	...

Table Creation

- Create a folder under db/ , name it ‘models/’
- Create two files under ‘models/’ named ‘blog.py’ , ‘user.py’
- Create the first model (user)

User model creation

```
from db.base_class import Base
from sqlalchemy import Boolean
from sqlalchemy import Column
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy.orm import relationship


class User(Base):
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, nullable=False, unique=True, index=True)
    password = Column(String, nullable=False)
    is_superuser = Column(Boolean(), default=False)
    is_active = Column(Boolean(), default=True)
    blogs = relationship("Blog",back_populates="author")
```

Run Migration for User

- Import User model into base.py
- Generate migration file
- Apply migration

Blog model creation

```
from datetime import datetime
from sqlalchemy import Column, Integer, Text, String, Boolean, DateTime,
ForeignKey
from sqlalchemy.orm import relationship

from db.base_class import Base

class Blog(Base):
    id = Column(Integer, primary_key=True)
    title = Column(String, nullable=False)
    slug = Column(String, nullable=False)
    content = Column(Text, nullable=True)
    author_id =  Column(Integer,ForeignKey("users.id"))
    created_at = Column(DateTime, default=datetime.now)
    is_active = Column(Boolean, default=False)
    author = relationship("User",back_populates="blogs")
```

Run Migration for Blog

- Import Blog model into base.py
- Generate migration file
- Apply migration

Introduce Pydantic in our project

- Add ‘pydantic==1.10.0’ into requirements.txt
- Install it
- Add a new folder named ‘schemas’ into the root directory
- Add two files (user.py) and (blog.py) under schema folder

First schema

Complete user schema for user creation

```
from pydantic import BaseModel, Field

#properties required during user creation
class UserCreate(BaseModel):
    email : str
    password : str = Field( ... , min_length=4)
```

This is responsible for handling the json like below

```
{'email':'demo@gmail.com','password':'testing'}
```

Add mechanism to get db

Modify db>session.py file. Add below code:

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

This ‘get_db()’ is responsible to give our database session to work with database

Test

```
>>> from db.session import get_db
Database URL is  postgresql://dummy:secret@localhost:5432/blogdb
>>> from db.models.user import User
>>> db = get_db().__next__()
>>> user = db.query(User).all()
```

Password should be hashed!

- Add ‘passlib==1.7.4’ in requirements.txt. And Install it.
- Create a folder named ‘utils’ under root directory. And create a file named ‘password_manager.py’ into it.

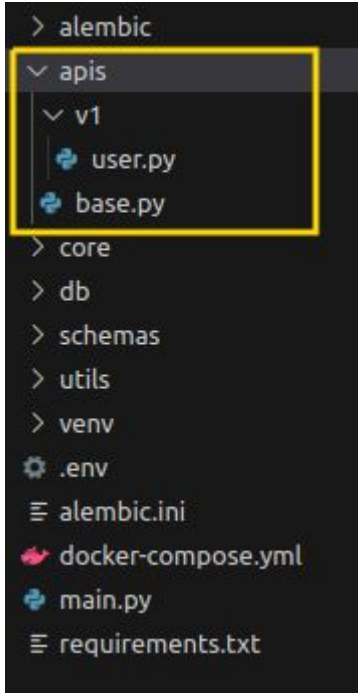
```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")


class PasswordManager():
    @staticmethod
    def verify_password(plain_password, hashed_password):
        return pwd_context.verify(plain_password, hashed_password)

    @staticmethod
    def get_password_hash(password):
        return pwd_context.hash(password)
```

Test

```
>>> from utils.password_manager import PasswordManager
>>> hash = PasswordManager.get_password_hash('12345')
>>> hash
'$2b$12$sINGaDf0V2nY0X8/PRK6QuJcxaOMdvYNGmfR30ht/w/vT0WlDY25G'
>>> PasswordManager.verify_password('12345', hash)
True
>>> PasswordManager.verify_password('abcd', hash)
False
```

Creating User Route

- Create this folders and files as shown here
- ‘apis/’ under root
- ‘v1/’ under ‘apis/’
- ‘base.py’ under ‘apis/’
- ‘user.py’ under ‘v1/’



Let’s make the router for the user. ‘apis/v1/user.py’ in this file, put the below code

```
from fastapi import APIRouter
from sqlalchemy.orm import Session
from fastapi import Depends
from db.session import get_db

router = APIRouter()

@router.post("/")
def create_user(db: Session = Depends(get_db)):
   # WE WILL SOMETHING DO HERE
   return {"message" : "hello"}
```

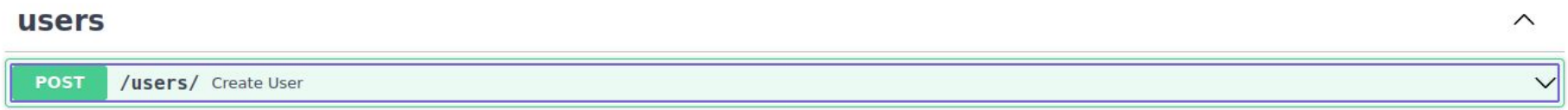
Register this user router into the router base (*apis/base.py*) .

```
from fastapi import APIRouter

from apis.v1 import user

api_router = APIRouter()
api_router.include_router(user.router, prefix="/users", tags=["users"])
```

Introduce router to our app (one time)

```
#main.py
from fastapi import FastAPI
from core.config import settings
from apis.base import api_router

def start_application():
   app = FastAPI(title=settings.PROJECT_NAME,version=settings.PROJECT_VERSION)
   return app

app = start_application()
app.include_router(api_router)

@app.get("/")
def home():
   return {"msg":"Hello World"}
```

This is what shows our swagger.



