# Complaints system

## 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 Flask app), you might find yourself in a position to solve circular import errors.

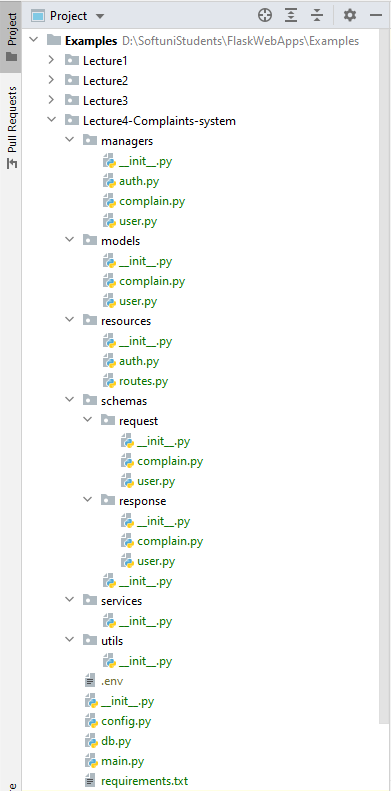
Today we will define the models, set up authorization and authentication, start building schemas, provide different access levels, and expose some of the endpoints in our app. Then, for better visualization, we will have a frontend project which will help us understand the logic better, and then we will 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 a payment provider because often, we will need such knowledge in the real world. Finally, at the end of part 2, we will add unit tests to our application to ensure we produce quality and free of bugs software as much as possible.

## Set up skeleton

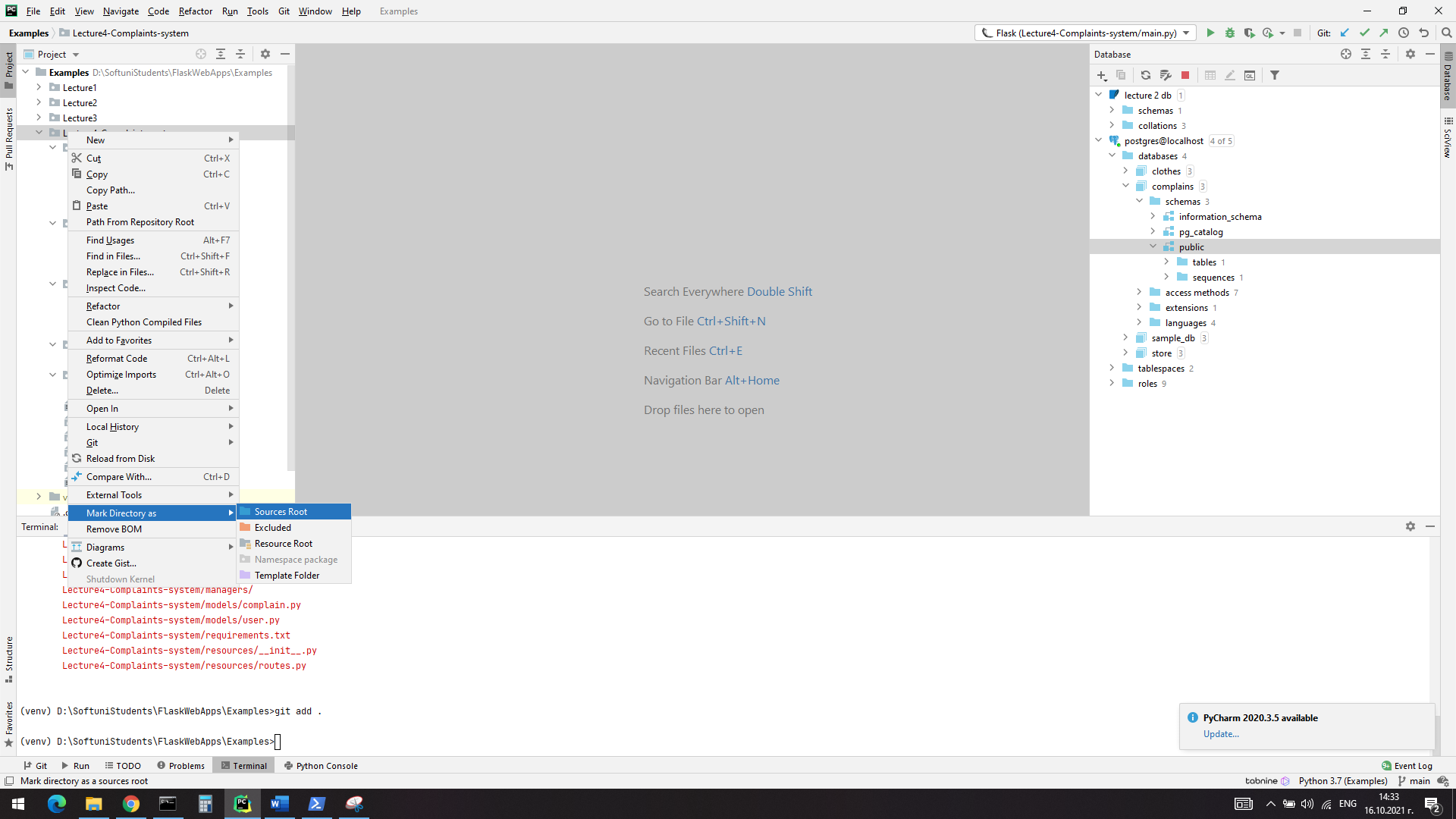
In this part we will take a look at some common practices of structuring REST API with Flask.

