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()

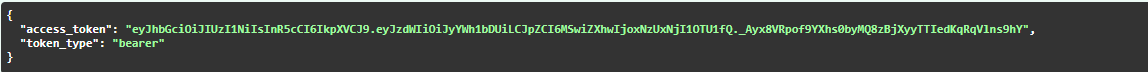
    if not user:

        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”)

SECTION 11: Authenticate requests

1. Post Todo with user authentication (User ID)

First of all you need to create a user dependency in your application file for e.g. bikes.py in your case

Now what we want is whenever we creat a bike request/ create a bike so the user will be authenticated first for that reason we will use the function get\_current\_user() from auth.py file.

So we import

from routers.auth import get\_current\_user

and now we add the dependency as shown below

user\_dependency = Annotated[dict, Depends(get\_current\_user)]

now we will modify the POST bike request as shown below.

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

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

    if not user:

        raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Unauthorized User")

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

    bike\_model  = Bikes(\*\*bike\_request.model\_dump(), owner\_id = user.get('id'))

    db.add(bike\_model)

    db.commit()

Note: here you can see that we are using owner\_id as foreign key from user class for that we have to delete the db file and change the class files in models.py and make the changes in other areas where these are used such as request class (BikeRequest)

Here you can check the models.py file for reference

from database import Base

from sqlalchemy import Column, ForeignKey, Integer, String, Boolean

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)

class Bikes(Base):

    \_\_tablename\_\_ = 'bikes'

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

    bikename = Column(String)

    cubic\_capacity= Column(Integer)

    mileage= Column(Integer)

    owner\_id = Column(Integer, ForeignKey('users.id'))

here you can see how we have to change the BikeRequest class

class BikeRequest(BaseModel):

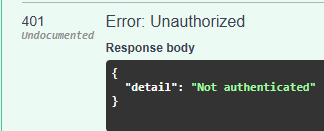
    #id is not needed in this class definition

    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)

Now if you create a request (create bike) using swagger or postman you will see an error



Now you have to authorize the user first

By clicking on the lock icon in the below request



1. Get All Todos(User ID)

GET ALL TODOS (Bikes for our case) BY USER

For this case you just need to modify the read\_all\_bikes like the below one.

@router.get("/")

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

    return db.query(Bikes).filter(Bikes.owner\_id == user.get('id')).all()

1. Get Todo (ID + User ID)

Get todo by ID with authentication

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

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

    bike = db.query(Bikes).filter(bike\_id == Bikes.id).filter(Bikes.owner\_id == user.get('id')).first()

    if bike is not None:

        return bike

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

1. Put Todo (user Id)

UPDATE A TODO WITH AUTHENTICATION

*Note: Remember to Login with correct credentials*

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

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

                         bike\_request: BikeRequest,

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

    if user is None:

        raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Unauthorized user")

    bike\_model = db.query(Bikes).filter(Bikes.id == bike\_id).filter(Bikes.owner\_id == user.get('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.owner\_id = user.get('id')

    db.add(bike\_model)

    db.commit()

1. Delete Todo (User Id)

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

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

    if user is None:

        raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="User Not authenticated")

    bike\_model = db.query(Bikes).filter(Bikes.id == bike\_id).filter(Bikes.owner\_id == user.get('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).filter(Bikes.owner\_id == user.get('id')).delete()

    db.commit()

1. Admin Router

Admin Route for user Role.

We are going to make a route for admin where only admin is going to run these endpoints

We are now add role to the create\_access\_token()

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

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

    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)

Then we also pass it in login\_for\_access\_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, user.role timedelta(minutes=20))

    return {

        'access\_token': token,

        'token\_type': 'bearer'

    }

We also decode ‘role’ in 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')

        user\_role: str = payload.get('role')

        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, 'user\_role': user\_role}

    except JWTError:

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

Create a new file in routers directory named it admin.py

From Bikes.py copy all imports, dependencies and get\_db()

Now in admin.py give prefix and tags of admin in APIRouter()

Now we write a get method in admin.py

@router.get("/bike", status\_code=status.HTTP\_204\_NO\_CONTENT)

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

    if user is None or user.get('role') != 'admin':

        raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="User not validated.")

    return db.query(Bikes).all()

Now in main.py we have to add the route of admin

app.include\_router(admin.router)

Now we add a delete request in admin.py

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

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

    if user is None or user.get('role') != 'admin':

        raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="User not validated.")

    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 Not Found to Delete")

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

    db.commit()

now you can check that only the admin has rights to check every Bikes but the user can check only their bike

and even admin can delete every bike from the database.

