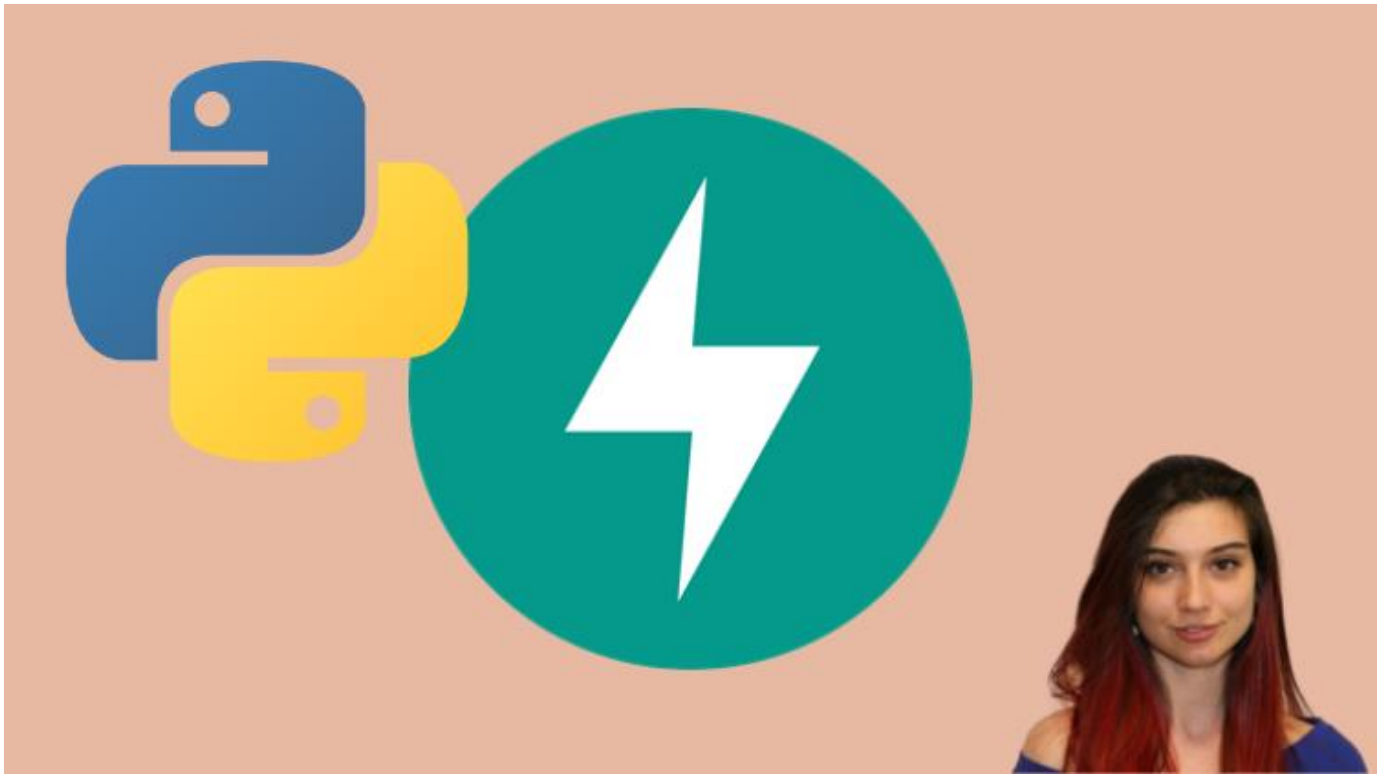
FastAPI REST – Part 5



CODE WITH FINESSE®

Ready to go?

Complaint system (course application – Part 1)