Not place a trailing '/'

users

POST /users/ Create User

We should avoid this practice. Any router shouldn't have a trailing '/' So edit 'apis/v1/user.py'

```
from fastapi import APIRouter
from sqlalchemy.orm import Session
from fastapi import Depends
from db.session import get_db

router = APIRouter()

@router.post("")
def create_user(db: Session = Depends(get_db)):
   # WE WILL SOMETHING DO HERE
   return {"message" : "hello"}
```

users

POST /users Create User

Add Request & Response Schema

It's better to have a concrete schema for request payload & response payload. Pydantic helps us to create such schemas. We already create a schema for request payload for user create. Now we will add a response schema in the same file '*schemas/user.py*'

```
from email_validator import validate_email, EmailNotValidError
from pydantic import Basemodel, Field
from fastapi import HTTPException, status

class UserCreate(BaseModel):
   email : str
   password : str = Field( ... , min_length=4)

   def __init__(self, **data):
       super().__init__(**data)

       if self.email:
           try:
               emailinfo = validate_email(self.email, check_deliverability=False)
               self.email = emailinfo.normalized
           except EmailNotValidError as e:
               raise HTTPException(
                   status_code=status.HTTP_400_BAD_REQUEST,
                   detail="Not a valid email!"
               )
```

#schemas/user.py continues

```
class UserView(BaseModel):
   id: int
   email : EmailStr
   is_active : bool

   class Config():  #tells pydantic to convert even non dict obj to json
       orm_mode = True
```

`UserCreate` Is for our request payload & `UserView` for our response model

Now , We will introduce these schemas to our /users router

```
from fastapi import APIRouter
from sqlalchemy.orm import Session
from fastapi import Depends
from db.session import get_db
from schemas.user import UserCreate , UserView

router = APIRouter()

@router.post("", response_model=UserView)
def create_user(
   payload: UserCreate,
   db: Session = Depends(get_db)
):
   print("The payload is: ", payload)
   return UserView(id=1,email="demo@example.com", is_active=True)
```

Now, goto the swagger to test the request & response payload. After hitting the route, we will see in the terminal.

```
The payload is:  email='user@example.com' password='abc123'
INFO:     127.0.0.1:46310 - "POST /users HTTP/1.1" 200 OK
```

users

POST `/users` Create User

Parameters

Cancel | Reset

No parameters

Request body required

application/json

```
{
  "email": "user@example.com",
  "password": "abc123"
}
```

Execute | Clear

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/users' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "email": "user@example.com",
  "password": "abc123"
}'
```

Request URL

```
http://127.0.0.1:8000/users
```

Server response

Code	Details
200	Response body

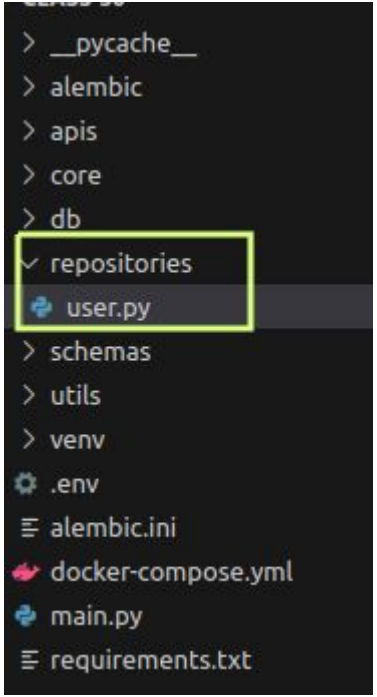
```
{
  "id": 1,
  "email": "demo@example.com",
  "is_active": true
}
```

Download

Now go to the database part.

As per ‘Repository Pattern’ , all db operations would be at repository layer. To handle user model we need a ‘user repository’ . Create a folder named ‘**repositories/**’ under root directory. And add a file named ‘user.py’

```
from sqlalchemy.orm import Session
from sqlalchemy.exc import IntegrityError
from typing import Optional
from db.models.user import User
from fastapi import HTTPException
from utils.password_manager import PasswordManager

class UserRepository:
   def __init__(self, db: Session):
       self.db = db

   def get_user_by_email(self, email: str) → Optional[User]:
       return self.db.query(User).filter(User.email == email).first()

   def create_user(self, email: str, password: str, is_active: bool = True, is_superuser: bool = False) → User:
       _hashed_password = PasswordManager.get_password_hash(password=password)
       db_user = User(email=email, password=_hashed_password, is_active=is_active, is_superuser=is_superuser)
       self.db.add(db_user)
       try:
           self.db.commit()
           self.db.refresh(db_user)
       except IntegrityError:
           self.db.rollback()
           raise HTTPException(status_code=400, detail="Email already registered")
       return db_user
```



Call Repo from api

