Complete RESTful API’s

PyDantic V1 vs PyDanticv2

* New information in this will include
  + FULL SQL DB
  + Authentication
  + Authorization
  + Hashing Passwords
* Creating a TODO Table
  + We will create new todo table model for our application.
  + We will be using these todos to save records throughout this project

Section 8: Setup Database

What is Database?

Organized collection of structured information of data, which is stored in a computer system.

1. The data can be easily accessed
2. The data can be modified
3. The data can be controlled and organized
4. Many databases use a structured query language (SQL) to modify or write data
5. Data can be related to about any object
6. For example, a user on application may have
   1. Name
   2. Image
   3. Email
   4. Password

* A DB is a collection of data since data, on its own, its just data a database allows management of this data
* Databases are organized in how data can be retrieved, stored and modified.
* There are many types of database management system’s
  + Sqlite
  + SQL
  + PosgreSQL

What is SQL?

* Pronounced as either as S-Q-L or See Quel
* Structured language for dealing with RDBMS
* SQL can be used to do different things with database records such as
  + - Create
    - Read
    - Update
    - Delete

DB Connection with ORM SQLAlchemy

* We need to pip install sqlalchemy for dealing with databases.

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from sqlalchemy.ext.declarative import declarative\_base

SQLALCHEMY\_DB\_URL = 'sqlite:///./bikes.db'

engine = create\_engine(SQLALCHEMY\_DB\_URL, connect\_args={'check\_same\_thread' : False})

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

Base = declarative\_base()

The above code is the boiler plate code for defining a Database (name the file database.py).

Database tables (Models)

from database import Base

from sqlalchemy import Column, Integer, String, Boolean

class Todos(Base):

    \_\_tablename\_\_ = 'bikes'

    id = Column(Integer, primary\_key=True, index=True)

    title = Column(String)

    description= Column(String)

    priority= Column(Integer)

    complete= Column(Boolean, default=False)

Create DB Connection For API

Create a file named main.py

from fastapi import FastAPI

import models

from database import engine

app = FastAPI()

models.Base.metadata.create\_all(bind=engine)

Installation of SQLite 3

* Download Zip file of precompiled binaries for windows from sqlite.org
* Extract the files.
  + Add to the system variable from environment variable
  + Run sqlite3 in cmd to check the DB.
  + Sqlite3 DB\_NAME.DB

Now the lesson is about SQL queries so we will skip that in this document.

To view formatted data in sqlite3 use the below commands.

* .mode Column
* .mode markdown
* .mode box
* .mode table

API Request Methods.

we have to create DB dependency inside main.py

def get\_db():  
 db = SessionLocal()  
 try:  
 yield db  
 finally:  
 db.close()  
  
  
db\_dependency = Annotated[Session, Depends(get\_db)]

Annotated is from typing

Session if from sqlalchemy.orm

Depends is from fastapi

Getting TODO by Id?

When we first run the server by writing the basic code and giving the database file name in our code for our example bikes.db

A DB file is created now we create a data using sql commands in Sqlite

The commands are

Sqlite3 bikes.db

.schema #this command will show you the schema

Then the insert command.

INSERT INTO bikes (id, bikename, cubic\_capacity, mileage, owned) values (2, 'Apache RTR 160', 160, 40, 0);

#API REQUEST METHODS

GET ALL BIKES

We will add the below code to the main.py and we get an API endpoint which will add the functionality of reading all bikes

db\_dependency = Annotated[Session, Depends(get\_db)]

@app.get("/")

async def read\_all\_bikes(db: db\_dependency):

    return db.query(Bikes).all()

GET BIKE WITH ID

@app.get("/bike/{bike\_id}")

async def get\_bike\_with\_id(db: db\_dependency, bike\_id: int):

    bike = db.query(Bikes).filter(bike\_id == Bikes.id).first()

    if bike is not None:

        return bike

    raise HTTPException(status\_code=404, detail="Bike Not Found")

HTTPException is from fastapi package don’t import it from http.client

* Now we will add custom status codes and apply some checks

@app.get("/bike/{bike\_id}", status\_code=status.HTTP\_200\_OK)

async def get\_bike\_with\_id(db: db\_dependency, bike\_id: int):

    bike = db.query(Bikes).filter(bike\_id == Bikes.id).first()

    if bike is not None:

        return bike

    raise HTTPException(status\_code=404, detail="Bike Not Found")

#POST REQUEST SQLITE 3

We will create a request class for our API using BaseModel that is from pydantic in order to manage dataytpes of class parameters, for our case for bikes we will make BikeRequest.