The folder structure will look similar to this one:



Keep in mind I have a couple of projects in my root folder (Examples). The case for you might be that you are already in a new project. If so, you can just add the files in folder Lecture4 (without nesting them in this folder and creating them under your root directly). However, if this is not the case and you are doing this project under a root with other projects, 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:

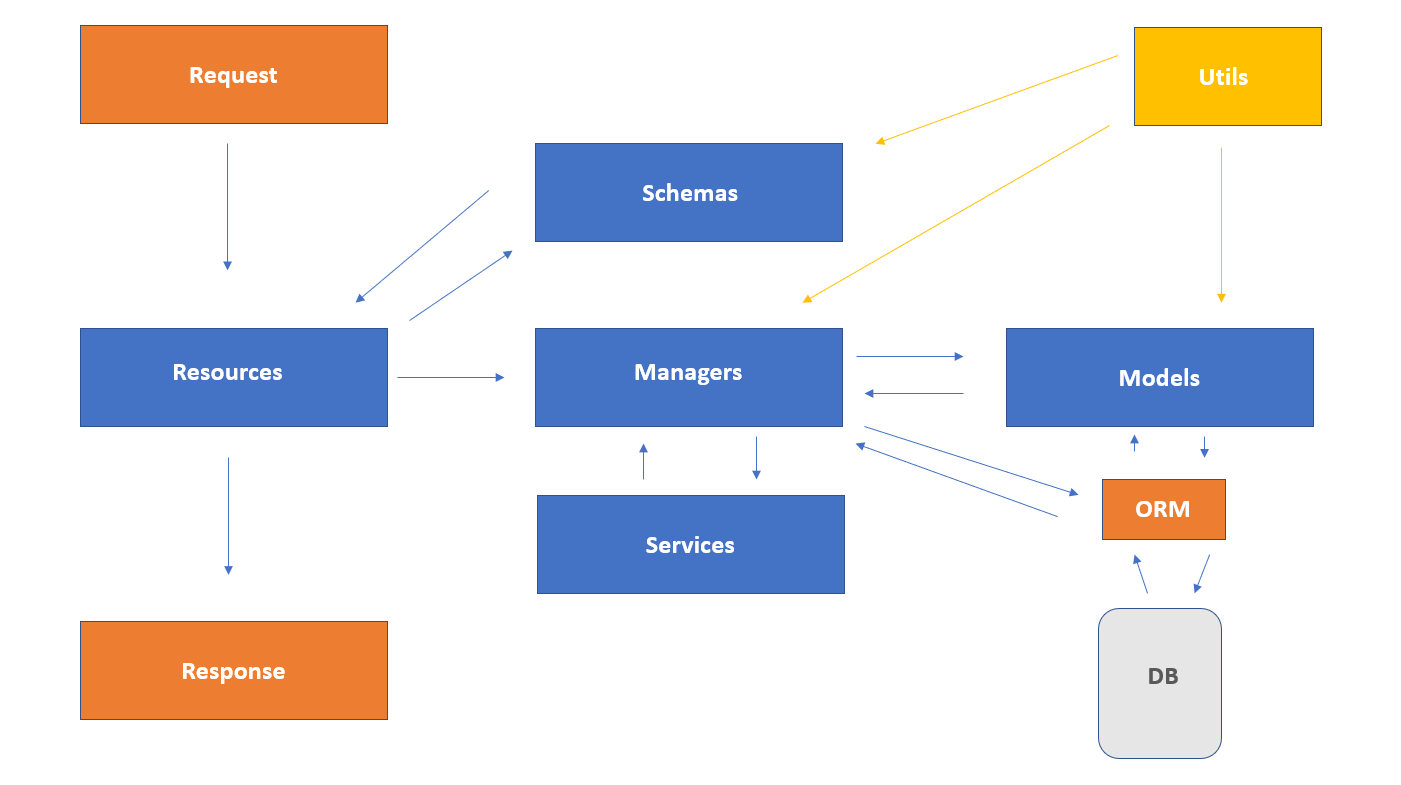


Right click on the directory you want to mark as root, hover ‘mark directory as’ and select ‘source root’. After that the directory will become blue in PyCharm and now you will be able to make imports like this directly “**form models.user import User**”.

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 classes which inherit from the Resource class from flask\_restfull. They will define the behavior on GET, POST, PUT, DELETE for different endpoints. 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 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:



## 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 delete the complains and add other admins and approvers.

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 users like this:

from db import db  
from models.enums import RoleType  
  
  
class BaseUserModel(db.Model):  
 \_\_abstract\_\_ = True  
  
 id = db.Column(db.Integer, primary\_key=True)  
 first\_name = db.Column(db.String(20), nullable=False)  
 last\_name = db.Column(db.String(20), nullable=False)  
 email = db.Column(db.String(120), nullable=False, unique=True)  
 phone = db.Column(db.String(100), nullable=False)  
 password = db.Column(db.String(255), nullable=False)  
  
  
class ComplainerModel(BaseUserModel):  
 \_\_tablename\_\_ = **'complainers'** complains = db.relationship(**"ComplaintModel"**, backref=**"complaint"**, lazy=**'dynamic'**)  
 role = db.Column(  
 db.Enum(RoleType),  
 default=RoleType.complainer,  
 nullable=False  
 )  
  
  
class ApproverModel(BaseUserModel):  
 \_\_tablename\_\_ = **'approvers'** certificate = db.Column(db.String(255), nullable=False)  
 role = db.Column(  
 db.Enum(RoleType),  
 default=RoleType.approver,  
 nullable=False  
 )  
  
  
class AdministratorModel(BaseUserModel):  
 \_\_tablename\_\_ = **'administrators'** role =db.Column(  
 db.Enum(RoleType),  
 default=RoleType.admin,  
 nullable=False  
 )

Note how we use the power of inheritance to bring all the properties that we need for the BaseUserModel in each other type of user.