1. Assignment Problem

Section 12: Large Production Database Setup.

1. Production Database
   1. This section will go over installing a production relational database for your application.
   2. The 2 DBMS applications we will be going over is MySQL and PosgreSQL
   3. Both DBMS Systems are used widely throughout enterprise applications and you can not go wrong with either one.
2. Production DBMS VS SQLite3
   1. Sqlite3 strives to provide local data storage for individual applications and devices.
   2. Sqlite3 emphasizes economy, efficiency and simplicity.
   3. For most small/medium applications sqlite3 will work perfectly.
   4. Sqlite3 focuses on different concepts than a production database management system.
   5. MySQL and PosgreSQL focuses on big difference compared to sqlite3
   6. These production DBMS focuses on Scalability, concurrency and control
   7. If your application is going to have 10’s of thousands of users, it may be wise to switch to a production database.
   8. If only you and few others are going to be user of your application then SQLite3 will work great.
3. PRODUCTION DBMS KEY NOTES
   1. Sqlite3 runs in-memory or local disk which allows development of a sqlite3 data to be easy, as it is a part of your application.
   2. Production DBMS runs on their own server and port which means you need to make sure that the database is running and have authentication linking to DBMS
   3. (SQLite3) for deployment you can deploy a sqlite3 database along with the application.
   4. (Prod DBMS) for deployment you will need to also deploy the database separate from the application.
4. OVERVIEW OF SECTION
   1. We will install production DBMS
   2. Setup the tables and data within the production DBMS
   3. Connect the production DBMS to our application
   4. Push data from application to our production DBMS
   5. CRUD Operations on Prod DBMS

POSTGRESQL

1. PRODUCTION DB INTRO(PostgreSQL)
   1. Production ready
   2. Open source RDBMS
   3. Secure
   4. Requires a server
   5. Scalable
2. WHAT WE WILL COVER
   1. Install the setup
   2. Setup SQL tables
   3. Connect postgresql to our application.
3. CREATE DATABASE TABLES
   1. We create our application server in server group
   2. We create our application database in database panel.
   3. Now we run SQL queries to create our table using query tool.

Modifying the database.py file as per the new (postgres SQL) requirements.

First of all, we need to install a pip package in order to deal with PostgreSQL



Now modify the database.py file as shown below.

from sqlalchemy import create\_engine  
from sqlalchemy.orm import sessionmaker  
from sqlalchemy.ext.declarative import declarative\_base  
  
SQLALCHEMY\_DATABASE\_URL = 'postgresql://postgres:password@localhost/TodoApplicationDatabase'  
  
engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)  
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)  
  
Base = declarative\_base()

MySQL DATABASE

1. ­MySQL Introduction:
   1. Open-Source relational database management system
   2. Requires a server
   3. Production ready
   4. Scalable
   5. Secure
2. Who USES/USED
   1. Facebook
   2. YouTube
   3. Tesla
3. Installation of MySQL
4. Create Tables in MySQL.
   1. We will create the tables using the command of SQL for reference file is also attached in course resources
5. Connect FastAPI to MySQL
   1. First of all we need to install a dependency using PIP (pymysql)
   2. pip install pymysql
   3. cd
   4. cd
   5. cd
   6. cd
   7. cd

ALEMBIC DATA MIGRATION

What is Alembic?

* Light weight data migration tool for when using SQLAlchemy
* Migration tool allows us to plan, transfer and upgrade resources within databases.
* Alembic allows you to change a SQLAlchemy database table after it has been created
* Currently SQLALchemy will only create new database tables for us, not enhance any.

How does alembic work?

* Alembic provide the creation and invocation of change management script.
* This allows you to be able to create migration environments and be able to change data how you like
* In this section we will learn how we can use alembic for our project.
* We already have some data within our database.
* Lets take a look at our current users table
* Create a new column on our data that already exists

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Email | First\_name | ……. | Phone\_number |
| 1 | [xyz@gmail.com](mailto:xyz@gmail.com) | Rahul | <Other Columns> | 91 739277493 |

* Alembic is a powerful migration tool that allows us to modify our database schemes
* As our application evolves, our database will need to evolve as well
* Alembic helps us be able to keep modifying our database to keep us with rapid development requirements
* We will be using alembic on tables that already have data. This allows us to be able to continually create additional content that works within our application.