```
from fastapi import APIRouter, HTTPException, status
from sqlalchemy.orm import Session
from fastapi import Depends
from db.session import get_db
from repositories.user import UserRepository
from schemas.user import UserCreate , UserView


router = APIRouter()

def get_user_repo(db: Session) → UserRepository:
   return UserRepository(db)


@router.post("", response_model=UserView)
def create_user(
   payload: UserCreate,
   db: Session = Depends(get_db)
):
   user_repo = get_user_repo(db=db)
   existing_user = user_repo.get_user_by_email(email=payload.email)

   if existing_user:
       raise HTTPException(
           status_code=status.HTTP_400_BAD_REQUEST,
           detail="Email already registered"
       )

   new_user = user_repo.create_user(email=payload.email,password=payload.password)
   return UserView(id=new_user.id,email=new_user.email, is_active=new_user.is_active)
```

Some Refactor

- What if an email is ‘nahid@gmail.com’ and another is ‘NaHiD@gmail.com’ . Do you think , these two emails are different ? Nope. But, in our existing codebase, it case-insensitive filtering is not done yet. Let’s do. Modify UserRepository -> **get_user_by_email()** function. *Of course import ‘func’ directly from sqlalchemy*

```
    def get_user_by_email(self, email: str) -> Optional[User]:
        return self.db.query(User).filter(func.lower(User.email) == func.lower(email)).first()
```

- Remember at the schema level, we did an ‘orm_mode=True’ ? But, we still don’t utilize it.

```
@router.post("", response_model=UserView)
def create_user( payload: UserCreate, db: Session = Depends(get_db)):
    # existing code remain same
    new_user = user_repo.create_user(email=payload.email,password=payload.password)
    return new_user
```

What's Next ?

blogs

GET	/blogs	Get Blogs
POST	/blogs	Create User
GET	/blogs/{blog_id}	Get Blogs
PUT	/blogs/{blog_id}	Update Blog
DELETE	/blogs/{blog_id}	Delete Blog

Make crud router's for our blog.

Blog CRUD (Post)

Schema

schemas/blog.py

```
from datetime import datetime
from pydantic import BaseModel
from typing import Optional
from slugify import slugify
import time
```

```
class BlogRead(BaseModel):
    id: int
    slug: str
    author_id: int
    created_at: datetime

    class Config:
        orm_mode = True
```

```
class BlogCreate(BaseModel):
    title: str
    content: str
    is_active: bool = False
    slug: Optional[str] = None

    @classmethod
    def create_slug(cls, title: str) → str:
        # Automatically generate a slug from the title
        _slugify = slugify(title)
        _time_hash = hash(time.time())
        return f"{_slugify}-{_time_hash}"

    # Override the `__init__` method to automatically generate the slug
    def __init__(self, **data):
        super().__init__(**data)
        if self.title:
            self.slug = self.create_slug(self.title)
```

Repository

repositories/blog.py

```
from sqlalchemy.orm import Session
from typing import List, Optional
from db.models.blog import Blog
from schemas.blog import BlogCreate,
BlogUpdate
from sqlalchemy.exc import IntegrityError
from fastapi import HTTPException
```

```
class BlogRepository:
    def __init__(self, db: Session):
        self.db = db

    def create_blog(self, blog: BlogCreate, author_id: int) → Blog:
        """
        Create a new blog in the database.
        """
        db_blog = Blog(
            title=blog.title,
            slug=blog.slug,
            content=blog.content,
            is_active=blog.is_active,
            author_id=author_id
        )
        try:
            self.db.add(db_blog)
            self.db.commit()
            self.db.refresh(db_blog)
        except IntegrityError as e:
            print(e)
            self.db.rollback()
            raise HTTPException(status_code=400, detail=f"Something went wrong!")

        return db_blog
```

Router

apis/v1/blog.py

```
from fastapi import APIRouter
from sqlalchemy.orm import Session
from fastapi import Depends
from db.session import get_db
from repositories.blog import BlogRepository
from schemas.blog import BlogCreate, BlogRead

router = APIRouter()

@router.post("", response_model=BlogRead)
def create_user(payload: BlogCreate, db: Session = Depends(get_db)):
    blog_repo = BlogRepository(db=db)
    new_blog = blog_repo.create_blog(blog=payload, author_id=2)
    return new_blog
```

apis/v1/base.py

```
from fastapi import APIRouter
from apis.v1 import user, blog

api_router = APIRouter()
api_router.include_router(user.router, prefix="/users", tags=["users"])
api_router.include_router(blog.router, prefix="/blogs", tags=["blogs"])
```

blogs

POST /blogs Create User

Parameters

Cancel Reset

No parameters

Request body required

application/json

```
{
  "title": "Python for beginners",
  "content": "string",
  "is_active": false
}
```

Request URL

```
http://127.0.0.1:8000/blogs
```

Server response

Code	Details
200	**Response body**

```
{
  "id": 3,
  "slug": "python-for-beginners-1683303925350013493",
  "author_id": 2,
  "created_at": "2025-01-11T13:13:57.765044"
}
```

Download

Blog CRUD (List Get)

db/models/blog.py

```
from datetime import datetime
from sqlalchemy import Column, Integer, Text, String, Boolean, DateTime, ForeignKey
from sqlalchemy.orm import relationship

from db.base_class import Base

class Blog(Base):
   id = Column(Integer, primary_key=True)
   title = Column(String, nullable=False)
   slug = Column(String, nullable=False)
   content = Column(Text, nullable=True)
   author_id =  Column(Integer,ForeignKey("users.id"))
   author = relationship("User",back_populates="blogs")
   created_at = Column(DateTime, default=datetime.now)
   is_active = Column(Boolean, default=False)

   author = relationship("User")
```

User is the model name

schemas/blog.py

We have to make sure that , list view should be paginated
For the performance!