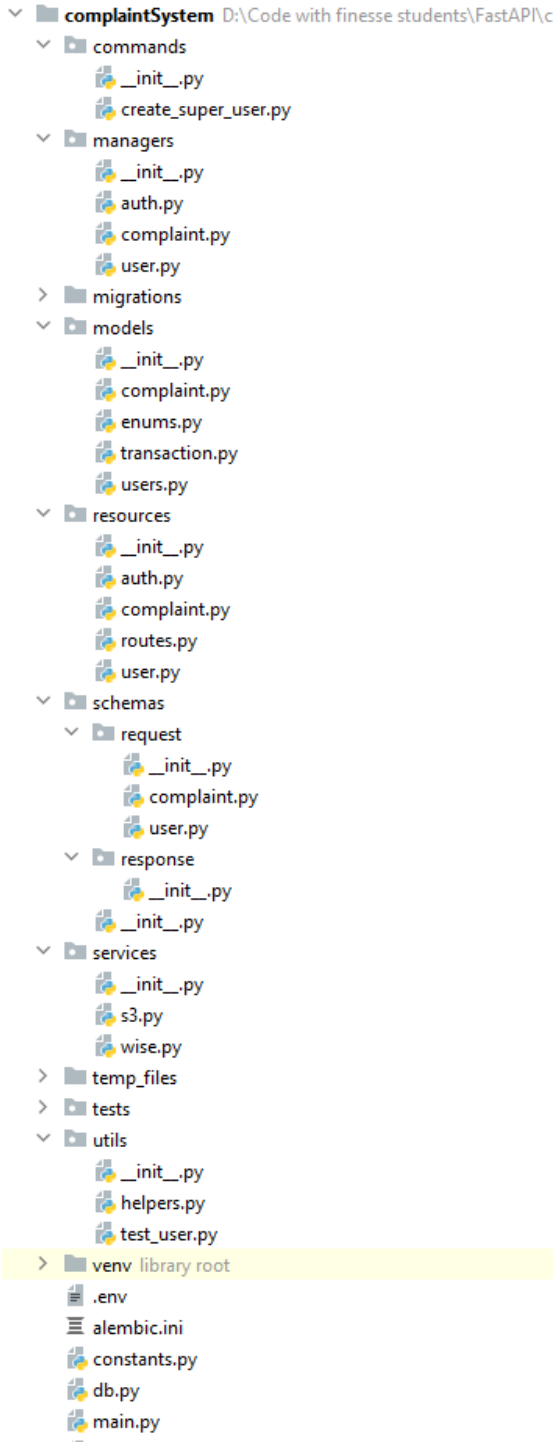
TABLE OF CONTENTS

1. Introduction

In this lecture, we will start building our first complete project. We will follow the good principles of structuring files, writing clean, maintainable, and reusable code. We will also pay attention to the imports because often (especially when you are building Python app), you might find yourself in a position to solve circular import errors.
Today we will define the models, set up authorization and authentication, start building schemas, provide different access levels, and expose some of the endpoints in our app. Then, for better visualization, we will have a frontend project which will help us understand the logic better, and then we will try to connect and see if it is working properly.
In the second part of our app, we will add a couple of third-party providers/services such as AWS (S3 bucket for storing images and SES – simple email service, again from AWS) and a payment provider because often, we will need such knowledge in the real world.

