[Build a Backend REST API with Python & Django - Advanced](https://www.udemy.com/course/django-python-advanced/)

1.2.

Budujemy API dla komputerowego dziennika. Będzie tam USER AUTHENTICATION, CREATING OBJECTS, FILTERING and ORDERING oraz UPLOADING I VIEWING IMAGES używając DJANGO, dowiemy się przy okazji jak to upgradować.

2.

Apka będzie zbudowana tak.

Budujemy backend w REST API, co pozwoli nam zrobić ją w wersji mobilnej bądź przeglądarkowej. Nie będziemy się bawić we Frontend, zrobimy sobie jedynie BACKEND, za to w pełni funkcjonalny.

FEATURES:

19 API ENDPOINT,

USER AUTHENTICATION,

BROWSABLE ADMIN INTERFACE,

BROWSABLE API(SWAGGER UI).

TECHNGOLOGIE:

Użyjemy: PYTHON(Foundation naszego API), DJANGO (HANDLES -> URL MAPPINGS, ORM, Admin Site),

DRF = Addon do DJANGO do RESTu

POSTGRES

DOCKER

SWAGGER UI

BUDOWA APLIKACJI DJANGO:

App – Django Project

App/core- > Kod udostępniany pomiędzy wiele appek(BAZA DANYCH

App/user -> User registration i auth tokeny

App/coś tam -> Tu będzie mięcho aplikacji

Co potrzebujemy?

PYCHARM, GIT, DOCKER DESKTOP

**SEKCJA 5. PROJECT SETUP.**

Pytanko, czemu używać dockera do produkcji? Daje nam spójne środowisko developmentu i produkcji aplikacji. Używamy tego samego środowiska do devu jak i do produkcji. Nie bez znaczenia jest też łatwiejsza współpraca z innymi. Nie ma wiadra błędów bo inny DB, Python czy SDKi.

Docker eliminuje wszystkie te problemy.

Dodatkowo docker wychwytuje wszystkie zależności w kodzie. Nic nie trzeba instalować manualnie.

Łatwiejsze jest sprzątanie po sobie :D. Możemy po skończeniu pracy nad tym projektem po prostu usunąć wszystkie te pliki. Są zachowane w Dockerze.

Jak będziemy używać DOCKera?

Zdefiniujemy Dockerfile, następnie skonfigurujemy DOCKER COMPOSE, a nim będziemy wykonywać wszystkie komendy

12. Tworzymy nowy GitHub projekt, następnie otwieramy terminal, gdzie bierzemy sklonowany kod do naszego terminala. W miejscu w którym chcemy go osadzić:

Git clone link 🡪 Następnie idziemy na hub,docker.com : account settings 🡪 security 🡪 Access Tokens 🡪 Create new token 🡪 (Read write and delete option). Dostajemy token, który trzeba skopiować do naszego GITa. Wchodzimy w ten projekt 🡺 Settings 🡪 Secrets 🡪Dependabot 🡪 Create new secret x2. DOCKERHUB USER i DOCKERHUB SECRET 🡪 Do tego drugiego wklejamy ten ów wyżej dostany token

ACCESS TOKEN DESCRIPTION = REST API App

ACCESS PERMISSIONS =Read, Write, Delete

**To use the access token from your Docker CLI client:**

1. Run docker login -u papabear928

2. At the password prompt, enter the personal access token.

Dzień 2.

13. Docker i Django

Połączymy Django i Dockera tugedo 🡪 Zaoszczędzi to nam mnóstwo czasu, zapisze wszystkie zmiany w kodzie, ułatwi sprzątanie, etc.

Wady ? 🡪 trudniejsze w użyciu zintegrowane funkcje oraz nie da się użyć VSCode

Używamy do ruszenia. Docker-compose w takiej postaci :

**Docker-compose run –rm app sh –c „python manage.py collectstatic”**

14. Uruchamianie tego wewnątrz.

W miejscu powyżej w terminalu po prostu zróbmy code . 🡪 To uruchomi nam VSC. Albo po prostu uruchomić to manualnie i zacząć projekt.

Tworzymy plik requirements. Txt i co tam wpisujemy?