Also, we can extend the model further depending on the table’s need (we add a certificate to the approver for example). You can see the relationship to the complaint class. Let’s see how this class in models.complaint.py file should look like:

from sqlalchemy import func  
  
from db import db  
from models.enums import State  
  
  
class ComplaintModel(db.Model):  
 \_\_tablename\_\_ = **'complaints'** id = db.Column(db.Integer, primary\_key=True)  
 title = db.Column(db.String(100), nullable=False)  
 description = db.Column(db.Text, nullable=False)  
 photo\_url = db.Column(db.String(255), nullable=False)  
 amount= db.Column(db.Float, nullable=False)  
 create\_on = db.Column(db.DateTime, server\_default=func.now())  
 status =db.Column(  
 db.Enum(State),  
 default=State.pending,  
 nullable=False  
 )  
 complainer\_id = db.Column(db.Integer, db.ForeignKey(**'complainers.id'**))  
 complainer = db.relationship(**'ComplainerModel'**)

You can see how we are defining here the backreference to the complainer model. This db is nothing but an SQLAlchemy object defined in db.py file:

from flask\_sqlalchemy import SQLAlchemy  
  
db = SQLAlchemy()

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 \*

## Registration and authentication of complainers

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 auth.verify token from previous lecture. We will do it dynamically with eval, so that it can work for each type of user.:

from datetime import datetime, timedelta  
from decouple import config  
import jwt  
from werkzeug.exceptions import Unauthorized  
from flask\_httpauth import HTTPTokenAuth  
  
*# Keep this imports because of the eval function*from models.user import ComplainerModel, ApproverModel  
  
  
class AuthManager:  
 @staticmethod  
 def encode\_token(user):  
 payload = {**"sub"**: user.id, **"exp"**: datetime.utcnow() + timedelta(days=2), **"type"**: user.\_\_class\_\_.\_\_name\_\_}  
 return jwt.encode(payload, key=config(**"SECRET\_KEY"**), algorithm=**"HS256"**)  
  
 @staticmethod  
 def decode\_token(token):  
 try:  
 info = jwt.decode(jwt=token, key=config(**'SECRET\_KEY'**), algorithms=[**"HS256"**])  
 return info[**'sub'**], info[**"type"**]  
 except Exception as ex:  
 raise ex  
  
  
auth = HTTPTokenAuth(scheme=**'Bearer'**)  
  
  
@auth.verify\_token  
def verify\_token(token):  
 try:  
 user\_id, type\_user = AuthManager.decode\_token(token)  
 return eval(**f"**{type\_user}**.query.filter\_by(id=**{user\_id}**).first()"**)  
 except Exception as ex:  
 raise Unauthorized(**"Invalid or missing token"**)

Then we can define a ComplainerManager 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 werkzeug.exceptions import BadRequest  
from werkzeug.security import check\_password\_hash, generate\_password\_hash  
  
from managers.auth import AuthManager  
from models.user import ComplainerModel  
from db import db  
  
  
class ComplainerManager:  
 @staticmethod  
 def register(complainer\_data):  
 *"""  
 Hashes the plain password  
 :param complainer\_data: dict  
 :return: complainer  
 """* complainer\_data[**"password"**] = generate\_password\_hash(complainer\_data[**'password'**], method=**'sha256'**)  
 complainer = ComplainerModel(\*\*complainer\_data)  
 try:  
 db.session.add(complainer)  
 db.session.flush()  
 return AuthManager.encode\_token(complainer)  
 except Exception as ex:  
 raise BadRequest(str(ex))  
  
 @staticmethod  
 def login(data):  
 *"""  
 Checks the email and password (hashes the plain password)  
 :param data: dict -> email, password  
 :return: token  
 """* try:  
 complainer = ComplainerModel.query.filter\_by(email=data[**"email"**]).first()  
 if complainer and check\_password\_hash(complainer.password, data[**"password"**]):  
 return AuthManager.encode\_token(complainer)  
 raise Exception  
 except Exception:  
 raise BadRequest(**"Invalid username or password"**)

Interesting approach here is to flush the record instead of directly committing it, do not worry we will commit at in the end of the request.