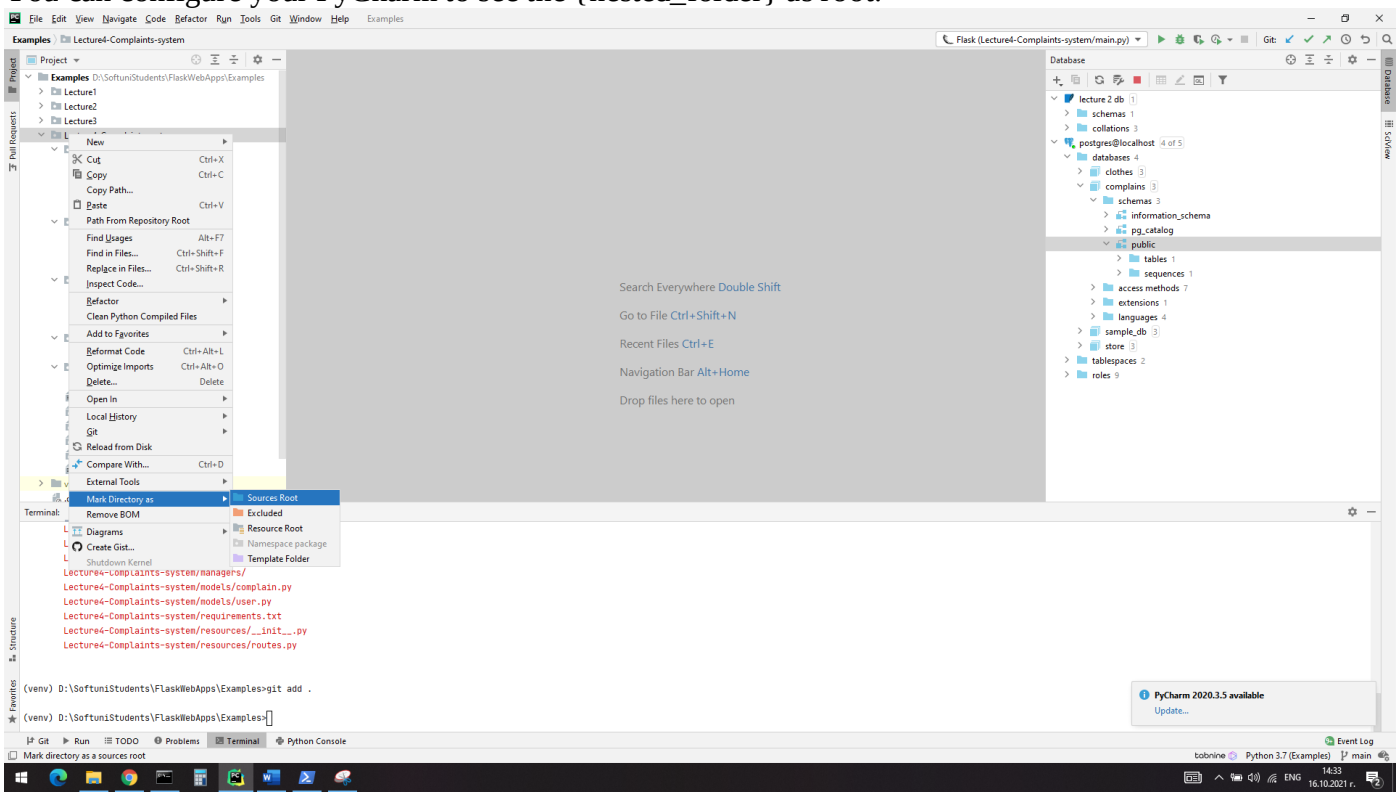
2. Set up skeleton

In this part we will take a look at some common practices of structuring REST API with FastAPI.
The folder structure will look similar to this one:



Do not worry in advanced for all this files, we will take a closer look to each of them and understand their meaning.

It is **highly unlikely**, but if you have a couple of projects under the same directory, then you might consider the following advice as applicable: When you are not in the root, you will have to write imports like this "**from {nested_folder}.models.user import user**". That would not be ideal because if you import just the {nested_folder} on the server, they won't work, and you have to configure paths, and maybe it will break stuff.
You can configure your PyCharm to see the {nested_folder} as root:



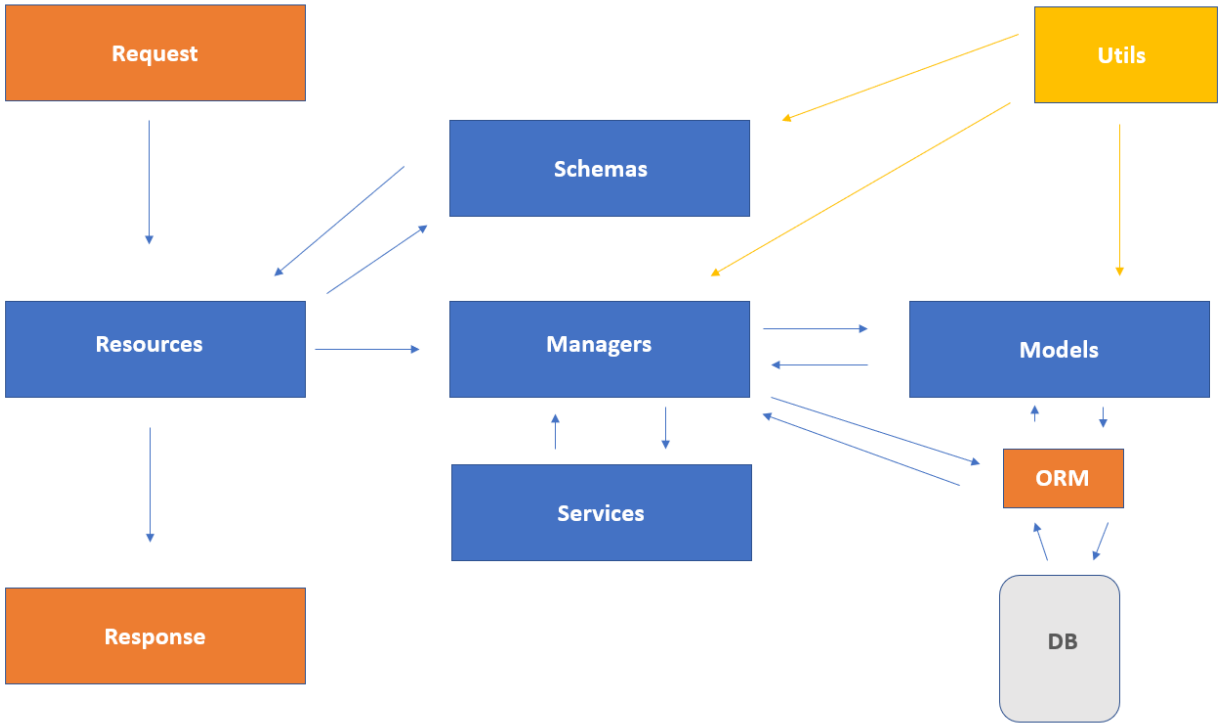
Right click on the directory you want to mark as root, hover ‘mark directory as’ and select ‘source root’. After that the directory will become blue in PyCharm and now you will be able to make imports like this directly “**form models.user import user**”.