Django >= 4.0.0

djangorestframework =>3.12.3,<3.13

Na razie to tak zostawiamy, później sobie wszystko uaktualnimy

15. Dockerfile.

Tworzymy dockerfile i zapisujemy w nim. Najpierw nazwę naszego API,

FROM python: 3.10.2-alpine3.16

Alpine to lekki linux, idealny do tego =🡺 PYTHONBUFFERED ma nam dać logi natychmiast po starcie. Kopiujemy requirements, aplikacje, wyznaczamy workdir i serwer

ENV PYTHONUNBUFFERED 1

COPY ./requirements.txt /tmp/requirements.txt

COPY ./app /app

WORKDIR /app

EXPOSE 8000

FROM python:3.9-alpine3.13

LABEL maintainer="Klapaucius1000"

ENV PYTHONUNBUFFERED 1

COPY ./requirements.txt /tmp/requirements.txt

COPY ./app /app

WORKDIR /app

EXPOSE 8000

I teraz komenda uruchamiająca:

RUN python -m venv /py && \

    /py/bin/pip install --upgrade pip && \

    /py/bin/pip install -r /tmp/requirements.txt && \

    rm -rf /tmp && \

    adduser \

        --disabled-password \

        --no-create-home \

        django-user

Chcemy aby, nasz obraz był jak najlżejszy, dlatego robimy komendę RUN w jednej komendzie za pomocą &&\

Uruchamiamy i tworzymy w niej VENV, od razu upgradujemy PIPA i instalujemy requirements, w linijce 4 usuwamy TEMPORARki(opcjonalne), w Piątej i dalej dodajemy no wiadomo co.

**NIE URUCHAMIAMY APLIKACJI NA ROOT USER, bo atakujący uzyska nam dostęp do całego DOCKERA!**

ENV PATH="/py/bin:$PATH"

Co to robi? Apdejt naszego ENV wewnątrz Image

USER django-user

Końcowa linijka, zawsze taka powinna być. Zawsze uruchamiamy docker Image jako django user

Tworzymy DOCKER IGNORE(**.dockerignore**), a w nim?

# Git

.git

.gitignore

# Docker

.docker

# Python

app/\_\_pychache\_\_/

app/\*/\_\_pychache\_\_/

app/\*/\*/\_\_pychache\_\_/

app/\*/\*/\*/\_\_pychache\_\_/

app/\*/\*/\*\*/\_\_pychache\_\_/

.env/

.venv/

venv/

No i tworzymy teraz nasz docker image w wierzu polceń.

C:\Users\karol\Desktop\DJANGO\Docker APP V2>docker build .

I działa()

DOCKER COMPOSE CONFIGURATOR.

Tworzymy plik docker-compose.yaml a w nim:

version: '3.9'

services:

  app:

    build:

      context: .

    ports:

      - "8000:8000"

    volumes:

      - ./app:/app

    command: >

      sh -c "python manage.py runserver 0.0.0.0:8000"

To je syntax dockera

Mamy tam wersję pajtona i main block. Po kolej w context deklarujemy naszą apkę, w ports port do którego się mamy odwoływać . We volumes co, a command to ułatwienie, żeby tego wiecznie nie poprawiać

Zainstujemy FLAKE 8, aby wykrywało nam błędy w kodzie

**Docker-compose run --rm app sh –c „flake8”**

Do testów będziemy używali pytesta.

**Docker-compose run --rm app sh –c „python manage.py test”**

KONFIGURACJA tego:

Zaczynamy od utworzenia requirements.dev.txt, a w nim:

flake8>=3.9.2,<3.10

A dlaczego plik dev. Txt? Bo ten plik będzie się uruchamiał jedynie kiedy uruchomimy serwer developerski

Żeby dopełnić konfiguracji wracamy do docker-compose.yaml i dokonujemy w nim paru zmian:

  context: .

      args:

        - DEV=true

Dodajemy args i dev równa się true.

Wracamy do dockerFile i :

COPY ./requirements.dev.txt /tmp/requirements.dev.txt

A nad run dodajemy to:

ARG DEV=false

Teraz w ramach RUN dokonujemy pewnych zmian w kodzie. Wprowadzamy instrukcje warunkową:

/py/bin/pip install -r /tmp/requirements.txt && \

    if [ $DEV = "true" ]; \

        then /py/bin/pip install -r /tmp/requirements.dev.txt ; \

    fi && \

Teraz wyłączymy sekcje z pod działania FLAKE8, bo w takim django np. będzie masa błędów. A na co nam to?

