**Configure Docker:**

Generate a token from:

<https://hub.docker.com/settings/security>

paste it in your GitHub projects Secrets actions with secret name: DOKERHUB\_TOKEN:

<https://github.com/yugants/recipe-app-api/settings/secrets/actions>

create another secret with: DOCKERHUB\_USER

and save docker username in it.

Create a requirements.txt and type versions of Django and djangorestframework

.dockerignore – ignored by docker

.gitignore – ignored by git

**Note:**

Make sure Docker-Desktop is running before executing any docker-compose command.

[What's the difference between Docker and Python virtualenv?](https://stackoverflow.com/questions/50974960/whats-the-difference-between-docker-and-python-virtualenv)

A virtualenv only encapsulates Python dependencies. A Docker container encapsulates an *entire OS*. With a Python virtualenv, you can easily switch between Python versions and dependencies, but you're stuck with your host OS. With a Docker image, you can swap out the entire OS - install and run Python on Ubuntu, Debian, Alpine, even Windows Server Core. There are Docker [images](https://hub.docker.com/_/python/) out there with every combination of OS and Python versions you can think of, ready to pull down and use on any system with Docker installed.

Python virtual environment will "containerize" only Python runtime i.e. python interpreter and python libraries whereas Docker isolates the whole system (the whole file-system, all user-space libraries, network interfaces). Therefore, Docker is much closer to a Virtual Machine than virtual environment.

# [Difference between AbstractUser and AbstractBaseUser in Django?](https://stackoverflow.com/questions/21514354/difference-between-abstractuser-and-abstractbaseuser-in-django)

The [documentation](https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#extending-the-existing-user-model) explains this fully. AbstractUser is a full User model, complete with fields, as an abstract class so that you can inherit from it and add your own profile fields and methods. AbstractBaseUser only contains the authentication functionality, but no actual fields: you have to supply them when you subclass.

What are Docker Containers?

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers – images become containers when they run on [Docker Engine](https://www.docker.com/products/container-runtime). Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

What is Dockerfile?

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

What is docker-compose?

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. Compose works in all environments: production, staging, development, testing, as well as CI workflows.

Using Compose is basically a three-step process:

* Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
* Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
* Run docker compose up and the [Docker compose command](https://docs.docker.com/compose/#compose-v2-and-the-new-docker-compose-command) starts and runs your entire app. You can alternatively run docker-compose up using Compose standalone (docker-compose binary).

**DockerFile:**

The Dockerfile is used to build our image, which contains a mini Linux Operating System with all the dependencies needed to run our project.

In Dockerfile we performed:

* Copy operation from local to docker
* Created a virtual env. on docker
* Created a user for docker named “django-user”
* Set the python env. path

Create an app folder in main DIR and run:

docker build .

**Docker-compose:**

We are initializing docker compose for our project, after configuring run:

docker-compose build

**PUT VS PATCH:**

In put() we need update all the content and in patch we can update certain fields also.

**Linting:**

That’s right! Linting is used to ensure code is formatted correctly. It highlights issues like invalid tab spacing and line lengths.

**We will use “flake8” for linting**

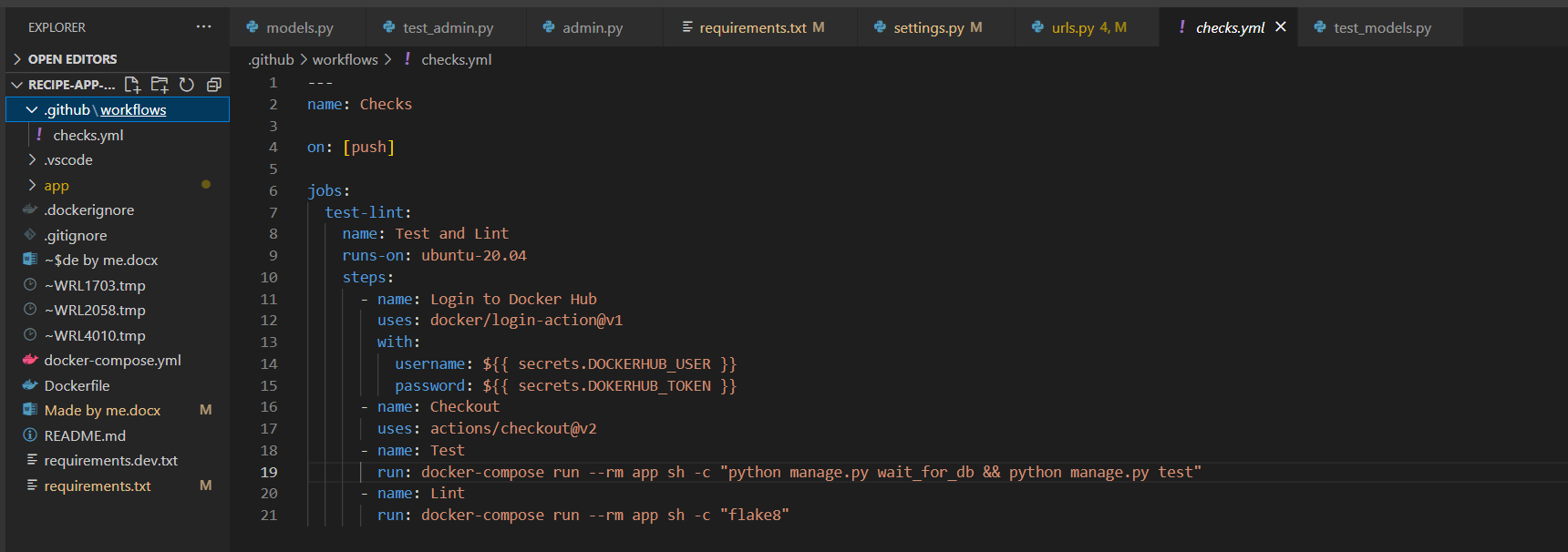
Now make changes in docker-compose, Dockerfile and create .flake8

To check if flake8 is installed correctly or not type, check it before pushing to repo:

docker-compose run --rm app sh -c "flake8"

**checks.yml:**

In this file we configure the pipeline for our code with GitHub actions.



**Create Project:**

docker-compose run --rm app sh -c "django-admin startproject app ."

**Run Server in Docker:**

docker-compose up

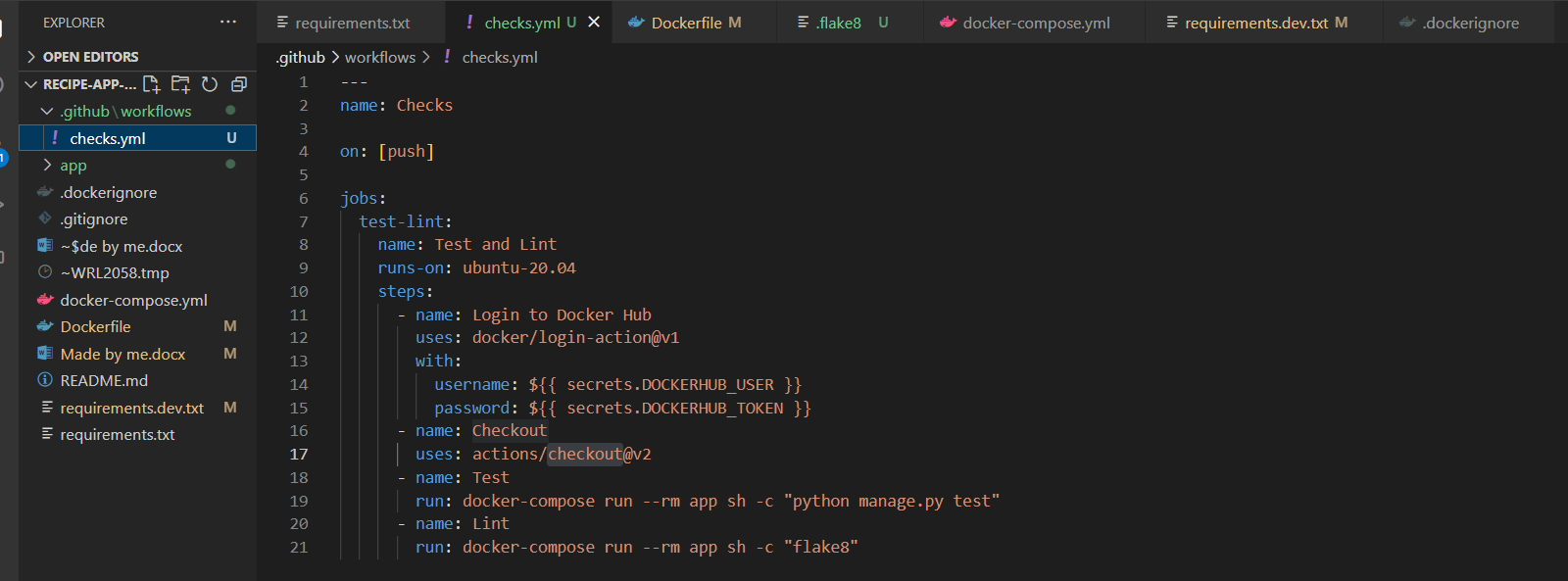
**GitHub Actions:**

* Automation tool
* Similar to Jenkins, GitLab CI/CD
* Run jobs when code changes
* Automate tasks

Our trigger for CI/CD is push operation on github.