Later we will remove or add files depending on our needs, but for now we did this structure with the following meaning:

- **Managers**: they will be responsible for handling the business logic of the app and they will communicate with the database (in our case the ORM). They will transform data if needed, add it to the database and pass the result.
- **Models**: they will be responsible for structuring our tables in the database. Define the schema for the objects we want to store as records (rows)
- **Resources**: They will be our functions for the endpoints of the application. They will define the behavior on GET, POST, PUT, DELETE for different routes. They will communicate with the managers, so that the application can achieve its purpose.

- **Schemas**: They will be responsible for validating the data and structure the response object. We have two folders – for request schemas and for response schemas, because if you remember from previous lecture, they won't always be the same.
- **Services:** They will be responsible for communicating with third party apps like S3, SES and payment provider (we will work on them in the second part of the app). Also, they will communicate with the managers.
- **Utils:** This is the file where we can define helper functions and files, which could be used everywhere in the app

If we try to visualize it, it would like something similar to this:



3. Models

We will have a complain which will be created by complainer and approver which will either approve or reject the complaint. Also, we will have admins, which can view users, delete them or change their roles.

Start with creating a enums.py file under models directory. The we will define the state and the roles:

```
import enum


class RoleType(enum.Enum):
    approver = "approver"
    complainer = "complainer"
    admin = "admin"


class State(enum.Enum):
    pending = "Pending"
    approved = "Approved"
    rejected = "Rejected"
```

Then in the user.py file we can define our user model like this:

```
import sqlalchemy as sqlalchemy
from db import metadata
from models.enums import RoleType

user = sqlalchemy.Table(
    "users",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("email", sqlalchemy.String(120), unique=True),
    sqlalchemy.Column("password", sqlalchemy.String(255)),
    sqlalchemy.Column("first_name", sqlalchemy.String(200)),
    sqlalchemy.Column("last_name", sqlalchemy.String(200)),
    sqlalchemy.Column("phone", sqlalchemy.String(20)),
    sqlalchemy.Column("role", sqlalchemy.Enum(RoleType), nullable=False,
server_default=RoleType.complainer.name),
    sqlalchemy.Column("iban", sqlalchemy.String(200)),
)
```

Now we will have one model, but the role will define, what a user can and can not do in our application.

Another important model is for the complaints. You can see the relationship to the complaint class. Let's see how this class in models.complaint.py file should look like:

```
import sqlalchemy as sqlalchemy
from db import metadata
from models.enums import State

complaint = sqlalchemy.Table(
    "complaints",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("title", sqlalchemy.String(120), nullable=False),
    sqlalchemy.Column("description", sqlalchemy.Text, nullable=False),
    sqlalchemy.Column("photo_url", sqlalchemy.String(200), nullable=False),
    sqlalchemy.Column("amount", sqlalchemy.Float, nullable=False),
    sqlalchemy.Column("created_at", sqlalchemy.DateTime,
server_default=sqlalchemy.func.now()),
    sqlalchemy.Column("status", sqlalchemy.Enum(State), nullable=False,
server_default=State.pending.name),
    sqlalchemy.Column("complainer_id", sqlalchemy.ForeignKey("users.id"),
nullable=False),
)
```

I highly recommend to export the models in the __init__.py of the **models** folder, so that they can be discovered and created from python before you access them from resource:

```
from models.complaint import *
from models.user import *
```

Now we need our metadata – in the db.py file we can write the following code:

```
from decouple import config
import databases
import sqlalchemy

DATABASE_URL =
f"postgresql://{config('DB_USER')}:{config('DB_PASSWORD')}@localhost:5433/complain
ts"
database = databases.Database(DATABASE_URL)
metadata = sqlalchemy.MetaData()
```

Now it is time for our migrations. Do you remember how on the previous examples we used the .env file for the database string in our app, but in the migration files it was unprotected (hardcoded)?