Wewnątrz folderu app tworzymi plik:

.flake8 a w nim dodajemy takie coś:

[flake8]

exclude =

    migrations,

    \_\_pychache.py,

    manage.py,

    settings.py

Zapisujemy i : docker-compose run –rm app sh –c „flake8”

Niby działa, a jednak nie. Jebać to. Później się naprawi

TWORZYMY PROJEKT DŻANGO:

**docker-compose run --rm app sh -c "django-admin startproject app ."** i odpalamy za pomocą **docker compose up**

**KOLEJNA SEKCJA:6. GIT HUB ACTIONS**

Skonfigurujemy sobie GIT HUB ACTIONS, czyli zautomatyzowanie commitów i pushów. Coś jak Jenkins, Travis-Ci, Gitlab. Zautomatyzuje nam to przy okazji deployment, linting i testy. W tym kursie nie poruszymy Deploymentu.

Konfigurujemy: Tworzymy nowy folder: **.github,** w nim **workflows** a w folderze plik **cheks.yml** . W pliku tym zaczynamy od ---, bo to sygnatura pliku yml.

A następnie:

On uruchamia nam push. Następnie za pomocą jobs konfigurujemy test-lin uruchomiony za pomocą ubuntu 20-04(jest tego wiele)

---

name: Checks

on: [push]

jobs:

  test-lint:

    name: Test and Lint

    runs-on: ubuntu-20.04

teraz konfigurujemy steps:

runs-on: ubuntu-20.04

    steps:

      - name: Login to Docker Hub

        uses: docker/login-action@v1

        with:

          username: ${{ secrets.DOCKERHUB\_USER }}

          password: ${{ secrets.DOCKERHUB\_TOKEN }}

      - name: Checkout

        uses: actions/checkout@v2

teraz dodajemy testy i linting za pomocą naszych poznanych już przyjaciół:

**Docker-compose run --rm app sh –c „python manage.py test”**

      - name: Checkout

        uses: actions/checkout@v2

      - name: Test

        run: docker-compose run --rm app sh –c „python manage.py test”

      - name: Lint

        run: docker-compose run --rm app sh –c „flake8”

**Cały proces jest zautomatyzowany i działa!**

25. Konfiguracja Git Hub Actions.

Nie jest to trudne:

**Git add.** , a następnie **git commit -am "Added Actions"** . Na razie nie działa nam hasło i token, ale tym zajmiemy się późni.

**SEKCJA 7. TESTING i TDD.**

Django ma zaimportowany wewnątrz Django TEST. Nauczymy się z niego korzystać. Test jest zbudowany na UNITTEST. Ma wszystko, nawet dummy web browser, symulację autentyfikacji czy nawet własną bazę danych tymczasową. Nie możemy w Django używać równocześnie test.py jak i folderu tests. Wiesz o tym, że Django ma własną testową DB, etc etc.

WRESZCIE MIĘCHO!

**SEKCJA 8 🡪 POSTGRES CONFIGURATION**

To popularna baza danych, w pełni integrowalna z Django i Dockerem. Duży plus to trwałe dane przy użyciu woluminów. Dzięki dockerowi w pełni konfiuguralne środowisko. Możliwa też konfiguracja z sieci(Zrobimy to).

Docker ma dwa serwisy, DATABASE(postgresql) i App(Django). Konfiguracja też jest prosta. Zaraz ją zrobimy.