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 put a decorator on them, so that we can be sure only appropriate data will be given to us:

from flask\_restful import Resource  
from flask import request  
from managers.user import ComplainerManager  
  
  
class RegisterComplainer(Resource):  
 def post(self):  
 data = request.get\_json()  
 token = ComplainerManager.register(data)  
 return {**"token"**: token}, 201

class LoginComplainer(Resource):  
 def post(self):  
 data = request.get\_json()  
 token = ComplainerManager.login(data)  
 return {**"token"**: toke, "role": "complainer" }

NB! \* as a home assignment, investigate how the role could be encoded in the token. Which method is better?

Last, but not least we will add the resources in resources/routes.py file in the routes tuple:

from resources.auth import RegisterComplainer, LoginComplainer  
  
routes = (  
 (RegisterComplainer, **"/register"**),  
 (LoginComplainer, **"/login"**)  
)

## Configuration of the application

Instead of configure our flask application in chunks like we did in the previous lectures, we will do all at once.

It provides us the flexibility to choose from different predefined environments. When we developing the application, we never use the production database, otherwise the data would be a nightmare.

For this we will need a config object. Define the following class in the config.py file:

from decouple import config  
  
  
class ProductionConfig:  
 FLASK\_ENV = **"prod"** DEBUG = False  
 TESTING = False  
 SQLALCHEMY\_DATABASE\_URI = (  
 **f"postgresql://**{config(**'DB\_USER'**)}**:**{config(**'DB\_PASSWORD'**)}**"  
 f"@localhost:**{config(**'DB\_PORT'**)}**/**{config(**'DB\_NAME'**)}**"** )  
  
  
class DevelopmentConfig:  
 FLASK\_ENV = **"development"** DEBUG = True  
 TESTING = True  
 SQLALCHEMY\_DATABASE\_URI = (  
 **f"postgresql://**{config(**'DB\_USER'**)}**:**{config(**'DB\_PASSWORD'**)}**"  
 f"@localhost:**{config(**'DB\_PORT'**)}**/**{config(**'DB\_NAME'**)}**"** )

The main.py file would like the following way:

from flask import Flask  
from flask\_migrate import Migrate  
from flask\_restful import Api  
  
from db import db  
from resources.routes import routes  
  
app = Flask(\_\_name\_\_)  
app.config.from\_object(**'config.DevelopmentConfig'**)  
  
api = Api(app)  
migrate = Migrate(app, db)  
  
  
@app.before\_first\_request  
def create\_tables():  
 db.init\_app(app)  
 db.create\_all()  
  
  
@app.after\_request  
def close\_request(response):  
 db.session.commit()  
 return response  
  
  
[api.add\_resource(\*route) for route in routes]  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 app.run()

Here we use app.config.from\_object(**'config.ProductionConfig'**) and this way we register all the configuration variables at once from our object class from config.py file

You can notice that we use the flask middleware before\_first\_request and after\_request. The first one is used in our case to create all the tables in the database. The second one is user for committing the data into the database (remember the .flush() previously?)

We have a list comprehension which is responsible for register all of our exposed routes. Of course, we have to migrate and upgrade as we did it previously.

## Schemas

We will reuse parts of the code from previous lecture, but we have to structure it better and also add additional fields. Create a file called ‘decorators’ under utils folder. There we will use our decorator, but we will replace the abort with bad request:

from functools import wraps  
from flask import request  
from werkzeug.exceptions import BadRequest, Forbidden  
  
  
def validate\_schema(schema\_name):  
 def decorator(f):  
 @wraps(f)  
 def decorated\_function(\*args, \*\*kwargs):  
 schema = schema\_name()  
 errors = schema.validate(request.get\_json())  
 if errors:  
 raise BadRequest(**f"Invalid fields** {errors}**"**)  
 return f(\*args, \*\*kwargs)  
 return decorated\_function  
 return decorator