We can solve this:
pip install alembic
alembic init migrations

In the alembic.ini file for the *sqlalchemy.url* instead of hardcoding it, we can do this:
`sqlalchemy.url = postgresql://%(DB_USER)s:%(DB_PASS)s@localhost:5433/complaints`

Please, create new database in PostgreSQL called "complaints"

This is a way of formatting the string, but this should be configured from elsewhere. Please go to *migrations/env.py* file. The file should look like this:

```
from logging.config import fileConfig

from sqlalchemy import engine_from_config
from sqlalchemy import pool

from alembic import context
from decouple import config as env_config
from db import metadata
import models

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config


section = config.config_ini_section
config.set_section_option(section, "DB_USER", env_config("DB_USER"))
config.set_section_option(section, "DB_PASS", env_config("DB_PASSWORD"))
# Interpret the config file for Python logging.
# This line sets up loggers basically.
fileConfig(config.config_file_name)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
target_metadata = metadata

# other values from the config, defined by the needs of env.py,
# can be acquired:
# my_important_option = config.get_main_option("my_important_option")
# ... etc.
```
… (the rest of the file remains the same)

The bold text is what we have added/changed.
We import here the models as well so that they can be visible for alembic, the metadata is bind to **target_metadata** and the new part is the section and configuration – this way we pass to alembic.ini the values for db user and password without hardcoding them.

4. Registration and authentication

First, we will define an auth manager in managers/auth.py file. It will be responsible for encoding and decoding tokens, also we will add the custom HTTPBearer class lecture:

```
from datetime import datetime, timedelta
from typing import Optional

from decouple import config
import jwt
from fastapi import HTTPException
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jwt import ExpiredSignatureError, InvalidTokenError
from starlette.requests import Request
from db import database
from models import RoleType


class AuthManager:
    @staticmethod
    def encode_token(user):
        try:
            to_encode = {"sub": user["id"], "exp":  datetime.utcnow() +
timedelta(minutes=120)}
            return jwt.encode(to_encode, config('JWT_SECRET'), algorithm='HS256')
        except Exception as e:
            raise e


class CustomHTTPBearer(HTTPBearer):
    async def __call__(
            self, request: Request
    ) -> Optional[HTTPAuthorizationCredentials]:
        res = await super().__call__(request)
        from models import user
        try:
            payload = jwt.decode(res.credentials, config('JWT_SECRET'),
algorithms=['HS256'])
            user = await database.fetch_one(user.select().where(user.c.id ==
payload["sub"]))
            request.state.user = user
            return user
        except ExpiredSignatureError:
            raise HTTPException(401, "Token expired.")
        except InvalidTokenError:
            raise HTTPException(401, "Invalid token.")


oauth2_scheme = CustomHTTPBearer()
```

Then we can define a UserManager in managers/user.py file. It would be responsible for communicating with the database, data changes (password hash for example) and communicating with authentication manager if it is necessary:

```
from asyncpg import UniqueViolationError
from fastapi import HTTPException
from passlib.context import CryptContext

from db import database
from managers.auth import AuthManager
from models import user, RoleType

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


class UserManager:
    @staticmethod
    async def register(user_data):
        data = user_data
        data["password"] = pwd_context.hash(data["password"])
        q = user.insert().values(**data)
        try:
            id_ = await database.execute(q)
        except UniqueViolationError:
            raise HTTPException(status_code=400, detail="User with this email
already exists")
        user_do = await database.fetch_one(user.select().where(user.c.id == id_))
        return AuthManager.encode_token(user_do)

    @staticmethod
    async def login(user_data):
        user_do = await database.fetch_one(user.select().where(user.c.email ==
user_data.email))
        if not user_do:
            raise HTTPException(status_code=400, detail="Wrong email or password")
        elif not pwd_context.verify(user_data.password, user_do["password"]):
            raise HTTPException(status_code=400, detail="Wrong email or password")
        return AuthManager.encode_token(user_do)
```

Now we can define our resources for login and register in resources/auth.py file – they will talk to our managers, later we will define schemas and use them within the routes, so that we can be sure only appropriate data will be given to us:

```python
from fastapi import APIRouter

from managers.user import UserManager
from schemas.request.user import UserRegisterIn, UserLoginIn


router = APIRouter(tags=["Auth"])


@router.post("/register/", status_code=201)
async def register(user_data):
    token = await UserManager.register(user_data.dict())
    return {"token": token}


@router.post("/login/")
async def login(user_data):
    token = await UserManager.login(user_data)
    return {"token": token}
```

Note that here we are using @router.post not **app.**
We can not use the app, as it is defined in the main.py file. This file (main.py) is the starting point of our application, so if we import the app from main in the above code, we will get circular import, and the app will never run.
Instead, we are using the APIRouter object from FastAPI and we will add all routes from all files in *resources/routes.py* like this:

```python
from fastapi import APIRouter
from resources import auth


api_router = APIRouter()
api_router.include_router(auth.router)
```

Next, we will see how all these routes are registered to the **app** object in main.py file.