**33. Add DB Service.**

Otwieramy plik docker-compose.yml. Tutaj zdefiniujemy naszą DB. Bardzo ważne jest, aby indentacja była taka jak przy APP(drugie wcięcie). Wybieramy postgres:13 alpine. Volumes to konfiguracja naszej bazy danych. Wazne, żeby enivroment był taki sam w APP jak i w DB!

    command: >

      sh -c "python manage.py runserver 0.0.0.0:8000"

    environment:

      - DB\_HOST=db

      - DB\_NAME=devdb

      - DB\_USER=devuser

      - DB\_PASS=changeme

    depends\_on:

      - db

  db:

    image: postgres:13-alpine

    volumes:

      - dev-db-data:/var/lib/postgresql/data

    environment:

      - POSTGRES\_DB=devdb

      - POSTGRES\_USER=devuser

      - POSTGRES\_PASSWORD=changeme

volumes:

  dev-db-data:

**34-35-36 Conf DB with Django**

Najpierw skonfigurujemy DJANGO, potem dodamy adapter do DB, a na końcu zainstalujemy ważne pakiety co chce je ten koleżka.

1.Django musi wiedzieć, co to za DB(Engine), Hostname(IP albo domena albo nazwa db), PORT(5432) oraz nazwa DB, hasło itp.

2. Potrzebujemy PSYCOPG2, najpopularniejszego adaptera do DJANGO. Jest najlepsza i łatwa do zainstalowania z DOCKEREM. Dlatego wygrywa

35. Instalujemy adaptoer. Zaczynamy w linii 14. Zaraz za PIP install w linijce 14 dajemy to co na czerwone. Tak samo za remove(rm) dodajemy apk del.

RUN python -m venv /py && \

    apk add --update --no-cache postgresql-client && \

    apk add --update --no-cache --virtual .tmp-build-deps \

        build-base postgresql-dev musl-dev && \

    /py/bin/pip install --upgrade pip && \

    /py/bin/pip install -r /tmp/requirements.txt && \

    if [ $DEV = "true" ]; \

        then /py/bin/pip install -r /tmp/requirements.dev.txt ; \

    fi && \

    rm -rf /tmp && \

    apk del .tmp-build-deps &&\

    adduser \

        --disabled-password \

        --no-create-home \

        django-user

W pierwszej czerwonej linijce instalujemy klienta, w drugiej maszynę wirtualną. Ważne żeby tmp-build-deps \ pasowało do ostatniej czerwonej linijki.

Teraz cza wejść do requirements.txt i dodać psycop2

NIE DO KOŃCA DZIAŁA NAM AUTOMATYZACJA Z GITEM. TO NIE DOBRA JEST.

36. Conf DB in DJANGO (Przesiadłem się na Pycharma bo nie mogie tego vsc)

Otwieramy settings w naszej apce:

Importujemy os i zmieniamy DATABASES na takie:

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.postgresql',  
 'HOST': os.environ.get('DB\_HOST'),  
 'NAME': os.environ.get('DB\_NAME'),  
 'USER': os.environ.get('DB\_USER'),  
 'PASSWORD': os.environ.get('DB\_PASS'),  
  
 }

**37. naprawa błędów w działaniu bazy danych**

Czasami użycie DEPENDS ON zapewnia start naszej DB. Ale nie wiemy czy serwis też ruszył. Może być tak, że aplikacja ruszy, ale bez bazy danych i żeby się nie skraszowały dodamy Wait for DB

**39. Piszemy testy dla WAIT\_FOR DB**

(Fajnie, że się nie zapisało)

*'''Test custom Django management commands.'''  
  
from* unittest.mock *import* patch  
  
*from* psycopg2 *import* OperationalError *as* Psycopg2Error  
  
*from* django.core.management *import* call\_command  
*from* django.db.utils *import* OperationalError  
*from* django.test *import* SimpleTestCase  
  
  
@patch('core.management.commands.wait\_for\_db.Command.check')  
*class* CommandTest(SimpleTestCase):  
 *'''Test commands'''  
  
 def* test\_wait\_for\_db\_ready(*self*, patched\_check):  
 *'''Test waiting for DB if DB is ready'''* patched\_check.return\_value = *True* call\_command('wait\_for\_db')  
  
 patched\_check.assert\_called\_once\_with(database=['default'])  
  
  
 @patch('time.sleep')  
 *def* test\_wait\_for\_db\_delay(*self*, patched\_sleep, patched\_check):  
 *'''Test waiting for DB when getting Operational Err'''* patched\_check.side\_effect = [Psycopg2Error] \* 2 + \  
 [OperationalError] \* 3 + [*True*]  
  
 call\_command('wait\_for\_db')  
  