**For Configuring CI/CD:**

Make a folder in main file: .github/workflows -> in it create a file -> checks.yml:



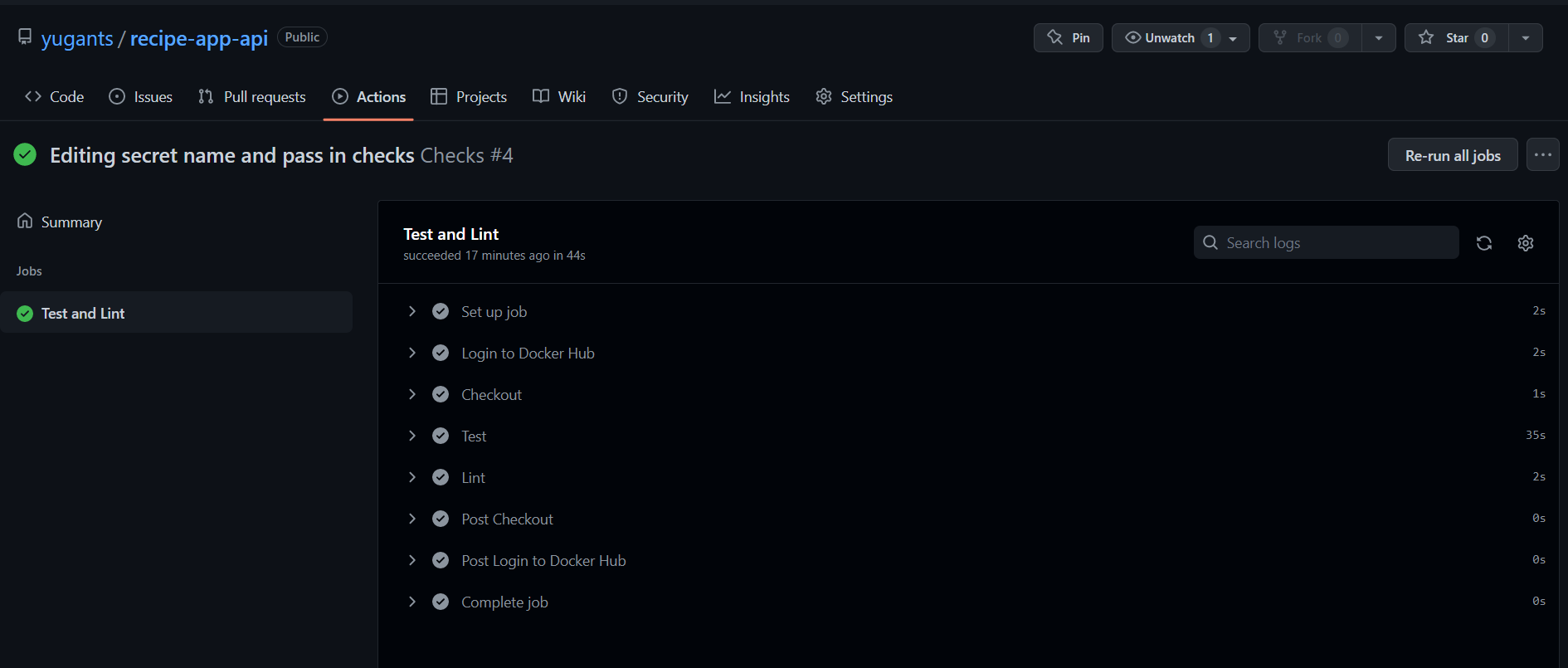
Then run the in shell to check:

docker-compose run --rm app sh -c "python manage.py test"

We are making a CI/CD pipeline in above SS:

on: execution method

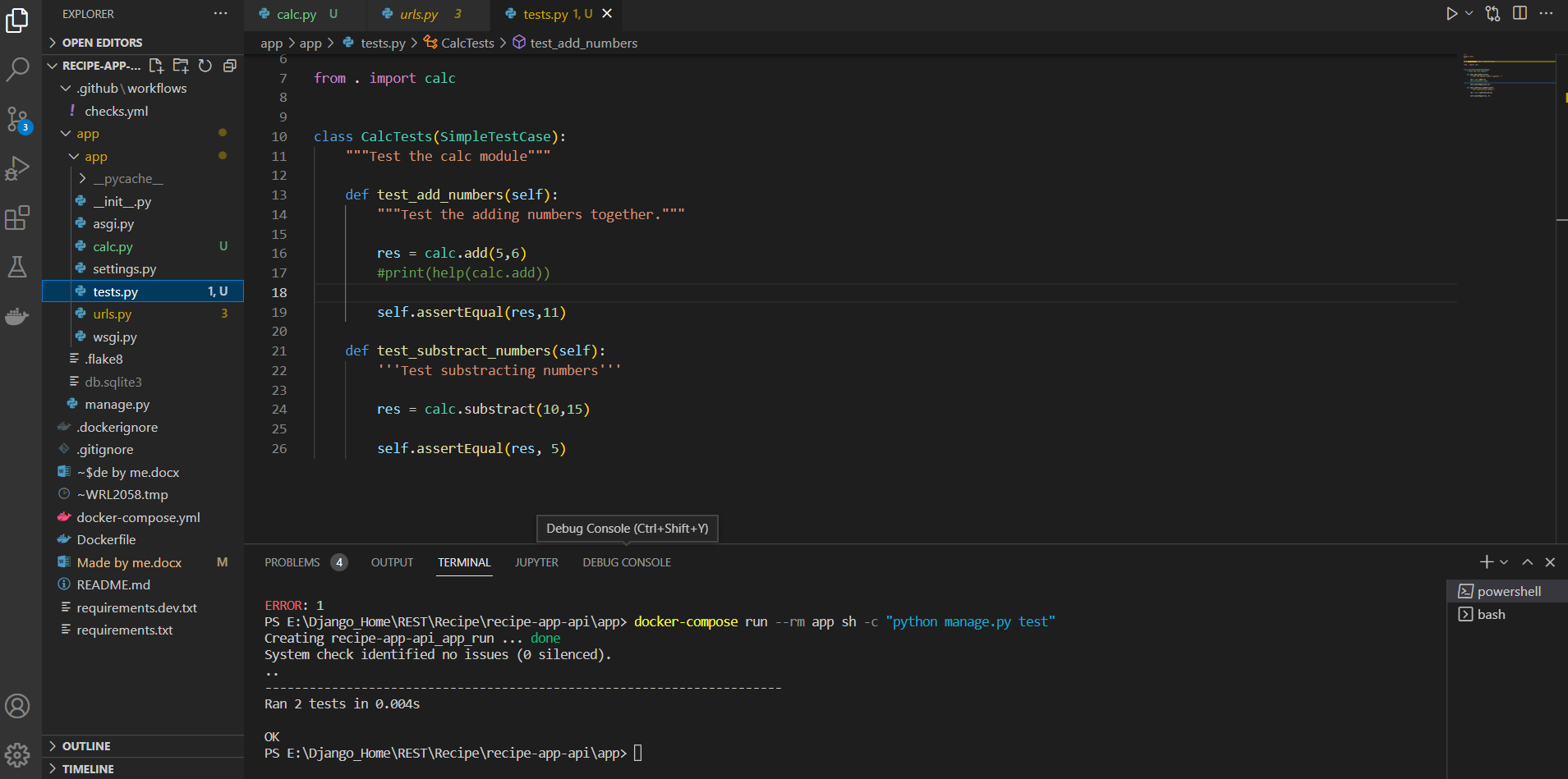
steps: we have-

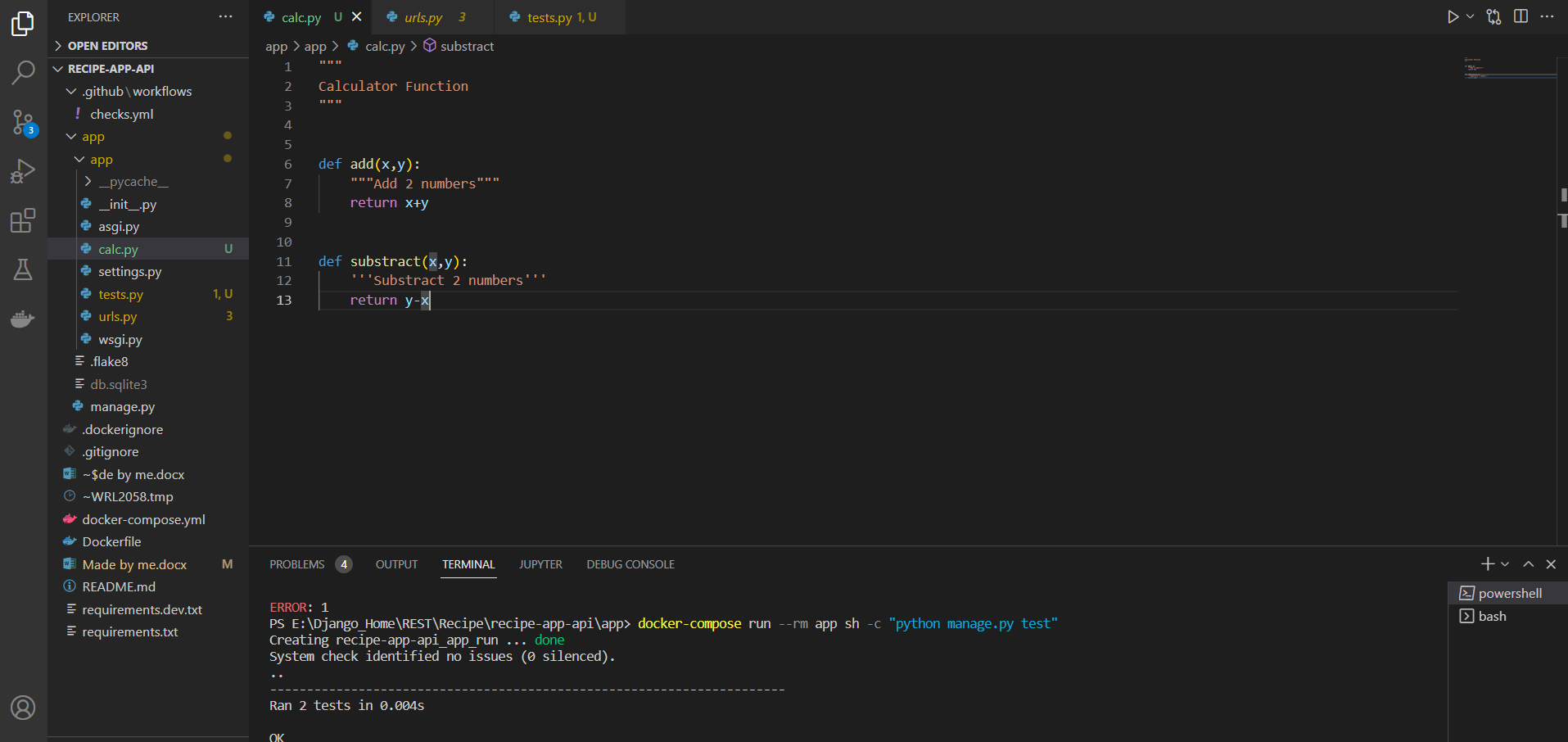


**Tests in Django:**

Every time we run a test, it automatically creates a database, migrate the changes and delete the database at the end for each test we run.

Create a file named tests.py or a folder named test or create a dir named tests and in it define an \_\_init\_\_() file, start files in module with test prefix and also start function name with test\_, we cannot use both of them either tests/ or tests.py





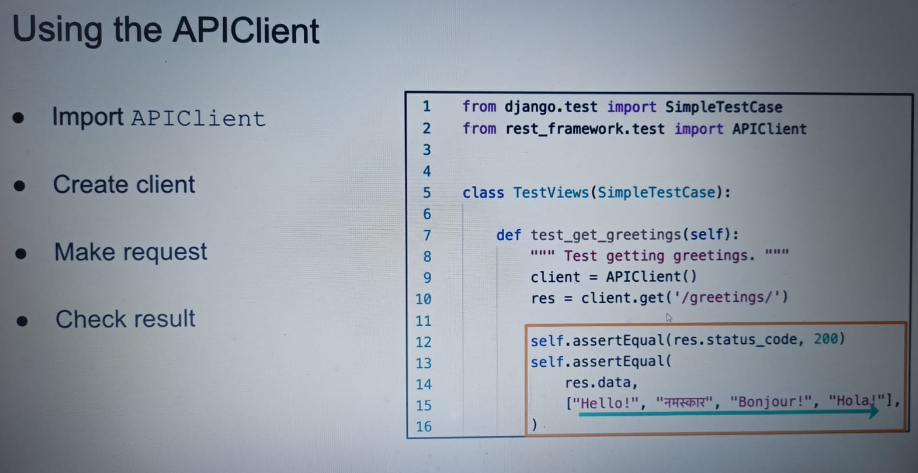
**Flow of Project:**

* Create test for a URL by passing its path
* Run the test, it should fail, because we haven’t implemented functionality yet.
* Implement functionality [url, view, serializers]
* Run the test again, this time they should pass

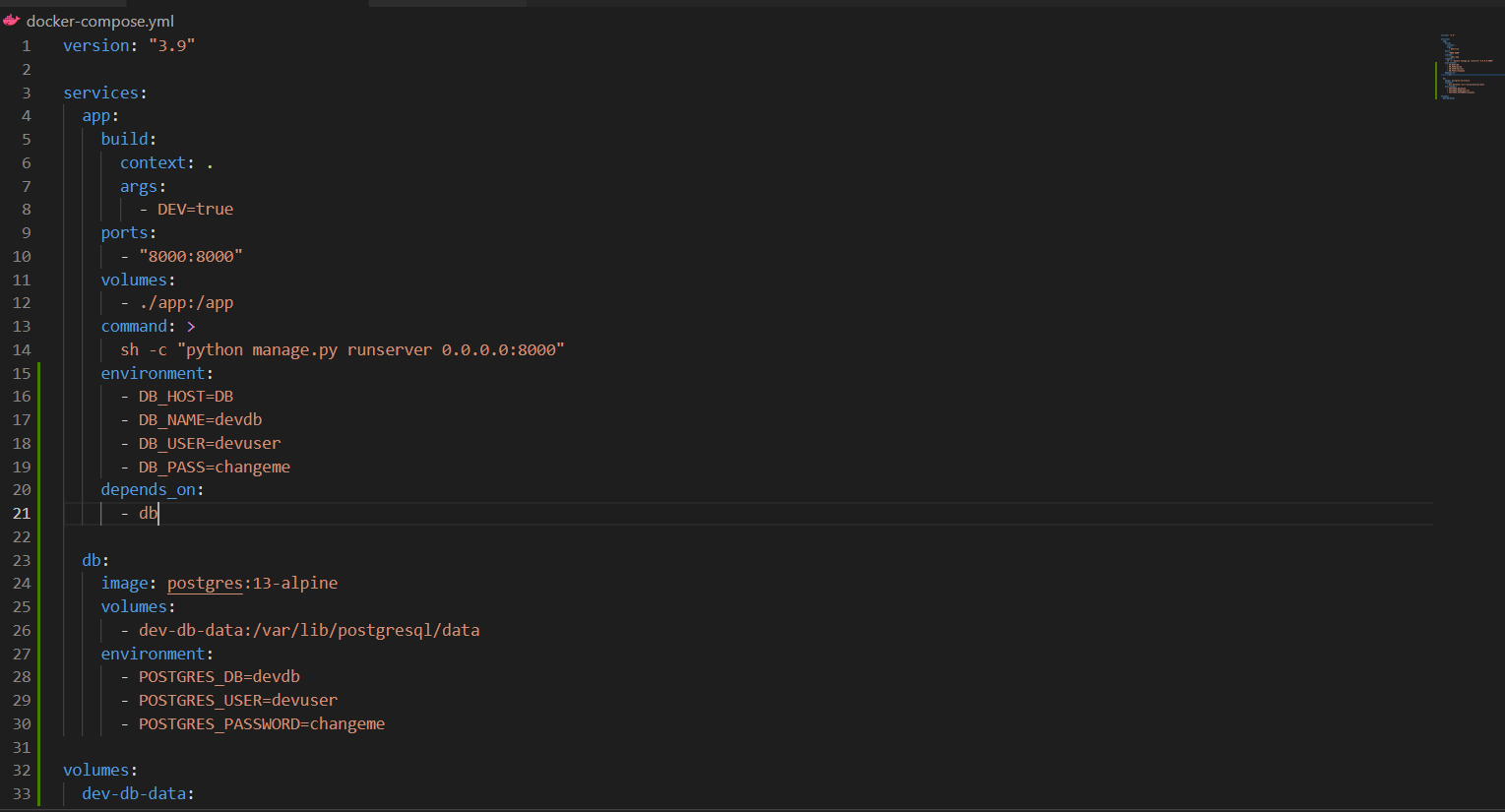
## What Is Mocking?