And this class is not in models.py

class BikeRequest(BaseModel):

    #id is not needed in this class definition

    bikename: str = Field(min\_length=5, max\_length=100) #Field is from pydantic

    cubic\_capacity: int = Field(gt=90, lt=1000)

    mileage: int = Field(lt=80, gt =5)

    owned: bool

this is how we should create a request class and define the types using the functionality of BaseModel present in pydantic

@app.post("/bike", status\_code=status.HTTP\_201\_CREATED)

async def create\_bike(db: db\_dependency, bike\_request: BikeRequest):

    # bike\_model  = Bikes(\*\*bike\_request.dict()) #deprecated

    bike\_model  = Bikes(\*\*bike\_request.model\_dump())

    db.add(bike\_model)

    db.commit()

and this is how we make the post method.

#PUT REQUEST SQLITE

Updating a bike enhancing a bike.

@app.put("/bike/{bike\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT)

async def update\_a\_bike(db: db\_dependency,

                         bike\_request: BikeRequest,

                         bike\_id: int = Path(gt=0)): #Path is from fastapi

    bike\_model = db.query(Bikes).filter(Bikes.id == bike\_id).first()

    if bike\_model is None:

        raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND,

                             detail="Bike id not found")

    bike\_model.bikename = bike\_request.bikename

    bike\_model.cubic\_capacity = bike\_request.cubic\_capacity

    bike\_model.mileage = bike\_request.mileage

    bike\_model.owned = bike\_request.owned

    db.add(bike\_model)

    db.commit()

#DELETE REQUEST in SQLITE

Delete request in fastAPI for sqlite database can be written as

@app.delete("/bike/{bike\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT)

async def delete\_a\_bike(db: db\_dependency, bike\_id: int = Path(gt=0)):

    bike\_model = db.query(Bikes).filter(Bikes.id == bike\_id).first()

    if bike\_model is None:

        raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND,

                            details = "Not Found")

    db.query(Bikes).filter(Bikes.id == bike\_id).delete()

    db.commit()

AUTHENTICATION AND AUTHORIZATION

* We will create auth.py inside project folder -> routers folder
* Create basic fastapi setup
* Now we will create endpoints

from fastapi  import FastAPI

app = FastAPI

@app.get("/auth/")

async def get\_user():

    return {

        "user": "authenticated"

    }

Note: now we will create routes for our API’s from this auth file.

* Import API router from fastapi in auth.py and remove import of auth.py
* Create an object of APIRouter() as router

Router = APIRouter()

Now @app.get() 🡺 @router.get()

Now in main.py

Below the mentioned line

models.Base.metadata.create\_all(bind=engine)

add this line

app.include\_router(auth.router) #from routers import auth

now the auth.py API’s are now available when you hit enter

uvicorn main:app --reload

#Enhance Application for Scalability

add new file in routers folder namely bikes.py for your API

Import APIRouter

Create router object

Copy everything from main.py to bikes.py

Change

app 🡺 router

FastAPI 🡺 APIRouter

Remove below mentioned lines

models.Base.metadata.create\_all(bind=engine)

app.include\_router(auth.router) #from routers import auth

now in main.py from routers import bikes

now add bikes router below auth router as shown

app.include\_router(auth.router)

app.include\_router(bikes.router)

now delete everything below it

remove all the non-needed imports

main.py will now look like this.

from fastapi import FastAPI

import models

from database import engine, SessionLocal

from routers import auth, bikes

app = FastAPI()

models.Base.metadata.create\_all(bind=engine)

app.include\_router(auth.router) #from routers import auth

app.include\_router(bikes.router)

#One to Many Relationship

What is one to many relationship?

A user can have multiple bikes

Similarly, we can have multiple users in our database, so we have 2 different tables **Users** and **Bikes**

Owner\_id can be foreign key

#Foreign Key’s

* A foreign key [FK] is a column within a relational database table that provides a link between two separate tables.
* A foreign key references a primary key of another table
* Most relational DB need foreign keys to be able to link tables together to preset data

How to Reference a user using Bike?

* Each API request a user will have their ID attached
* if we have the user id attached to each request, we can use their id to find their bike’s
* e.g. SELECT \* FROM BIKES WHERE OWNER\_ID = 2;

User Table Creation

In database.py change bikes.db to bikesapp.db

NOTE: *Sqlalchemy can not enhance a table for us it can only create a table for us*

Delete bikes.db file (as we are designing the db again according to our changes we will create 2 tables in the db file one is user and other is bikes.)