```
from schemas.user import UserView


class BlogRead(BaseModel):
   id: int
   slug: str
   created_at: datetime
   author: UserView

   class Config:
       orm_mode = True

class BlogPagination(BaseModel):
   total_count: int
   skip: int
   limit: int
   data: List[BlogRead]

   class Config:
       orm_mode = True
```

This must be same as 'Blog' model's relationship variable name

Pydantic is smart enough to get the related object from the query

repositories/blog.py

Create an instance method in `BlogRepository` class

```
def get_blogs(self, skip: int = 0, limit: int = 100) → List[Blog]:
        """
        Retrieve a list of blogs with pagination.
        """
        total_count = self.db.query(func.count(Blog.id)).scalar()  # Get the total number of blogs
        blogs = self.db.query(Blog).offset(skip).limit(limit).all()

        return BlogPagination(
            total_count=total_count,
            skip=skip,
            limit=limit,
            data=blogs
        )
```

apis/v1/blog.py

```python
@router.get("", response_model=BlogPagination)
def get_blogs(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    blog_repo = BlogRepository(db=db)
    blogs = blog_repo.get_blogs(skip=skip, limit=limit)
    return blogs
```

```
curl -X 'GET' \
  'http://127.0.0.1:8000/blogs?skip=0&limit=100' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/blogs?skip=0&limit=100
```

Server response

Code	Details
200	

Response body


```
{
  "total_count": 1,
  "skip": 0,
  "limit": 100,
  "data": [
    {
      "id": 3,
      "slug": "python-for-beginners-1683303925350013493",
      "created_at": "2025-01-11T13:13:57.765044",
      "author": {
        "id": 2,
        "email": "rr@gmail.com",
        "is_active": true
      }
    }
  ]
}
```


Download

Blog CRUD (Single Get)

`repositories/blog.py`

Create an instance method in `BlogRepository` class

```
def get_blog(self, blog_id: int) -> Optional[Blog]:
    """
    Retrieve a single blog by its ID.
    """
    return self.db.query(Blog).filter(Blog.id == blog_id).first()
```

apis/v1/blog.py

```python
@router.get("/{blog_id}", response_model=BlogRead)
def get_blogs(blog_id: int, db: Session = Depends(get_db)):
    blog_repo = BlogRepository(db=db)
    blog = blog_repo.get_blog(blog_id=blog_id)
    if not blog:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Blog not found!"
        )
    return blog
```

Request URL

http://127.0.0.1:8000/blogs/3

Server response

Code	Details
200	**Response body**


```
{
  "id": 3,
  "slug": "python-for-beginners-1683303925350013493",
  "created_at": "2025-01-11T13:13:57.765044",
  "author": {
    "id": 2,
    "email": "rr@gmail.com",
    "is_active": true
  }
}
```


Download

Exercise!

Could you make a router, which takes ‘blog slug’ as a parameter and returns it’s details ?

Blog CRUD (Update)

schemas/blog.py

```
class BlogUpdate(BaseModel):
   title: Optional[str] = None
   content: Optional[str] = None
   is_active: Optional[bool] = None
   slug: Optional[str] = None


   @classmethod
   def create_slug(cls, title: str) → str:
       # Automatically generate a slug from the title
       _slugify = slugify(title)
       _time_hash = hash(time.time())
       return f"{_slugify}-{_time_hash}"

   # Override the `__init__` method to automatically generate the slug
   def __init__(self, **data):
       super().__init__(**data)
       if self.title:
           self.slug = self.create_slug(self.title)
```

repositories/blog.py

Create an instance method in `BlogRepository` class

```
def update_blog(self, blog_id: int, blog: BlogUpdate) → Optional[Blog]:
        """
        Update an existing blog by its ID.
        """
        db_blog = self.db.query(Blog).filter(Blog.id == blog_id).first()

        if not db_blog:
            raise HTTPException(status_code=404,detail="Blog not found!")

        if blog.title:
            db_blog.title = blog.title
            db_blog.slug = blog.slug
        if blog.content is not None:
            db_blog.content = blog.content
        if blog.is_active is not None:
            db_blog.is_active = blog.is_active

        self.db.commit()
        self.db.refresh(db_blog)
        return db_blog
```

apis/v1/blog.py

```
@router.put("/{blog_id}")
def update_blog(blog_id: int,payload: BlogUpdate, db: Session = Depends(get_db)):
    blog_repo = BlogRepository(db=db)
    blog_repo.update_blog(blog_id=blog_id, blog=payload)
    return {
        "success" : "Blog updated successfully"
    }
```

Blog CRUD (Delete)

repositories/blog.py

Create an instance method in `BlogRepository` class