5. Configuration of the application

The main.py file would like the following way:

```
from fastapi import FastAPI

from db import database
from resources.routes import api_router

app = FastAPI()
app.include_router(api_router)


@app.on_event("startup")
async def startup():
    await database.connect()


@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()
```

We are starting/stopping the database (imported from *db.py* file) in the @app.on_event functions and the new thing here is:
app.include_router(api_router)
This is actually all the routes we have registrated in routes.py (and we will continue register them as the application grows)

6. Schemas

We will reuse parts of the code from previous lecture, but we have to structure it better and also add additional fields. We will create a couple of schemas under schemas/request/user:

```
from pydantic import BaseModel


class UserBase(BaseModel):
    email: str


class UserRegisterIn(UserBase):
    password: str
    phone: str
    first_name: str
    last_name: str
    iban: str


class UserLoginIn(UserBase):
    password: str
```

And now we need to configure our routes to user these two schemas for the input data:

```
@router.post("/register/", status_code=201)
async def register(user_data: UserRegisterIn):
    token = await UserManager.register(user_data.dict())
    return {"token": token}


@router.post("/login/")
async def login(user_data: UserLoginIn):
    token = await UserManager.login(user_data)
    return {"token": token}
```

7. Stop and test

Do not forget to add the header Content-Type: application/json if you are using postman, if you are using the built-in swagger it is appended automatically.
When building an application, especially from scratch, it is essential to stop and test it manually regularly to ensure it is working as expected. So far, we have done a feature for login and registration, and it is time to test it via Postman or swagger (http://127.0.0.1:8000/docs)
I recommend using the debugger, going through each step we have written so far, and ensuring it works correctly.
Make sure you test:

- Validation of the schemas
- The register route
- Login route

Check the database to assure we are actually storing the data correctly.

Also if you are having troubles, because your project is not directly in the root folder, you might consider these commands helpful. You have to execute them in the folder of you project:

```
set PYTHONPATH=./
```

(You have to run this command from the root folder of the project for which you want to set it)

8. List/Create complaints

Often in work you will read tickets as stories. For example: “As a complainer I want to be able to create a complain and send it for review. I want to be able to review all of my previously submitted complains”. This story tells us that we need a resource class with get and post methods which will be restricted only to complainers to see their own info and create new complains.
We will have to create a manager who can deal with the claims, of course we need to add schemas (this time we will have an output schema as well).

In the managers/complain.py file we can add manager

```
from db import database
from models import complaint, RoleType, State


class ComplaintManager:
    @staticmethod
    async def get_complaints(user):
        q = complaint.select()
        if user["role"] == RoleType.complainer:
            q = q.where(complaint.c.complainer_id == user["id"])
        elif user["role"] == RoleType.approver:
            q = q.where(complaint.c.state == State.pending)
        return await database.fetch_all(q)

    @staticmethod
    async def create_complaint(complaint_data, user):
        data = complaint_data.dict()
        data["complainer_id"] = user["id"]
        id_ = await database.execute(complaint.insert().values(**data))
        return await database.fetch_one(complaint.select().where(complaint.c.id ==
id_))
```

The schemas would look like this *schemas/request/complaint.py*:

```
from pydantic import BaseModel


class ComplaintIn(BaseModel):
    title: str
    description: str
    photo_url: str
    amount: float
```

And in the *schemas/response/complaint.py*

```
from datetime import datetime

from pydantic import BaseModel

from models import State


class ComplaintOut(BaseModel):
    id: int
    title: str
    description: str
    photo_url: str
    amount: float
    created_at: datetime
    status: State
```

The fields title, description, photo_url,amount are repeated twice, so we can extract the logic. Create a file under *schemas* called *base.py*:

```
from pydantic import BaseModel


class BaseComplaint(BaseModel):
    title: str
    description: str
    photo_url: str
    amount: float
```

And now we refactor the request and response file:

```
from schemas.base import BaseComplaint



class ComplaintIn(BaseComplaint):
    pass
```

And the response schema would be:

```python
from datetime import datetime

from models import State
from schemas.base import BaseComplaint


class ComplaintOut(BaseComplaint):
    id: int
    created_at: datetime
    status: State
```

Let's create a simple authorization – only users with role "complainer" can make claims – in *managers/auth.py* we can add the following:

```python
def is_complainer(request: Request):
    if not request.state.user["role"] == RoleType.complainer:
        raise HTTPException(status_code=403, detail="Forbidden")


def is_approver(request: Request):
    if not request.state.user["role"] == RoleType.approver:
        raise HTTPException(status_code=403, detail="Forbidden")


def is_admin(request: Request):
    if not request.state.user["role"] == RoleType.admin:
        raise HTTPException(status_code=403, detail="Forbidden")
```

This way we can configure different routes to be accepting only requests from specific roles – only complainer can create complain, but all roles can get complaints, only admin can delete complaint, only approver can approve/reject complaint and ect.