After that we will create a couple of schemas under schemas/request/user:

from marshmallow import Schema, fields  
  
  
class UserSchema(Schema):  
 email = fields.Email(required=True)  
 password = fields.String(required=True)  
   
   
class RequestRegisterUserSchema(UserSchema):  
 first\_name = fields.String(min\_length=2, max\_length=20, required=True)  
 last\_name = fields.String(min\_length=2, max\_length=20, required=True)  
 phone = fields.String(min\_length=10, max\_length=13, required=True)  
   
  
class RequestLoginUserSchema(UserSchema):  
 pass

The name convention might seem like an overkill now, but later on, when the project grows up it will serve us good.

Last, we need to decorate our methods with the appropriate schemas:

from flask\_restful import Resource  
from flask import request  
from managers.user import ComplainerManager  
from schemas.request.user import RequestRegisterUserSchema, RequestLoginUserSchema  
from utils.decorators import validate\_schema  
  
  
class RegisterComplainer(Resource):  
 @validate\_schema(RequestRegisterUserSchema)  
 def post(self):  
 data = request.get\_json()  
 token = ComplainerManager.register(data)  
 return {**"token"**: token}, 201  
  
  
class LoginComplainer(Resource):  
 @validate\_schema(RequestLoginUserSchema)  
 def post(self):  
 data = request.get\_json()  
 token = ComplainerManager.login(data)  
 return {**"token"**: token , "role": "complainer" }

## Stop and test the app manually

**Do not forget to add the header Content-Type: application/json** to **each** request so that you have access to request.get\_json()

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.

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 FLASK\_APP=./main.py

set PYTHONPATH=./

## Create/List complains

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 these manager (not the get\_all\_complainer\_claims):

from db import db  
from models.complaint import ComplaintModel  
  
  
class ComplaintManager:  
 @staticmethod  
 def get\_all\_complainer\_claims(user):

if isinstance(user, ComplainerModel):

return ComplaintModel.query.filter\_by(complainer\_id=user.id).all()

return ComplaintModel.query.all()  
  
 @staticmethod  
 def create(data, complainer\_id):  
 data[**"complainer\_id"**] = complainer\_id  
 c = ComplaintModel(\*\*data)  
 db.session.add(c)  
 db.session.flush()  
 return c

Because the complain schema for request and response will have a lot of similar fields we can create a file called ‘bases.py’ under schema folder and we can define the base schema like this:

from marshmallow import Schema, fields  
  
  
class BaseComplainSchema(Schema):  
 title = fields.String(required=True)  
 description = fields.String(required=True)  
 photo\_url = fields.String(required=True)  
 amount = fields.Float(required=True)

Then we can define our request and response complain schemas as following:

In the schemas/request/complaint.py:

In the schemas/response/complaint.py:

from marshmallow import fields  
from models.enums import State  
from marshmallow\_enum import EnumField  
from schemas.bases import BaseComplainSchema  
  
  
class ComplaintResponseSchema(BaseComplainSchema):  
 id = fields.Integer(required=True)  
 status = EnumField(State, by\_value=True)  
 create\_on = fields.DateTime(required=True)

from schemas.bases import BaseComplainSchema  
  
  
class RequestComplainSchema(BaseComplainSchema):  
 pass

In order to be able to serialize enum fields we need to install marshmallow\_enum and use EnumField.

Now, we need to define our resource class in resources/complaint.py:

from flask\_restful import Resource  
  
from managers.auth import auth  
from managers.complain import ComplaintManager  
from models.enums import RoleType  
from flask import request  
  
from schemas.request.complain import RequestComplainSchema  
from schemas.response.complain import ComplaintResponseSchema  
from utils.decorators import permission\_required, validate\_schema  
  
  
class ComplaintListCreate(Resource):  
 @auth.login\_required  
 @validate\_schema(RequestComplainSchema)  
 def get(self):  
 user = auth.current\_user()  
 complains = ComplaintManager.get\_all\_complainer\_claims(user)  
 *# Use dump, not load when schema and object are not the same* return ComplaintResponseSchema().dump(complains, many=True)