Now in models.py add the class for users as shown below as per your need.

class Users(Base):

    \_\_tablename\_\_ ='users'

    id  = Column(Integer, primary\_key=True)

    email  = Column(String, unique=True)

    username = Column(String, unique =True)

    firstname = Column(String)

    lastname = Column(String)

    hashed\_password = Column(String)

    is\_active = Column(Boolean, default=True)

    role = Column(String)

#Create Users

Now we will write an API to create a user inside auth.py file

Either you can remove the entire code and write the below one or just write the required code.

from fastapi import APIRouter

from pydantic import BaseModel

from models import Users

router = APIRouter()

class CreateUserRequest(BaseModel):

    email: str

    username: str

    first\_name: str

    last\_name: str

    password: str #different from schema there it is hashed\_password

    # is\_active: str "is it is in db then it is active"

    role: str

@router.post("/auth/")

async def create\_user(create\_user\_request: CreateUserRequest):

    create\_user\_model  = Users(

        email = create\_user\_request.email,

        username = create\_user\_request.username,

        firstname = create\_user\_request.first\_name,

        lastname = create\_user\_request.last\_name,

        hashed\_password = create\_user\_request.password,

        role = create\_user\_request.role,

        is\_active = True

    )

    return create\_user\_model

#Hash User’s Passwords

To hash password, we need to install 2 dependencies passlib and bcrypt==4.0.1

Import the below piece of code

from passlib.context import CryptContext

create an object of the same as shown below

bcrypt\_context = CryptContext(schemes=['bcrypt'], deprecated = 'auto')

now change the post method to make the password stored I hashed form.

@router.post("/auth/")

async def create\_user(create\_user\_request: CreateUserRequest):

    create\_user\_model  = Users(

        email = create\_user\_request.email,

        username = create\_user\_request.username,

        firstname = create\_user\_request.first\_name,

        lastname = create\_user\_request.last\_name,

        # hashed\_password = create\_user\_request.password,

        hashed\_password = bcrypt\_context.hash(create\_user\_request.password),

        role = create\_user\_request.role,

        is\_active = True

    )

    return create\_user\_model

now the password will be stored in hashed form.

{

"email": "xyz@gmail.com",

"username": "jd\_user",

"first\_name": "john",

"last\_name": "doe",

"password": "pass",

"role": "admin"

}

For this body we get the below response



#SAVE USER TO DATABASE

Now we will add get\_db functionality and db\_dependency to the auth.py file.

def get\_db():

    db = SessionLocal() #from database.py file

    try:

        yield db

    finally:

        db.close()

db\_dependency = Annotated[Session, Depends(get\_db)]

# Annotated is from typing

# Session is from sqlalchemy.orm

# Depend is from fastapi

Now we have to accept db dependency in the post method and add 2 lines in the end of the post function

@router.post("/auth/")

async def create\_user(db: db\_dependency,create\_user\_request: CreateUserRequest):

    create\_user\_model  = Users(

        email = create\_user\_request.email,

        username = create\_user\_request.username,

        firstname = create\_user\_request.first\_name,

        lastname = create\_user\_request.last\_name,

        # hashed\_password = create\_user\_request.password,

        hashed\_password = bcrypt\_context.hash(create\_user\_request.password),

        role = create\_user\_request.role,

        is\_active = True

    )

    db.add(create\_user\_model)

    db.commit()

#AUTHENTICATE A USER

We will create a post method at endpoint named “/token”

* we need to install python package named python-multipart
* from fastapi.security import OAuth2PasswordRequestForm

now import the below dependency

from fastapi.security import OAuth2PasswordRequestForm

now we will add authenticate\_user() function in auth.py

def authenticate\_user(username: str, password: str, db):

    user = db.query(Users).filter(Users.username == username).first()

    if not user:

        return False

    if not bcrypt\_context.verify(password, user.hashed\_password):

        return False

    return True

the post method at endpoint “/token” will look like

@router.post("/token")

async def login\_for\_access\_token(form\_data: Annotated[OAuth2PasswordRequestForm,Depends()], db: db\_dependency):

    user = authenticate\_user(form\_data.username, form\_data.password, db)

    if not user:

        return "Authentication Failed"

    return "Successfully Authenticated"

now run the server and check on “/token”

it will only Print “successfully authenticated” if the credentials are true.