**ALEMBIC INTRODUCTION:**

* We need to install Alembic into our project
* Pip install alembic

|  |  |
| --- | --- |
| alembic init <folder name> | Initializes a new, generic environment |
| alembic revision -m <message> | Creates a new revision of the environment |
| alembic upgrade <revision #> | Run our upgrade migration to our database |
| alembic downgrade -1 | Run our downgrade migration to our database |

How does alembic work?

* After we initialize our project with alembic, two new items will appear in our directory
  + alembic.ini
  + alembic directory
* these are created automatically by alembic so we can upgrade, downgrade, and keep data integrity of our application

**Alembic.ini file**

* Has all environment properties for alembic
* Holds all revisions of your application
* Where you can call the migrations for upgrading
* Where you can call the migrations for downgrading.

**Alembic installation and setup**

* pip install alembic
* alembic init <alembic name> 🡺 alembic init alembic
* now you will one file named alembic.ini will be created also an directory name alembic is created.

**Alembic Revision**

* Alembic revision is how we create a new alembic file where we can add some type of database upgrade.
* When we run |alembic revision -m “create phone number col in users table” |
* Creates a new file where we can write the upgrade code.
* Each new revision will have a revision id.

Alembic Upgrade?

* Alembic upgrade is how we actually run the migration
* Enhances our database to now have a new column within our users table called ‘phone\_number’
* Previous data within database does not change.
* To run the upgrade migration
* Alembic upgrade <revision\_id>
* This will successfully implement the change within the upgrade functionality

Alembic Downgrade

* Alembic downgrade is how we revert a migration

def downgrade() -> None:  
 op.drop\_column('users', 'phone\_number')

* Reverts our database to remove the last migration change
* Previous data within database does not change unless it was on the column ‘phone\_number’ because we deleted it
* To run the downgrade migration
* alembic downgrade -1

**Alembic Revision Upgrade**

* first of all we head into alembic.ini file and give the path of our db to the sqlalchemy.url variable without quotes
* now we head into env.py inside our alembic directory
* import models
* we remove the if condition from logging functionality
* change targer\_metadata = None 🡺 targer\_metadata = models.Base.metadata
* command | alembic revision -m “Create Phone Number for user column” |
* now new file will be created inside alembic => versions => new\_created\_file.py

def upgrade() -> None:  
 *"""Upgrade schema."""* op.add\_column('users', sa.Column('phone\_number', sa.String(length=20), nullable=True))

* we will modify the upgrade function in this manner.
* Now we run command alembic upgrade <revision\_id> revision id will be found inside the newly created py file that is created after alembic revision command.
* Now we can check that the ‘phone\_number’ field is available in our database but in order to get that field in our API we need to modify the models.py file. By just adding the ‘phone\_number’ in the users models as shown below.

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)  
 phone\_number = Column(String)

**Alembic Revision Downgrade:**

* to downgrade the database, you just need to modify the downgrade function in file under version folder as shown below.

def downgrade() -> None:  
 *"""Downgrade schema."""* op.drop\_column('users', 'phone\_number')

and now run the command *alembic downgrade -1*

* now if you inquire on users table you will see that phone\_number column is now deleted in order to but to make the API work as intended you need to modify the Users Class in models.py file.
* Now you might be thinking of how you should create a table? instead of a column inside a table using alembic
* Yes you can create a table using alembic for that you need to add the Class file of new table in models.py (if you are using MySql then you have to define the length of each field.)

class Product(Base):  
 \_\_tablename\_\_ = 'products'  
  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String(100), nullable=False)  
 price = Column(Integer, nullable=False)

* Now run the command |alembic revision --autogenerate -m "Create products table"|

This will generate a file under alembic/versions/ with something like:

def upgrade():  
 op.create\_table(  
 'products',  
 sa.Column('id', sa.Integer(), nullable=False),  
 sa.Column('name', sa.String(length=100), nullable=False),  
 sa.Column('price', sa.Integer(), nullable=False),  
 sa.PrimaryKeyConstraint('id')  
 )

* Now apply the migration to the database using command |alembic upgrade head|

Notes:

1. If you **don’t use** autogenerate, you can also manually write op.create\_table(...) in your migration file.
2. Alembic doesn’t manage models directly — it only compares what's in your Base.metadata to the current DB state.
3. If you're using **MySQL**, always specify lengths for String() fields

**ASSIGNMENT.**

* Add a phone number field as required when we create a new user within our auth.py file
* Create a new @put request in our users.py file that allows a user to update their phone number.