@auth.login\_required  
 @permission\_required(RoleType.complainer)  
 @validate\_schema(RequestComplainSchema)  
 def post(self):  
 complainer = auth.current\_user()  
 data = request.get\_json()  
 complain = ComplaintManager.create(data, complainer.id)  
 *# Use dump, not load when schema and object are not the same* return ComplaintResponseSchema().dump(complain)

Last, we need to register the resource in resources/routes.py file:

from resources.auth import RegisterComplainer, LoginComplainer  
from resources.complaint import ComplaintListCreate  
  
routes = (  
 (RegisterComplainer, **"/register"**),  
 (LoginComplainer, **"/login"**),  
 (ComplaintListCreate, **"/complainers/complaints"**)  
)

New finished the complains feature as well. Do not forget to test it with postman. If you experience any difficulties, you can pull the repo and revert to [this](https://github.com/InesIvanova/Flask-course-prep-materials/commit/7029f41276d864ed636513e6d93065154a72e68d) commit.

## Admin part – register approvers and other admins, delete complaints

This part is a **home assignment**. You can reuse all the logic we have written so far for decorators and functions/methods that you think can be helpful.

You have to create new resources and register the new routes. You have to protect the routes to be accessible only by admins.

Test your application with Postman on each feature and debug it.

If you find any difficulties, you can refer to this [commit](https://github.com/InesIvanova/Flask-course-prep-materials/commit/126a1926c80381d650cdc8bd9b249398e07d310d). It creates all the necessary stuff for this part. Keep in mind the code is working, but it is not refactored on purpose. You can try to reduce the repetitive code by introducing some level of abstraction as we did for the tables.

## 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 easer 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  
def approve(id\_):  
 ComplaintModel.query.filter\_by(id=id\_).update({**"status"**: State.approved})  
  
@staticmethod  
def reject(id\_):  
 ComplaintModel.query.filter\_by(id=id\_).update({**"status"**: State.rejected})

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

class ApproveComplaint(Resource):  
 @auth.login\_required  
 @permission\_required(RoleType.approver)  
 def put(self, id\_):  
 ComplaintManager.approve(id\_)  
 return 200  
  
  
class RejectComplainComplaint(Resource):  
 @auth.login\_required  
 @permission\_required(RoleType.approver)  
 def put(self, id\_):  
 ComplaintManager.reject(id\_)  
 return 200

In the routes tupple we will add two endpoints:

routes = (

….

(ApproveComplaint, "/approvers/complaints/<int:id\_>/approve"),

(RejectComplainComplaint, "/approvers/complaints/<int:id\_>/reject"),

…..

)

And this should be enough. Do not forget to test the app with Postman, you should send requests to these endpoints with approver’s token. The commit is [here](https://github.com/InesIvanova/Flask-course-prep-materials/commit/fc2f921f9455164cdc8801c57b174d8ed0847551)

## More tips and tricks for quality code

* **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](https://www.youtube.com/watch?v=dxFsjgtyAHw&ab_channel=FedericoTartarini)
* **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**

**from models.user import ComplainerModel, ApproverModel, AdministratorModel**

* **.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](https://github.com/github/gitignore/blob/master/Python.gitignore)
* 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](https://github.com/bbc/REST-API-example/blob/master/README.md)

## BONUS – add CORS

pip install flask\_cors

In main.py we need to add the following code:

from flask import Flask  
from flask\_cors import CORS  
from flask\_migrate import Migrate  
from flask\_restful import Api  
  
from db import db  
from resources.routes import routes  
  
app = Flask(\_\_name\_\_)  
app.config.from\_object(**'config.DevelopmentConfig'**)  
  
api = Api(app)  
migrate = Migrate(app, db)  
CORS(app)

….

[Here](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) you can info more info what is CORS