#JSON WEB TOKEN

* JSON Web token is a self-contained way to securely transmit data and information b/w 2 parties using JSON Object.
* JSON web tokens can be trusted because each JWT can be digitally signed which in return allows the server to know if the JWT has been changed at all
* JSON Web token should be used when dealing with authorization
* JWT is a great way of information to be exchanged b/w server and client

JWT STRUCTURE

A JWT is created of 3 Separate parts separated by dots (.) which include

* Header: (a)
* Payload: (b)
* Signature: (c)

Aaaaa.bbbbb.ccccc

JWT HEADER

* A JWT header usually consists of 2 parts
  + (alg) the algorithm for signing
  + “typ” the specific type for token
* The JWT token then encoded using Base64 to create the first part of the JWT (a)

{

    "alg": "HS256",

    "typ": "JWT"

}

JWT PAYLOAD

* A JWT Payload consists of the data. The payload data contains claims and there are 3 different types of claims
  + Registered
  + Private public

{

    "sub" : "1234567890",

    "name" : "Rahul Sharma",

    "given\_name" : "Rahul",

    "family\_name" : "Sharma",

    "email" : "xyz@gmail.com",

    "admin" : True

}

* The JWT payload is then encoded using Base64 to create the second part of the JWT (b)

JWT SIGNATURE

* A JWT signature is created by using the algorithm in the header to hash out the encoded header, encoded payload with a secret
* The secret can be anything but is saved somewhere on the server that the client does not have access to
* The signature is third and final part of the JWT (c)

HMACSHA(

    base64UrlEncode(header) + "." +

    base64UrlEncode(paylaod),

    secret

)

*JWT example:*

0481e35d7e070c0b8698.f2cd39c33e449af7e54b.d81f31ba301d99a093c6

#Encode a JWT

* First we need to install a library “python-jose[cryptography]”
* In terminal write openssl rand -hex 32, now you will get a random hexadecimal string
* In auth.py make a variable SECRET\_KEY and assign it with the hexadecimal string that you get from openssl command.
* Just below the string make a variable named “ALGORITHM” and assign it with ‘HS256’
* Now create a function named create\_access\_token()
* Import below mentioned dependencies

from jose import jwt

from datetime import timedelta

now create a function named create\_access\_token()

def create\_access\_token(username: str, user\_id: int, expires\_delta: timedelta): #from datetime

    encode = {'sub':username, 'id': user\_id}

    expires = datetime.now(timezone.utc) + expires\_delta #from datetime

    encode.update({'exp': int(expires.timestamp())}) #from datetime

    return jwt.encode(encode, SECRET\_KEY, algorithm=ALGORITHM)

now modify the login for access token method at endpoint “/token”

@router.post("/token")

async def login\_for\_access\_token(form\_data: Annotated[OAuth2PasswordRequestForm,Depends()], db: db\_dependency):

    user = authenticate\_user(form\_data.username, form\_data.password, db)

    if not user:

        return "Authentication Failed"

    token = create\_access\_token(user.username, user.id, timedelta(minutes=20))

    return {

        'access\_token' : token,

        'token\_type' : 'bearer'

    }

And we have to change one more thing which is in authenticate user function instead of returning true we have to return user

def authenticate\_user(username: str, password: str, db):

    user = db.query(Users).filter(Users.username == username).first()

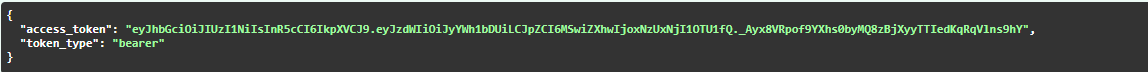
    if not user:

        return False

    if not bcrypt\_context.verify(password, user.hashed\_password):

        return False

    return user



Now you will get a response like this.

#Decode a JSON Web token.

* Import OAuth2PasswordBearer from fastapi.security
* Import JWTError from jose
* Add dependency

Oauth2\_bearer = OAuth2Bearer(tokenUrl=‘token’)

* Below create access\_token() add a function named get\_current\_user()

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_bearer)]):

    try:

        payload = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])

        username: str = payload.get('sub')

        user\_id : int = payload.get('id')

        if username is None or user\_id is None:

            raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="could not validate user")

        return {'username': username, 'id': user\_id}

    except JWTError:

        HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid Token")

The instructor didn’t run this now but we will use this in future.

#Authentication Enhancement

router = APIRouter() 🡺 router= APIRouter(prefix = “/auth”, tags=[“auth”])

oauth2\_bearer = OAuth2PasswordBearer(tokenUrl= ”token”) 🡺

oauth2\_bearer 🡺 OAuth2PasswordBearer(tokenUrl = “auth/token”)