And now we need to define our endpoints in *resources/complaint.py*:

```
from typing import List

from fastapi import APIRouter, Depends
from starlette.requests import Request

from managers.auth import oauth2_scheme, is_complainer
from managers.complaint import ComplaintManager
from schemas.request.complaint import ComplaintIn
from schemas.response.complaint import ComplaintOut

router = APIRouter(tags=["Complaints"])


@router.get("/complaints/", dependencies=[Depends(oauth2_scheme)],
response_model=List[ComplaintOut])
async def get_complaints(request: Request):
    user = request.state.user
    return await ComplaintManager.get_complaints(user)


@router.post("/complaints/", dependencies=[Depends(oauth2_scheme),
Depends(is_complainer)], response_model=ComplaintOut)
async def create_complaint(request: Request, complaint: ComplaintIn):
    user = request.state.user
    return await ComplaintManager.create_complaint(complaint, user)
```

The first one only needs authentication, the second one needs authorization as well for specific role.

Last, but not least, we need to register the router in *resources/routes.py*:

```
from fastapi import APIRouter
from resources import auth, complaint


api_router = APIRouter()
api_router.include_router(auth.router)
api_router.include_router(complaint.router)
```

Now we have finished the complains feature as well. Do not forget to test it with postman/swagger and assure everything is working just fine.

9. Admin part

As admins we need to be able to delete complaints. First, we can start by adding logic in managers/complaint.py in ComplaintManager:

```
@staticmethod
async def delete(complaint_id):
    await database.execute(complaint.delete().where(complaint.c.id ==
complaint_id))
```

And then we can expose an endpoint, which is strictly guarded only for admin usage in *resources/complaint.py*:

```
@router.delete("/complaints/{complaint_id}/",
dependencies=[Depends(oauth2_scheme), Depends(is_admin)], status_code=204)
async def delete_user(complaint_id: int):
    await ComplaintManager.delete(complaint_id)
```

And we are done! We do not need to register anything, because this router is already presented in the *resource.py* file.

Now we can add new logic – to view user, search it by email, or get all, or to change the role of any existing user to approver or admin. Let's start with adding a logic to UserManager class:

```
@staticmethod
async def get_user_by_email(email):
    return await database.fetch_all(user.select().where(user.c.email == email))

@staticmethod
async def get_users():
    return await database.fetch_all(user.select())

@staticmethod
async def change_role(role: RoleType, user_id: int):
    await database.execute(user.update().where(user.c.id ==
user_id).values(role=role))
```

Then we can go to resources/user.py and add the following endpoints:

```
from typing import Optional

from fastapi import APIRouter, Depends

from managers.auth import oauth2_scheme, is_admin
from managers.user import UserManager
from models import RoleType

router = APIRouter(tags=["Users"])


@router.get("/users/", dependencies=[Depends(oauth2_scheme), Depends(is_admin)])
async def get_users(email: Optional[str] = None):
    if email:
        return await UserManager.get_user_by_email(email)
    return await UserManager.get_users()


@router.put("/users/{user_id}/make-admin", dependencies=[Depends(oauth2_scheme),
Depends(is_admin)], status_code=204)
async def make_admin(user_id: int):
    await UserManager.change_role(RoleType.admin, user_id)


@router.put("/users/{user_id}/make-approver",
dependencies=[Depends(oauth2_scheme), Depends(is_admin)], status_code=204)
async def make_admin(user_id: int):
    await UserManager.change_role(RoleType.approver, user_id)
```

In the GET endpoint we are defining an optional query param “email”, if there is a param, we will search for a specific user, if not we simply return all users.
Also you can move the UserBase class from schemas/request/user.py to the base.py in schemas and create a new file schemas/responses/user.py and create UserOut schema:

```
from models import RoleType
from schemas.base import UserBase


class UserOut(UserBase):
    id: int
    first_name: str
    last_name: str
    phone: str
    role: RoleType
    iban: str
```

How will you configure it in the GET /users/ endpoint?