A [mock object](https://en.wikipedia.org/wiki/Mock_object) substitutes and imitates a real object within a [testing environment](https://realpython.com/python-testing/). It is a versatile and powerful tool for [improving the quality of your tests](https://realpython.com/python-cli-testing/#mocks). One reason to use Python mock objects is to control your code’s behaviour during testing. For example, if your code makes [HTTP requests](https://realpython.com/python-requests/) to external services, then your tests execute predictably only so far as the services are behaving as you expected. Sometimes, a temporary change in the behaviour of these external services can cause intermittent failures within your test suite. Because of this, it would be better for you to test your code in a controlled environment. [Replacing the actual request with a mock object](https://realpython.com/testing-third-party-apis-with-mocks/) would allow you to simulate external service outages and successful responses in a predictable way.

**API Client:**

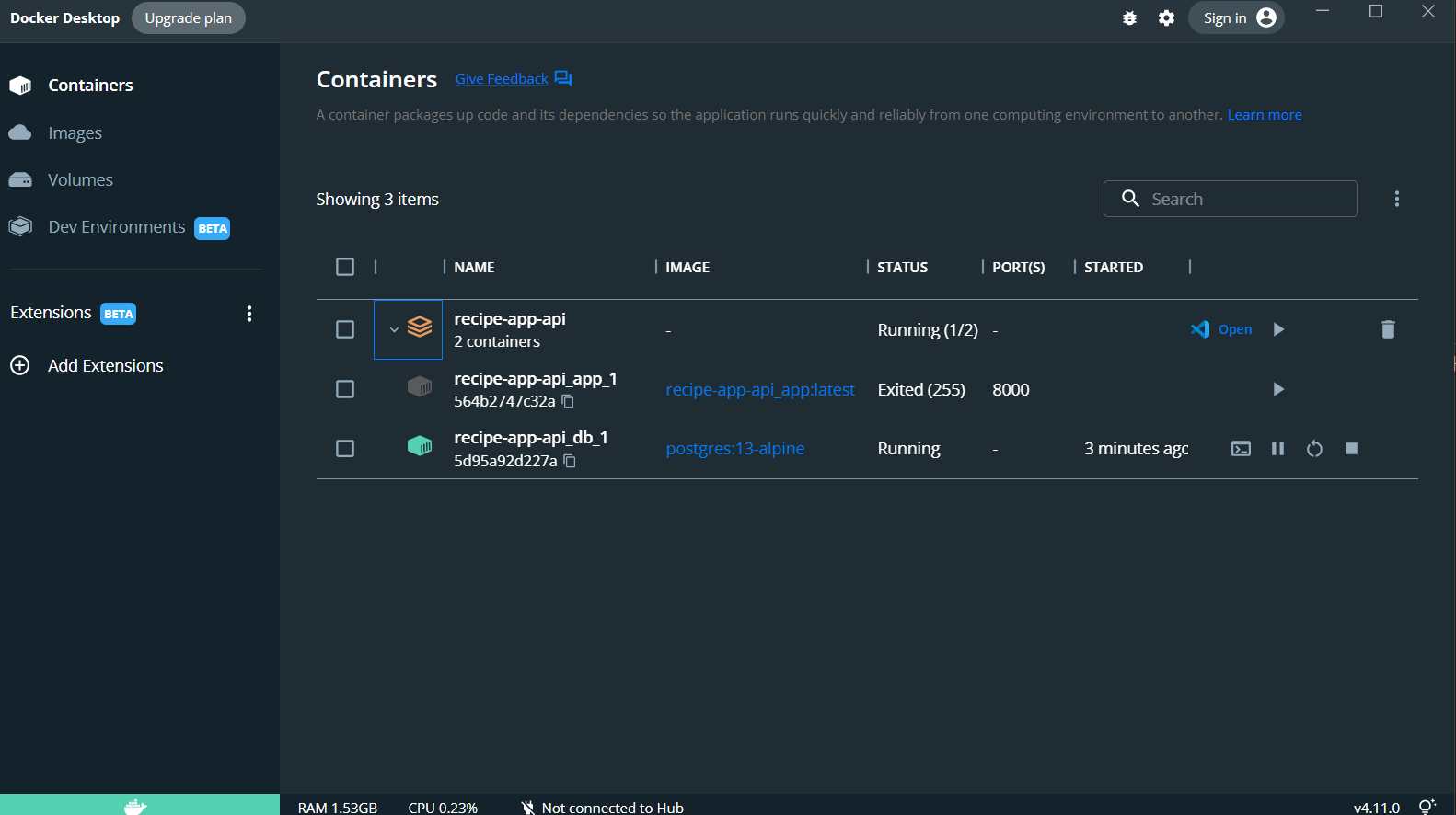


Postgres Configuration in docker-compose:



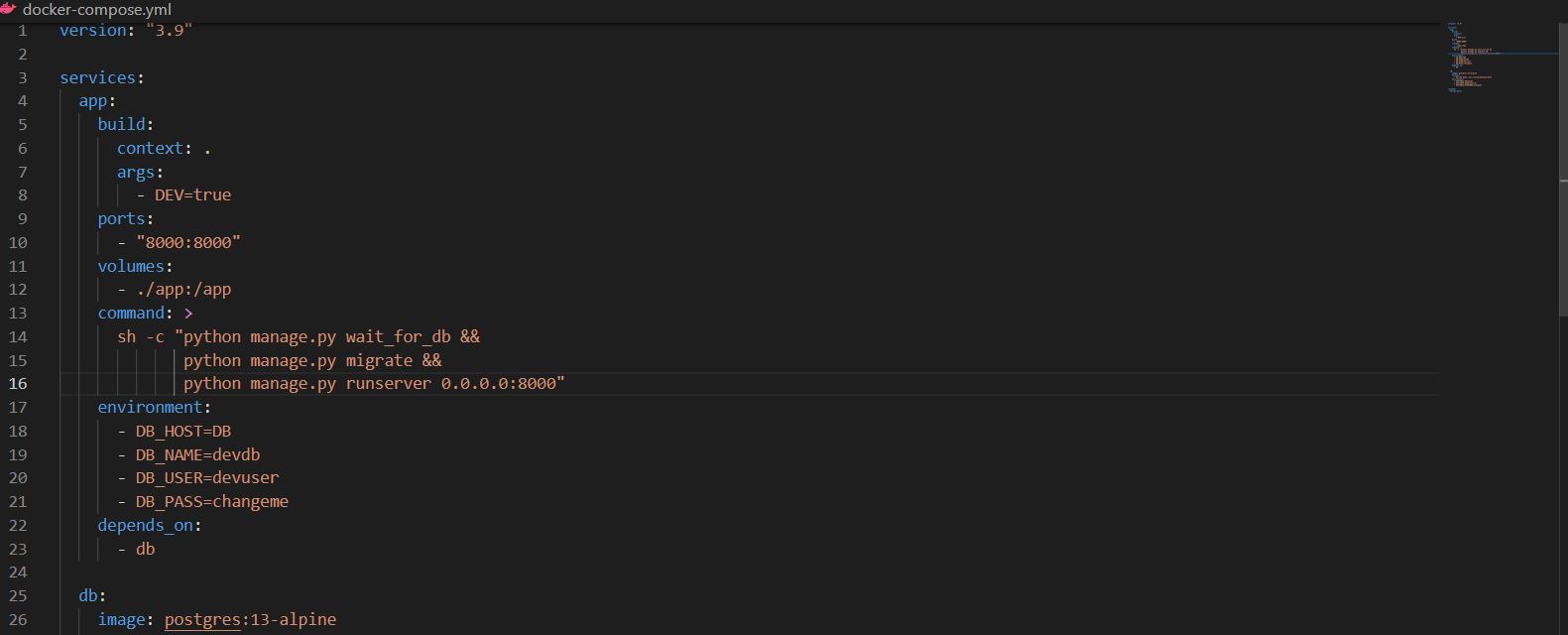
**Wait\_for\_db:**

We are creating a custom command in Django which we will be able to run from manage.py, When are executing two containers on docker:



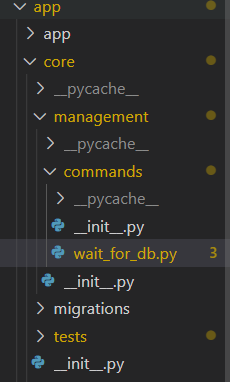
1. For Django
2. For Postgres

We need Django to wait for postgres container to initialize and run sql properly for that we need to introduce this wait\_for\_db command otherwise they both will start at same time and Django container will throw an error that database is not running because Django container is dependent on the postgres service.