```
def delete_blog(self, blog_id: int) → bool:
        """
        Delete a blog by its ID.
        """
        db_blog = self.db.query(Blog).filter(Blog.id == blog_id).first()
        if not db_blog:
            raise HTTPException(status_code=404,detail="Blog not found!")

        self.db.delete(db_blog)
        self.db.commit()
```

apis/v1/blog.py

```
@router.delete("/{blog_id}")
def delete_blog(blog_id: int,payload: BlogUpdate, db: Session = Depends(get_db)):
    blog_repo = BlogRepository(db=db)
    blog_repo.delete_blog(blog_id=blog_id, blog=payload)
    return {
        "success" : "Blog deleted successfully"
    }
```

Optimization

In the list query, we wrote the query which occurs (n+1) problem

from sqlalchemy.orm import joinedload

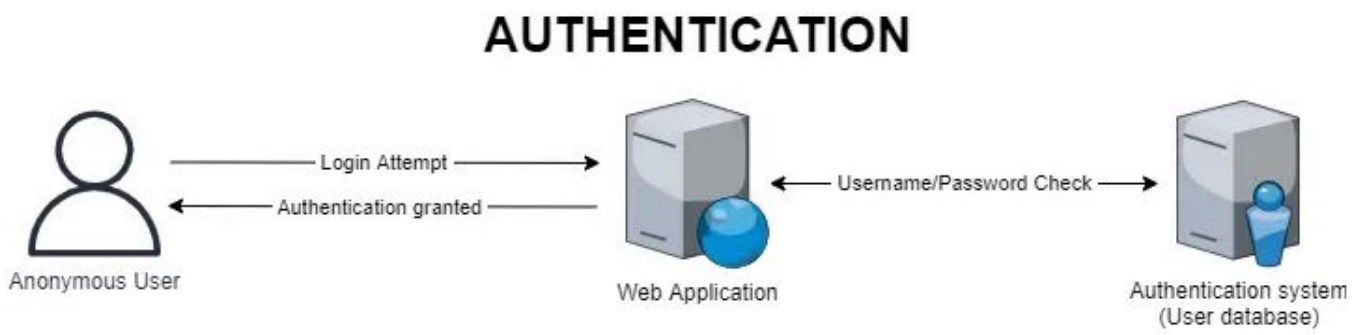
```
blogs = self.db.query(Blog).options(joinedload(Blog.author)).offset(skip).limit(limit).all()
```

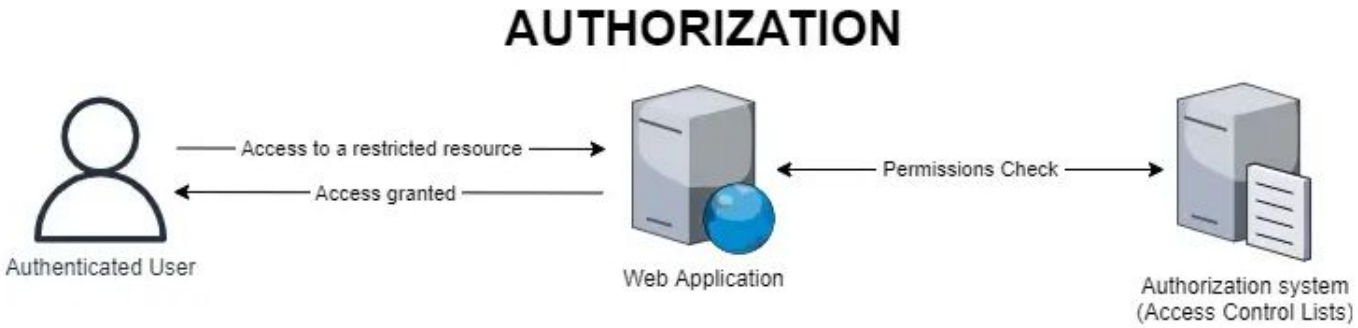
Refactor