Ans:

In auth.py

class CreateUserRequest(BaseModel):  
 email: str  
 username: str  
 first\_name: str  
 last\_name: str  
 password: str  
 role: str  
 phone\_number: str

and

@router.post("/")  
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=bcrypt\_context.hash(create\_user\_request.password),  
 role=create\_user\_request.role,  
 is\_active= True,  
 phone\_number=create\_user\_request.phone\_number,  
 )  
 db.add(create\_user\_model)  
 db.commit()

Section 14 Unit and Integration testing.

**Testing Overview**

What is testing?

* It is way for us to make sure our application is working as intended.
* Part of software development lifecycle that aims to identify:
  + Bugs
  + Errors
  + Defects
* Meet User requirement and specifications.
* Ensure software is high quality, reliable, secure and user friendly

Manual testing:

It means we run the API’s and manually test whether they are working as intended or not

Unit testing:

* Involves testing individual components or units of software in isolation from the rest of the application
* Validates that each unit of the software performs as designed.
* Unit = Testable part of the application
* Developers write unit test’s during the development phase.
* Tests are automated and are executed by testing framework (Pytest)
* Benefit is to identify bugs early in the development process.

Integration Testing:

* Focusing on testing the interaction between different units or components of the peace of software
* The scope is broader than unit testing, as we are testing multiple units together.
* Helps identify problems for the entire solution
* Example: call an API endpoint and make sure correct solution is returned.

Pytest:

* Popular testing framework for python
* Known for simplicity, Scalability and ability to handle both unit and integration tests.
* To reasons to use pytest:
  + Simple and flexible – Native Assertions
  + Fixtures – Feature setup and teardown
  + Parameterized testing – Run same test with different data

**Getting started with testing overview.**

First Step

* Create a new directory on our project called ‘test’.
* Inside our test directory create a new file called: “\_\_init\_\_.py”
* Inside our test directory create a new file called: “test\_example.py”

test\_example.py:

* Pytest will run all tests automatically that sit within files that have the name ‘test’ in them.
* For our demo, all test will be in test\_example.py so pytest can find them easily.
* When we write tests for our application, we will create new tests from a new file that matched naming conventions of our project.
* Example: todos.py will be tested by test\_todos.py

Create our first unit test:

* Write our first assertion test.
  + Assertion = Statement that checks if a condition is true.
  + If condition is true = test passes.
  + If condition is false = test fails.

test\_example.py

def test\_equal\_or\_not\_equal():

    assert 3 == 3

in terminal write pytest

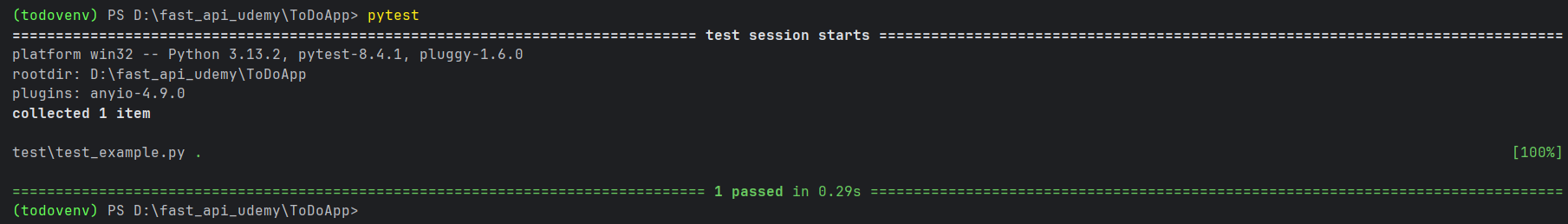
terminal output => 1 passed

**Pytest Introduction:**

def test\_equal\_or\_not\_equal():  
 assert 3 == 3

now in terminal run command “pytest”

now you will get result as shown below,



**Pytest Basics Overview:**

* Validate integers

def test\_equal\_or\_not\_equal():  
 assert 3 == 3

pass

* Validate instances

def test\_is\_instance():  
 assert isinstance('This is a String', str)  
 assert not isinstance(10, int)  
 # pass

* Validate Booleans

def test\_boolean():  
 validated = True  
 assert validated is True  
 assert('hello' == 'world') is False

* Validate Types

def test\_type():  
 assert type ("hello" is str)  
 assert type ("world" is not int)

* Validate greater than and less than value

def test\_greater\_and\_less\_than():  
 assert 7 > 3  
 assert 4 < 10

* Validate Types:

def test\_list():  
 num\_list = [1, 2, 3, 4, 5]  
 any\_list = [False, False]  
 assert 1 in num\_list  
 assert 7 not in num\_list  
 assert all(num\_list)  
 assert not any(any\_list)

Pytest Basics