See line 22.

Structure of creating a Django command, now we can run it with manage.py:-



app – project\_name

core – app\_name

core => management => commands => wait\_for\_db

create \_\_init\_\_.py in all of the above directories.

**Always create a custom user model:**

* We can set email as a login field
* Making changes later to it will be easy

**Docker Volumes:**

What does volume mean in Docker?

What are Docker Volumes? Docker volumes are **file systems mounted on Docker containers to preserve data generated by the running container**. The volumes are stored on the host, independent of the container life cycle. This allows users to back up data and share file systems between containers easily.

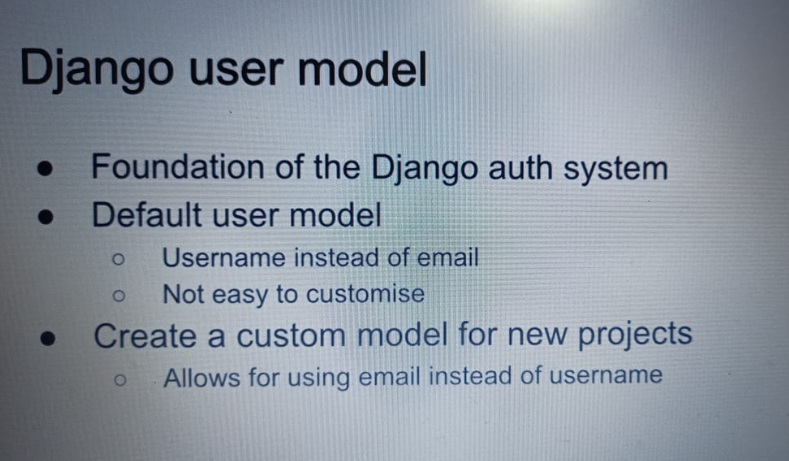
docker volume ls

docker volume rm <volume\_name>

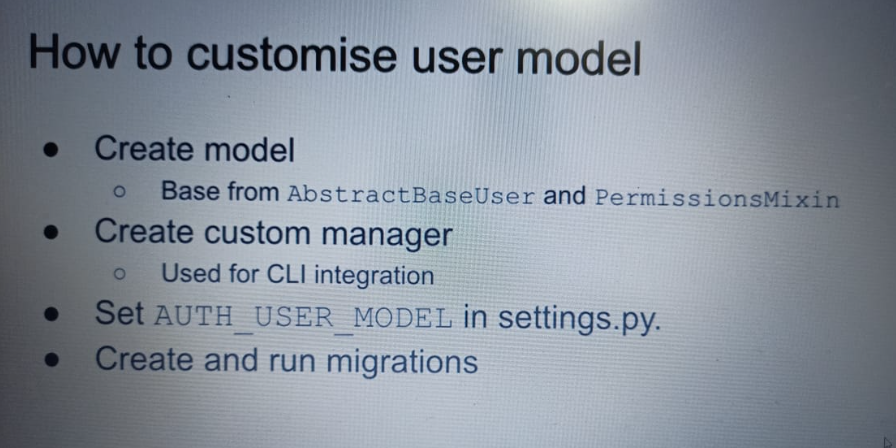
**Command for migrate:**

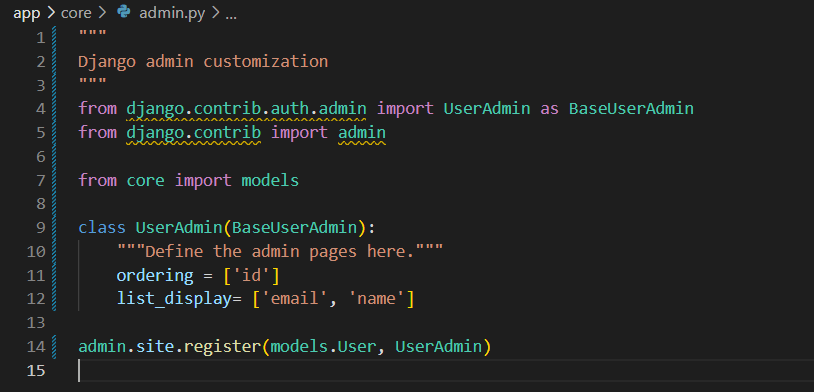
docker-compose run --rm app sh -c "python manage.py wait\_for\_db && python manage.py migrate"