```
db_blog = self.db.query(Blog).filter(Blog.id == blog_id).first()
if not db_blog:
    raise HTTPException(status_code=404,detail="Blog not found!")
```

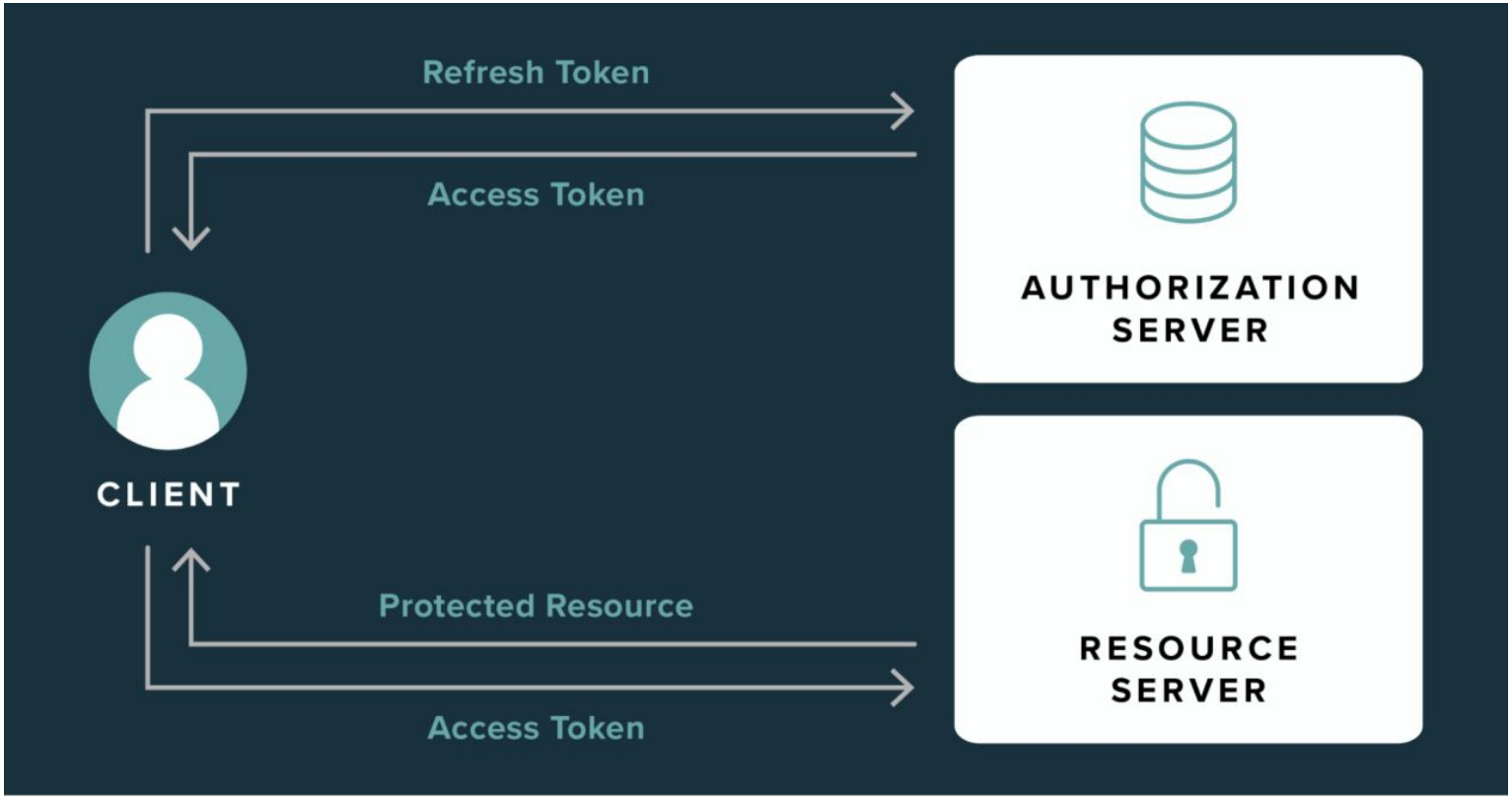
This part is common for single get, update and delete. So move this part into a separate instance method and from other functionality call this function. And obviously the new function must return the db_blog object

Authentication & Authorization





Authentication & Authorization



Authentication & Authorization

Authentication

Verifies your identity, let's you in

Happens once

Authorization

Verifies your destination, check you in

Happens repeatedly

Permit.io

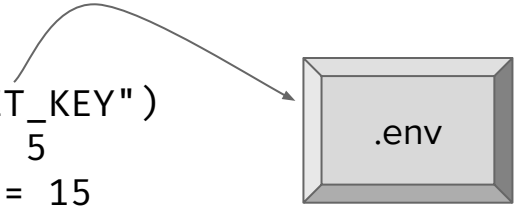
Authentication & Authorization (Implementation)

Add these to requirements.txt and install it:

- `PyJWT==2.10.1`
- `python-multipart==0.0.20`

core/config.py add these `Settings` class variable.

```
class Settings:
    # rest of the code
    SECRET_KEY: str = os.getenv("SECRET_KEY")
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 5
    REFRESH_TOKEN_EXPIRE_MINUTES: int = 15
    JWT_ALGORITHM: str = "HS256"
```



Create a new utility file under utils/ named '*jwt_manager.py*' . And write below code:

```
import jwt
from datetime import datetime, timedelta, timezone
from typing import Optional
from core.config import settings
```

```
def create_access_token(
   data: dict,
   expires_delta: Optional[timedelta] = settings.ACCESS_TOKEN_EXPIRE_MINUTES
):
   to_encode = data.copy()
   expire =  datetime.now(timezone.utc) + timedelta(minutes=expires_delta)
   to_encode.update({"exp": expire})
   return jwt.encode(to_encode, settings.SECRET_KEY, algorithm=settings.JWT_ALGORITHM)
```

```
def create_refresh_token(
   data: dict,
   expires_delta: Optional[timedelta] = settings.REFRESH_TOKEN_EXPIRE_MINUTES
):
   to_encode = data.copy()
   expire =  datetime.now(timezone.utc) + timedelta(minutes=expires_delta)
   to_encode.update({"exp": expire})
   return jwt.encode(to_encode, settings.SECRET_KEY, algorithm=settings.JWT_ALGORITHM)
```

Continue in the '*jwt_manager.py*'

```
def verify_token(token: str):
    try:
        payload = jwt.decode(
            token,
            settings.SECRET_KEY,
            algorithms=[settings.JWT_ALGORITHM]
        )
        return payload
    except jwt.ExpiredSignatureError as e:
        print(e)
        return None
    except jwt.PyJWTError as e:
        print(e)
        return None
```

Add code in the *schemas/user.py*

```
class Token(BaseModel):
   access_token: str
   refresh_token: str
   token_type: str = "bearer"
```

Add some methods in *UserRepository*

```
from fastapi.security import OAuth2PasswordBearer
from db.session import get_db
from utils.password_manager import PasswordManager

oauth2_scheme = OAuth2PasswordBearer(tokenUrl=" /users/token")
```

This '/users/token' will be our token api or endpoint.