10. Approve/Reject complaints

We need to provide two endpoints – one for approving a specific complaint, one for rejecting a specific complaint. Whatever action is taken on the complaint, we need to update its status with appropriate one. Now it would be a lot easier for us, because we have code base we can reuse. We still need to create a couple of new things though. First add in managers/complain.py in ComplaintManager class two new methods:

```
@staticmethod
async def approve(id_):
    await database.execute(complaint.update().where(complaint.c.id ==
id_).values(status=State.approved))

@staticmethod
async def reject(id_):
    await database.execute(complaint.update().where(complaint.c.id ==
id_).values(status=State.rejected))
```

Then in the resources/complaints.py we will create two new endpoints. Each one will be accessible only by approvers:

```
@router.put("/complaints/{complaint_id}/approve",
dependencies=[Depends(oauth2_scheme), Depends(is_approver)], status_code=204)
async def approve_complaint(complaint_id: int):
    await ComplaintManager.approve(complaint_id)


@router.put("/complaints/{complaint_id}/reject",
dependencies=[Depends(oauth2_scheme), Depends(is_approver)], status_code=204)
async def approve_complaint(complaint_id: int):
    await ComplaintManager.reject(complaint_id)
```

And this should be enough. Do not forget to test the app with Postman/Swagger, you should send requests to these endpoints with approver's token.

11. More tips and tricks

- It is vital to format the code itself when we are creating a project. There are many different formatters out there for python. You can install whatever you find best, but I would recommend 'black'. You can integrate it in your PyCharm as an external tool or create 'on commit' hook, which will format your files on each commit. If you are struggling with the installation it, you can follow this tutorial

- If you have browsed any python projects on Github or elsewhere, you have probably noticed a file called requirements.txt This requirements.txt file is used for specifying what python packages are required to run the project you are looking at. Typically the requirements.txt file is located in the root directory of your project. It helps us keep track of the packages we are using and their versions and also makes the installation of all external packages on the server really easy. To freeze your packages run this command in the root folder:
  `pip freeze > requirements.txt`
- You should structure imports as well. The most common way of structuring them is:

  1. Standard library imports
  2. Third-party imports
  3. Application-specific imports

  It is not needed to do it by hand, you can go to some python file and press ctrl+alt+o this will order and optimize (remove unused imports) the imports. **Note that you should not remove this part even though it looks like it is not used**
  Import models for migrations/env.py file

- .gitignore file - To avoid unwanted files to be committed in Remote Repository of GIT. Simply provide appropriate description in .gitignore file like, if I want to avoid files with extension of pyc file or .class file then, *.pyc *.class I'll put these two lines in .gitignore file. in future I dont need to bother for these files which will never committed in GIT repo. A good .gitignore file you can use directly is this one

- READme file - We mostly tend to ignore the README file as a trivial non-essential part of your project. However, a good README file is what an Index is to a Book. README file enables users to navigate and identify essential elements of your project quickly.A goo readme file example you can find here

12. Create the first user

You are probably wondering – ok, now it is easy to create complainer, because they can register themselves. After that I can make some of them approvers or admins, but how can I make the very first admin?
The right answer to this question is via script. Why? Because it is really easy to run a script on a server, and then the user will be created. Because we are using asynchronous approach we will install:

```
pip install asyncclick
```

You can view more info here
Then in *commands/create_super_user.py*:

```
import asyncclick as click

from db import database
from managers.user import UserManager
from models import RoleType


@click.command()
@click.option("-f", "--first_name", type=str, required=True)
@click.option("-l", "--last_name", type=str, required=True)
@click.option("-e", "--email", type=str, required=True)
@click.option("-p", "--phone", type=str, required=True)
@click.option("-i", "--iban", type=str, required=True)
@click.option("-pa", "--password", type=str, required=True)
async def create_user(first_name, last_name, email, phone, iban, password):
    user_data = {"first_name": first_name, "last_name": last_name, "email": email,
"phone": phone, "iban": iban, "password": password, "role": RoleType.admin}
    await database.connect()
    await UserManager.register(user_data)
    await database.disconnect()

if __name__ == "__main__":
    create_user(_anyio_backend="asyncio")
```

With the options we are able to pass the arguments from the console. Select a Git Bash console and then try the following:
export PYTHONPATH=./

python commands/create_super_user.py -f Test -l Admin -e test@admin.com -p 123456789 -i BNBG96611020345678 -pa 123

Now start the server and try to login with this user.

13. Bonus – add CORS

In main.py we need to add the following code:
Here you can info more info what is CORS

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from db import database
from resources.routes import api_router


origins = [
    "http://localhost",
    "http://localhost:4200",
]

app = FastAPI()
app.include_router(api_router)
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


@app.on_event("startup")
async def startup():
    await database.connect()


@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()
```

We are specifying http://localhost:4200 because this will be the url and port that we will use for the frontend application at the end.