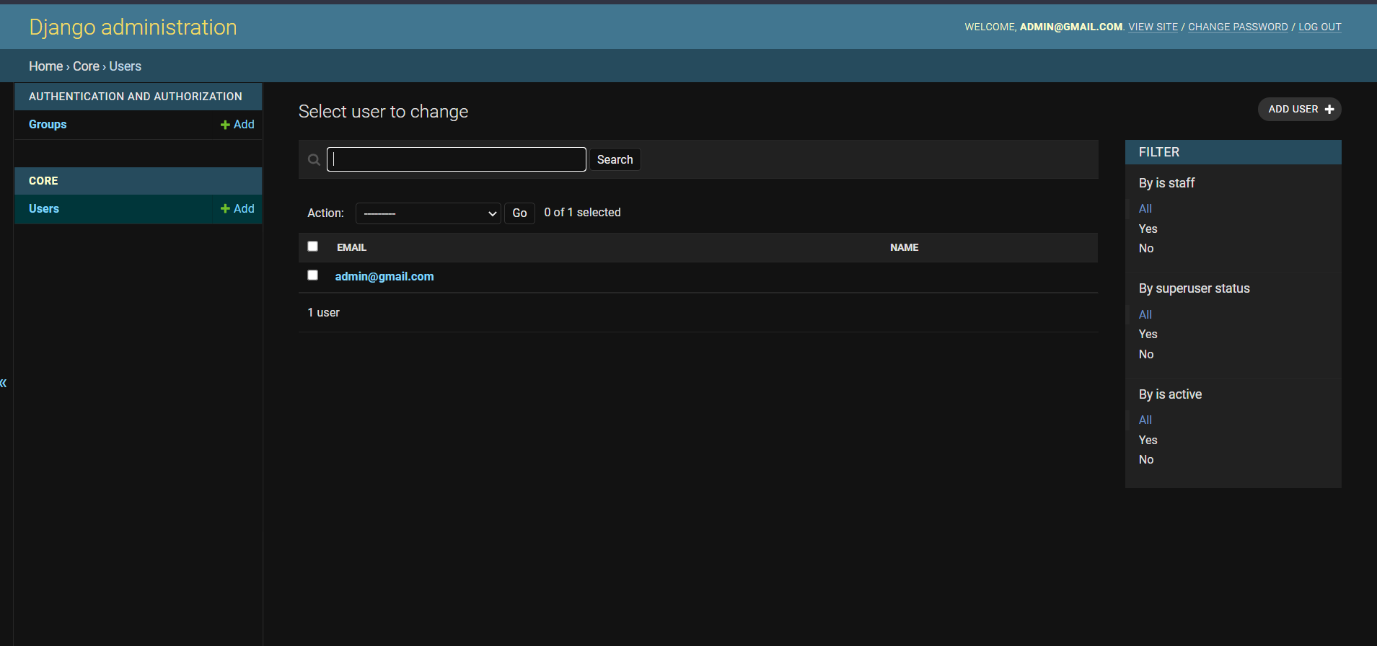
**Problem with default User Model:**

****

**Creating Custom Admin and User model:**



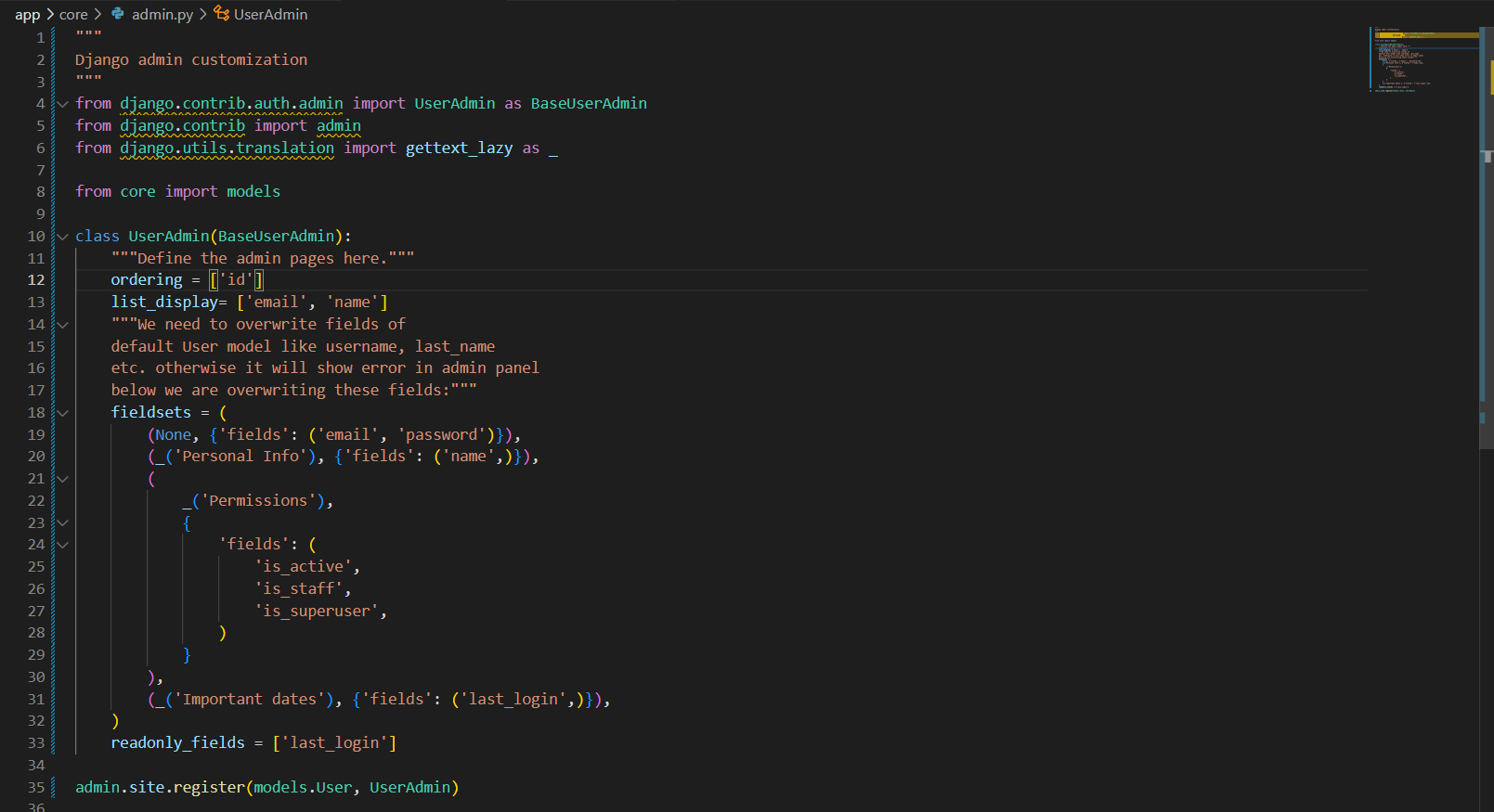


When I ran admin panel:

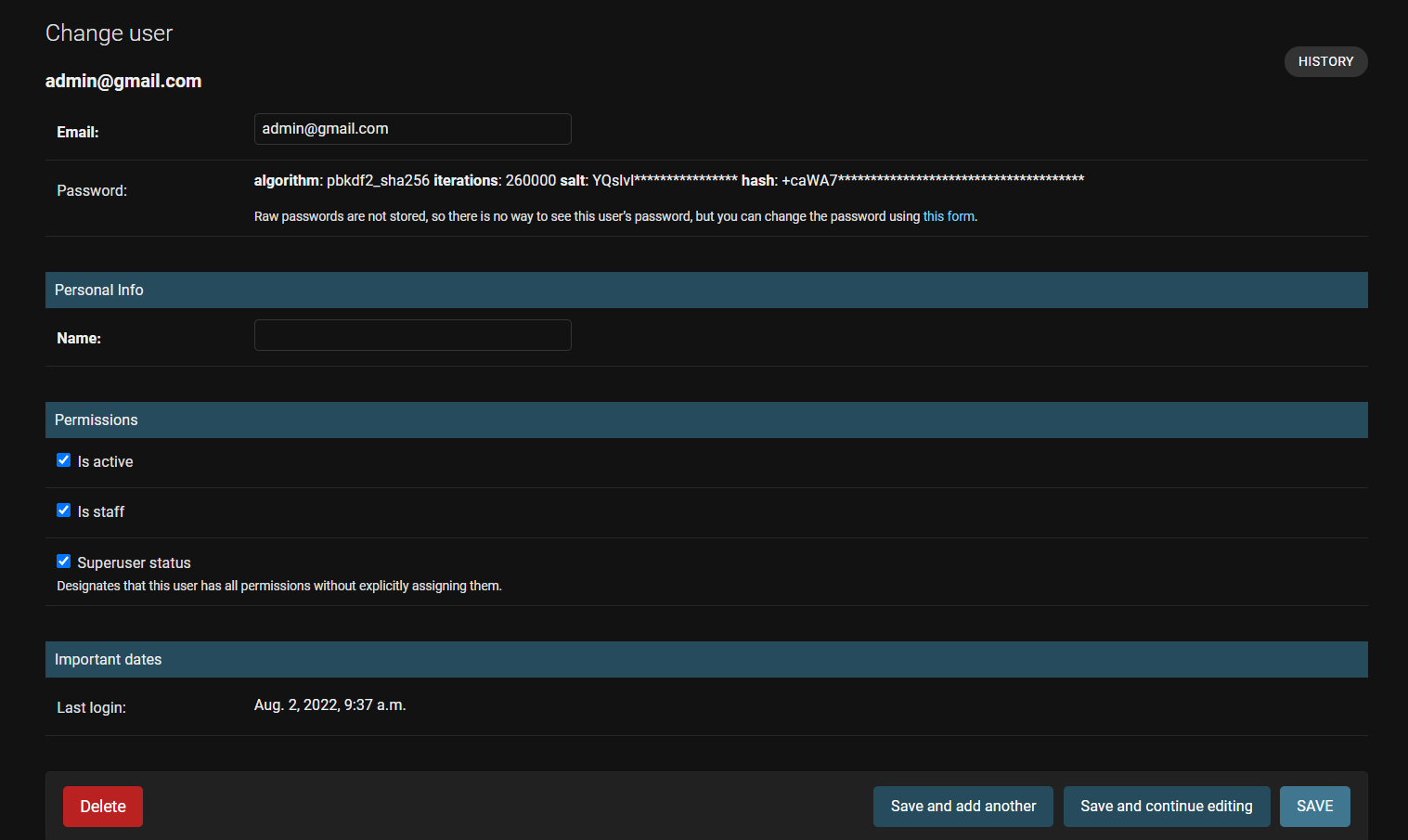
But if I clicked on the user, its showing:



Overwritten the Fields in User model:



Now admin panel of a user:

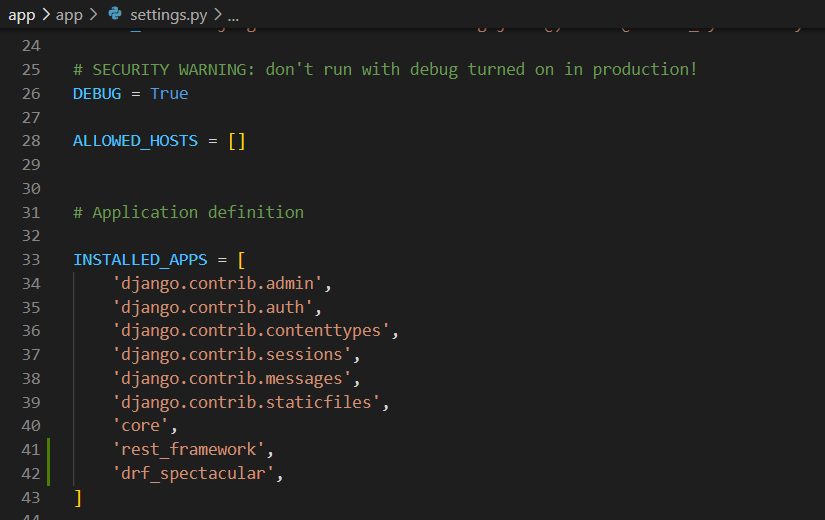


**Automatic Documentation in DRF:**

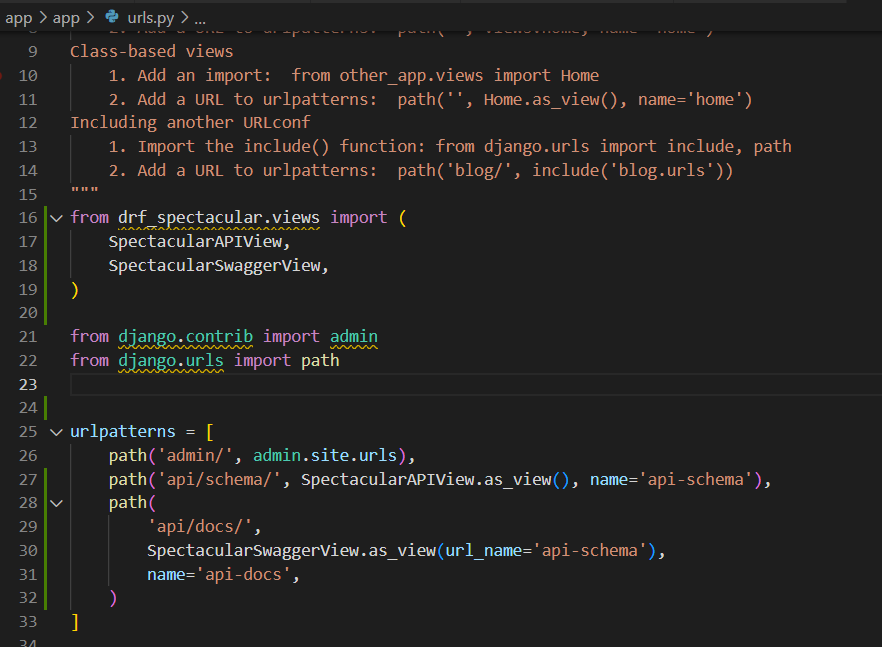
pip install drf\_spectacular

It generates schema file.

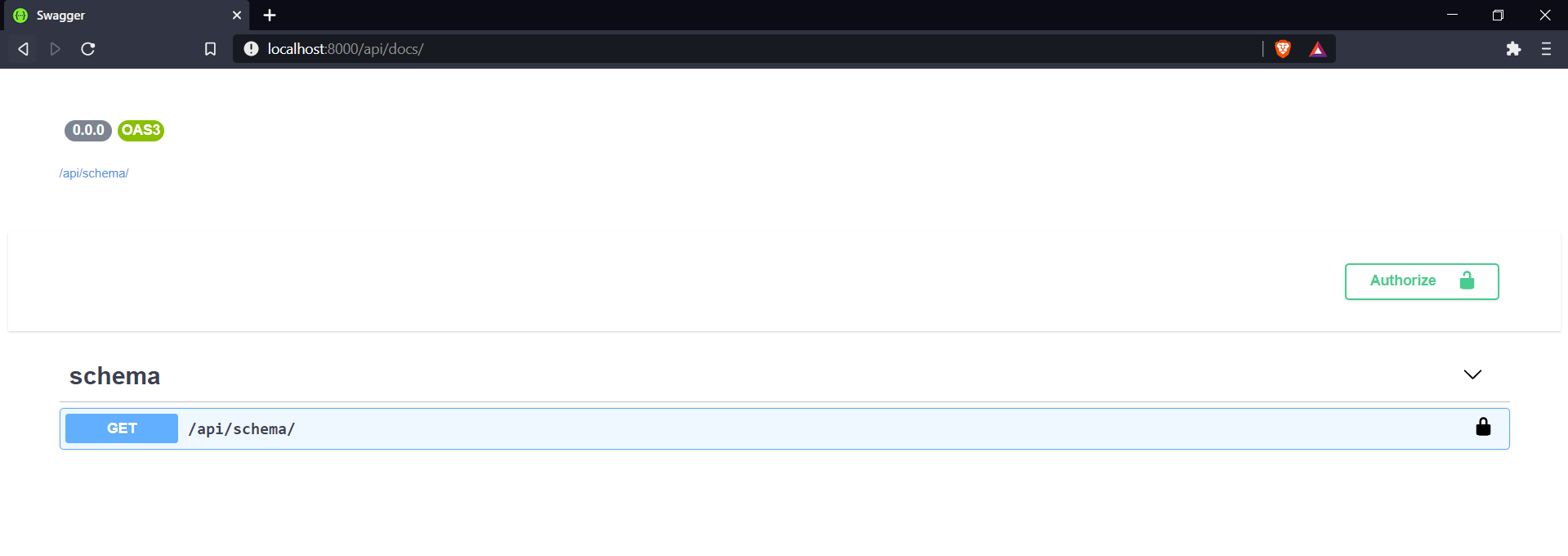
**drf-spectacular Configuration:**



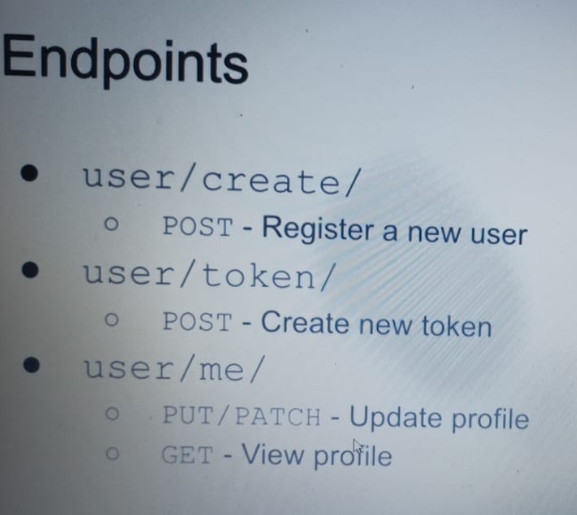




After that It will look like:

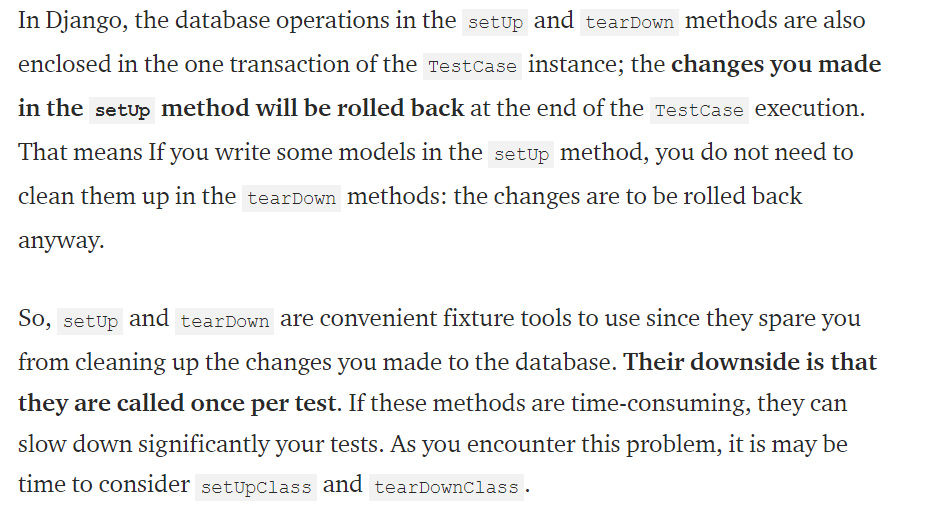


User API:



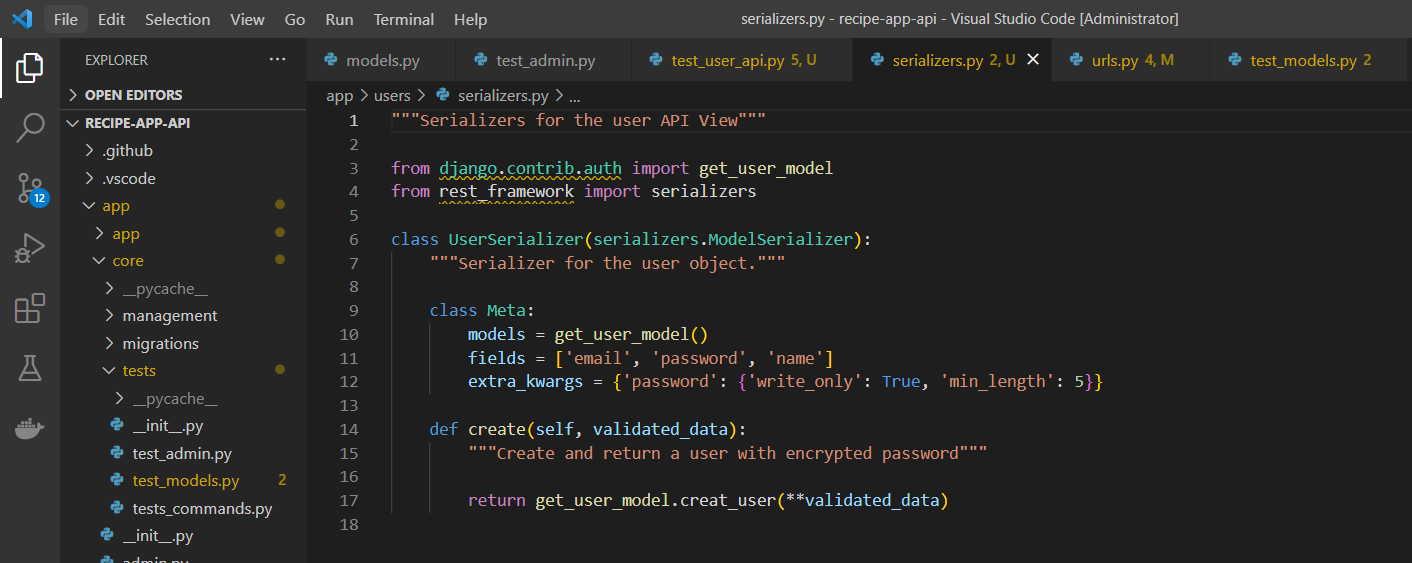
For that I have created Users app

**SetUp() in Test class:**



User Serializer:

With min\_length pass and write only field:

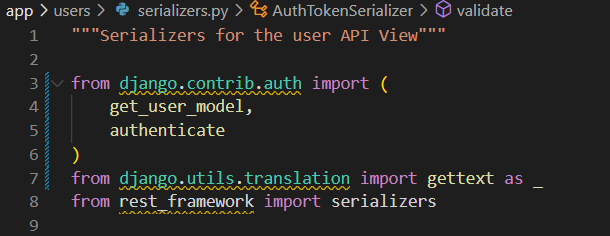


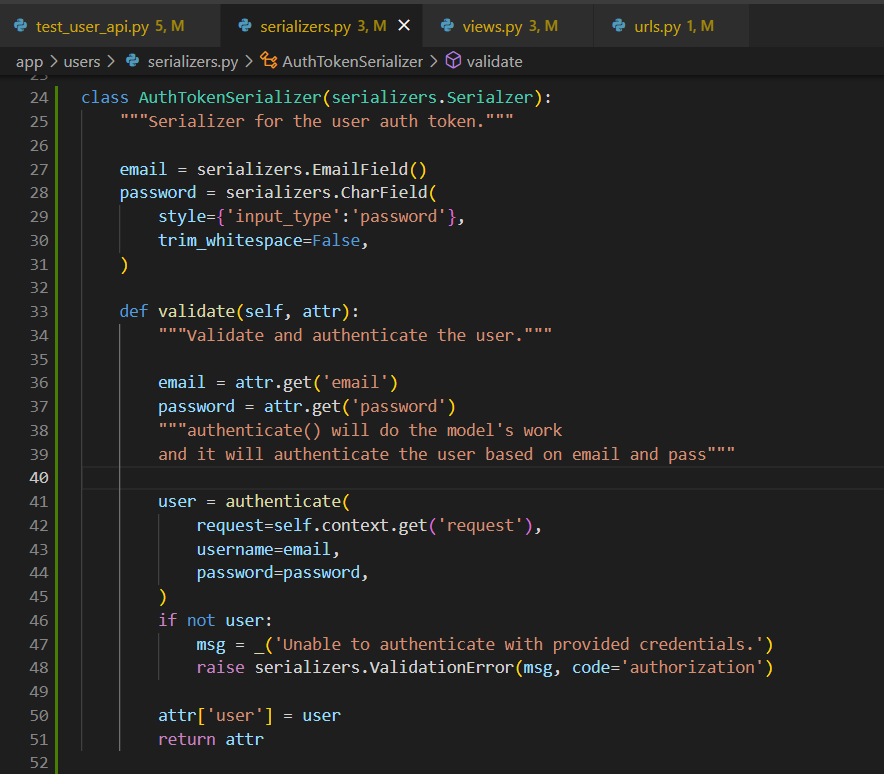
**Token Authentication:**

Include in APPS in settings.py:

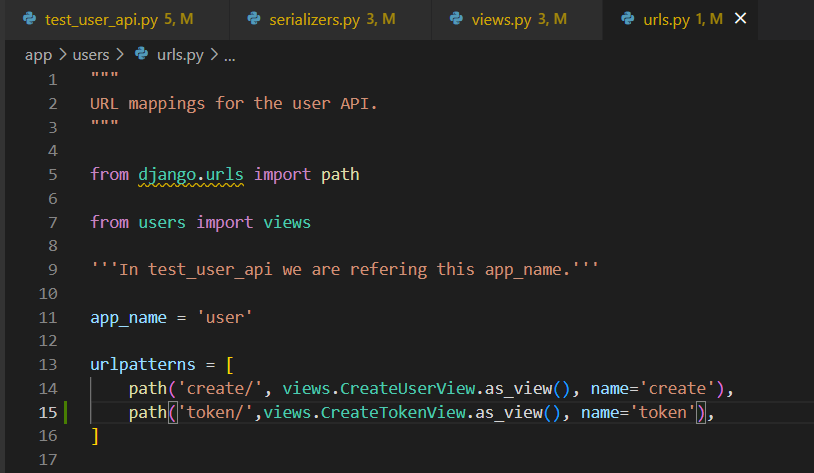
'rest\_framework.authtoken',

Serializers.py configuration:

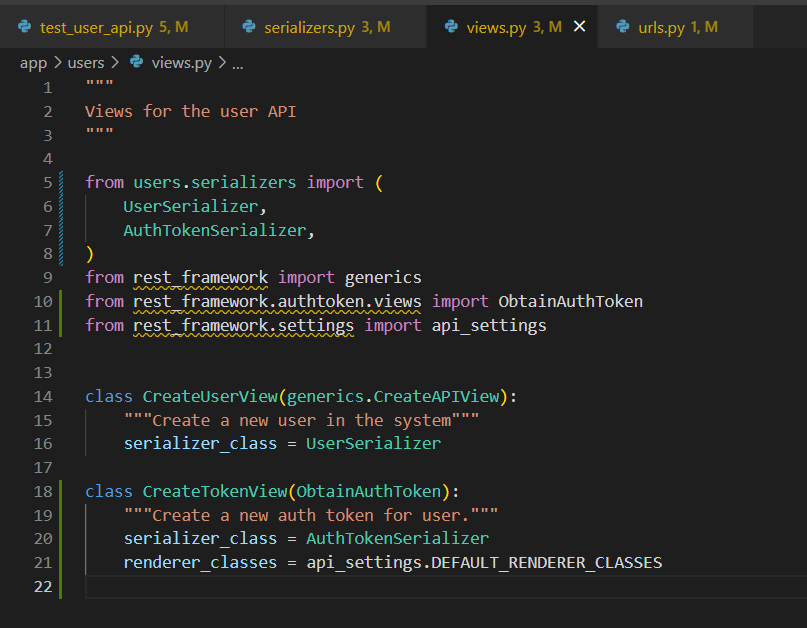




Urls.py:



Views.py;



**Testing Swagger API:**

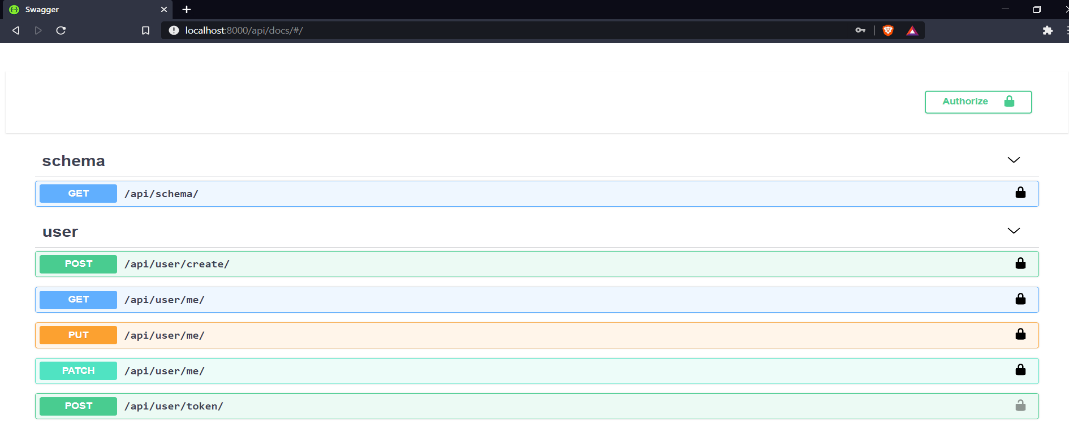
Write:

docker-compose up

in the URL tab of browser type:

<http://localhost:8000/api/docs/>

You will see:



* First create a user
* Then create a token with the user
* On top right, click on Authorize and in Token section type
* Token <token\_value>
* Now we can see <me> urls