```
def get_user_by_id(self, id: int) -> Optional[User]:
    return self.db.query(User).filter(User.id == id, User.is_active==True).first()
```

```
def get_user_for_token(self, email: str, password: str) -> Optional[User]:
    user = self.get_user_by_email(email=email)
    if not user:
        raise HTTPException(status_code=401, detail="Invalid credentials")
    is_password_matched = PasswordManager.verify_password(password, user.password)
    if not is_password_matched:
        raise HTTPException(status_code=401, detail="Invalid credentials")
    return user
```

Continue adding methods in the UserRepository

```
    @staticmethod
    def get_current_user(token: str = Depends(oauth2_scheme) , db: Session = Depends(get_db)):
        payload = verify_token(token)
        if payload is None:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token")
        user = db.query(User).filter(User.id == payload.get("sub")).first()
        if user is None:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="User not found")
        return user
```

Fastapi , Automatically call token endpoint whenever this method is being called.

Let’s add two endpoint. (/token) (/refresh). Add these to user route. (apis/v1/user.py)

```
from fastapi.security import OAuth2PasswordRequestForm
from utils.jwt_manager import create_access_token, create_refresh_token, verify_token
```

```
# Route for getting the token (login)
@router.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends(), db: Session =
Depends(get_db)):
   user = UserRepository(db=db).get_user_for_token(
       email=form_data.username, password=form_data.password
   )
   access_token = create_access_token(data={"sub": str(user.id)})
   refresh_token = create_refresh_token(data={"sub": str(user.id)})

   return {"access_token": access_token, "refresh_token": refresh_token}
```

Continue in ‘*apis/v1/user.py*’

```
@router.post("/refresh", response_model=Token)
async def refresh_access_token(refresh_token: str, db: Session = Depends(get_db)):
   payload = verify_token(refresh_token)
   if payload is None:
       raise HTTPException(status_code=401, detail="Invalid refresh token")

   payload_subject = payload.get("sub")
   user = UserRepository(db=db).get_user_by_id(id=payload_subject)
   if user is None:
       raise HTTPException(status_code=404, detail="User not found")

   access_token = create_access_token(data={"sub": str(user.id)})
   new_refresh_token = create_refresh_token(data={"sub": str(user.id)})

   return {"access_token": access_token, "refresh_token": new_refresh_token}
```

Let’s add a protected router in main.py

```
@app.get("/protected")
async def protected_route(current_user: User = Depends(UserRepository.get_current_user)):
    return {"message": f"Hello {current_user.email}, you are authorized!"}
```

So these routers has been listed under the ‘users’ tag

users

POST	/users	Create User
POST	/users/token	Login For Access Token
POST	/users/refresh	Refresh Access Token

Image Upload

Starts with main.py (to set the static file directory)

```
from fastapi.staticfiles import StaticFiles
app.mount("/static", StaticFiles(directory="uploads/images"), name="static")
```

Uploaded file would be saved in this path.
And /static router point this directory

Create a new file 'const.py' under utils/ directory. And write below:

```
UPLOAD_FOLDER = "uploads/images"
```

Don't forget to add 'uploads/' in the '.gitignore' file. No static file should not be in git

Add a new column in the User model.

```
class User(Base):
   # rest of the columns
   image_url = Column(String, nullable=True)
```

Whenever you change in the model structure, please run the migration

```
alembic revision -m 'image column added'
```

Check the generated migration file, is it correct or not. If correct , then done migration:

```
alembic upgrade head
```

Now add a router for uploading the image

```
@router.put("/upload_image")
async def upload_image(
    file: UploadFile = File( ... ),
    current_user: User = Depends(UserRepository.get_current_user),
    db: Session = Depends(get_db)
):

   # Ensure the file is an image (you can extend the validation if needed)
    if not file.content_type.startswith('image'):
        raise HTTPException(status_code=400, detail="Uploaded file
is not an image")

    user_repo = UserRepository(db=db)

    # If the user already has an image, remove the old image
    old_image_path = current_user.image_url

    if old_image_path:
        user_repo.remove_previous_image(old_image_path)

  # More code in this function next slide
```

We will create `remove_previous_image` method in the UserRepository class.

Rest of the router code.

```
# Generate a unique filename for the new image
unique_filename = f"{uuid4()}_{file.filename}"
new_file_path = os.path.join(UPLOAD_FOLDER, unique_filename)

# Save the new image file to the server
with open(new_file_path, "wb") as buffer:
    buffer.write(await file.read())

user_repo.save_image_path_to_db(user=current_user, new_image_path=unique_filename)

return {
    "success" : "Successfully uploaded image!",
    "path" : f"/static/{unique_filename}"
}
```

We will create `save_image_path_to_db` method in the UserRepository class.

Add two new methods in the UserRepository

```
def remove_previous_image(self, old_image_path):
        # Remove the old image from the file system if it exists
        try:
            # Convert the image_url to the actual file path
            old_image_file_path = os.path.join(UPLOAD_FOLDER, old_image_path)
            if os.path.exists(old_image_file_path):
                os.remove(old_image_file_path)
        except Exception as e:
            print(e)
            return False
        return True

def save_image_path_to_db(self, user: User, new_image_path:str):
        user.image_url = new_image_path
        try:
            self.db.commit()
        except:
            self.db.rollback()
```

Now test the router from swagger.

User Profile Router

We will create a router for the current user profile. Add UserProfileView schema.

```
class UserProfileView(BaseModel):
    id: int
    email : str
    is_active : bool
    image_url : Optional[str] = None # Optional for the users who doesn't upload image

    @validator('image_url')
    def add_static_prefix(cls, v):
        # Add '/static/' to the filename if not already present
        return f"/static/{v}" if not v.startswith("/static") else v

    class Config(): #tells pydantic to convert even non dict obj to json
        orm_mode = True
```

The router itself is below:

```
@router.get("/profile", response_model=UserProfileView)
async def get_current_user(current_user: User = Depends(UserRepository.get_current_user)):
   return current_user
```

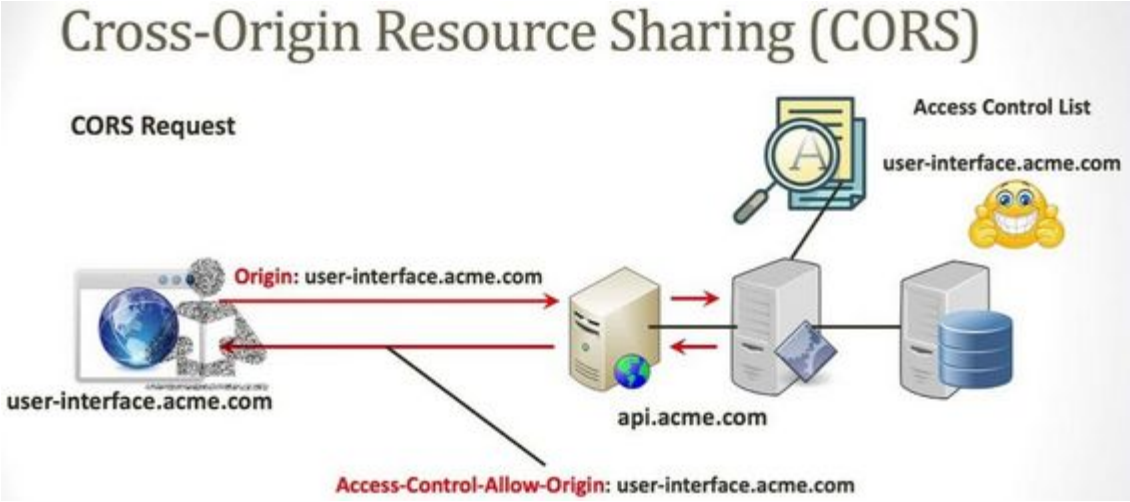
Could now make these routers protected ?

- [POST] /blog (get current user and save it as author_id)
- [PUT] /blog (check if the requested user is the author of the blog or not)
- [DELETE] /blog (check if the requested user is the author of the blog or not)

Fastapi way to get environment variables

```
#core/config.py
from pydantic import BaseSettings

class Settings(BaseSettings):
   PROJECT_NAME:str = "Innovative 🚀"
   PROJECT_VERSION: str = "1.0.0"

   POSTGRES_USER: str
   POSTGRES_PASSWORD: str
   POSTGRES_SERVER: str
   POSTGRES_PORT: str
   POSTGRES_DB: str

   SECRET_KEY: str
   ACCESS_TOKEN_EXPIRE_MINUTES: int = 5
   REFRESH_TOKEN_EXPIRE_MINUTES: int = 15
   JWT_ALGORITHM: str = "HS256"

   @property
   def DATABASE_URL(self) → str:
       return f"postgresql://{self.POSTGRES_USER}:{self.POSTGRES_PASSWORD}@{self.POSTGRES_SERVER}:{self.POSTGRES_PORT}/{self.POSTGRES_DB}"

   class Config:
       env_file = ".env"

settings = Settings()
```

CORS


Cross-Origin Resource Sharing (CORS)
CORS Request
Access Control List
user-interface.acme.com
Origin: user-interface.acme.com
user-interface.acme.com
api.acme.com
Access-Control-Allow-Origin: user-interface.acme.com

CORS setup

Add below code in the ‘main.py’ file

```
from fastapi.middleware.cors import CORSMiddleware

# List of allowed origins, you can add specific domains or '*' for all origins
origins = [
   "http://localhost:3000",  # Frontend during development
   "https://your-frontend-domain.com",  # Production frontend
   "*",  # Allow all origins (use cautiously)
]

# Adding CORSMiddleware to the FastAPI app
app.add_middleware(
   CORSMiddleware,
   allow_origins=origins,  # List of allowed origins
   allow_credentials=True,  # Allow cookies and credentials to be sent
   allow_methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"],  # Allowed HTTP methods
   allow_headers=["*"],  # Allowed headers
)
```