 *self*.assertEqual(patched\_check.call\_count, 6)  
 patched\_check.assert\_called\_with(databases=['default'])

**40. ADD WAIT\_FOR DB.**

Teraz możemy dodać nasze wait for DB. Idziemy do settings.py

Ważnym elementem jest składnia **stdout,** typowa dla dockera i opóźniania

*import* time  
  
*from* django.core.management.base *import* BaseCommand  
*from* django.db.utils *import* OperationalError  
  
*from* psycopg2 *import* OperationalError *as* Psycopg2Error  
  
  
*class* Command(BaseCommand):  
 *'''Command to wait for DB'''  
  
 def* handle(*self*, \*args, \*\*options):  
 *'''Entry for command'''  
 self*.stdout.write('Waiting for DB...')  
 db\_up = *False  
 while* db\_up *is False*:  
 *try*:  
 *self*.check(databases=['default'])  
 db\_up = *True  
 except* (Psycopg2OpError, OperationalError):  
 *self*.stdout.write('Database unvailable, waiting one sec...')  
 time.sleep(1)

*self*.stdout.write(*self*.style.SUCCESS('Data is here!'))

Co tu zaszło? Wołamy metodę Handle. Zmieniamy dB na False i gdy ono tak jest to sprawdzamy czy jest baza danych załadowana. Jeżeli nie, używamy except. I czekamy jedną sekundę. Jeżeli jest tru to wlatuje, że baza jest i tyle.

Sprawdzamy czy wszystko działa:

Dzięki działa:

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

Sprawdzamy tero czy działa z naszą aplikacją

**docker-compose run --rm app sh -c "python wait\_for\_db"**

No i linting:

docker-compose run --rm app sh -c "python wait\_for\_db && flake8"

42. Migracje.

Django ma migracje, ORM.

Naprawimy sobie CI/CD zmieniając docker compose.yaml

command: >  
 sh -c "python manage.py runserver 0.0.0.0:8000"

Na to:

command: >  
 sh -c "python manage.py wait\_for\_db &&  
 python manage.py migrate &&  
 python manage.py runserver 0.0.0.0:8000"

Teraz docker-compose down i docker-compose up

No i teraz checks.yaml gdzie:

- name: Test  
 run: docker-compose run --rm app sh –c „python manage.py wait\_for\_db && python manage.py test”

**Sekcja 9. CREATE USER MODEL.**

44-45. Stworzymy model użytkownika z pomocą Django Admin. Mamy system autentyfikacji już w standardzie. System domyślny jest całkiem okej, ale nie do końca. Najlepiej jest zawsze robić swój model.

Stworzymy Abstract Base user i Permission Mixin.

Błędy?

Migracje przed stworzeniem USERa.

TYPOSy w Configu.

Indentacja w modelu menadżera.

Będziemy mieli email. Name, is\_active(boolean), is\_staff(Boolean)

**46. Dodajemy testy do custer model usera.**

Tworzymy plik w testach : test\_models.py a w nim:

*'''  
Test for models.  
'''  
  
from* django.test *import* TestCase  
*from* django.contrib.auth *import* get\_user\_model  
  
*class* ModelTests(TestCase):  
 *'''Test models.'''  
  
 def* test\_create\_user\_with\_email\_success(*self*):  
 *''' Test creating user with an email is successful.'''* email = "test@example.com"  
 password = 'testpass123'  
 user =get\_user\_model().objects.create\_user(  
 email = email,  
 password=password,  
 )  
  
 *self*.assertEqual(user.email, email)  
 *self*.assertTrue(user.check\_password(password))

Na koniec:

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

Wyrzuci błąd bo nie mamy zdefiniowanego użytkownika. Co też właśnie zaczynamy czynić.

Tworzymy modele użytkownika:

*from* django.db *import* models  
*from* django.contrib.auth.models *import* AbstractUser, BaseUserManager, PermissionsMixin  
  
*class* User(AbstractBaseUser, PermissionsMixin):  
 *"""User in the app."""* email = models.EmailField(max\_length=255, unique=*True*)  
 name = models.CharField(max\_length=255)  
 is\_active = models.BooleanField(default=*True*)  
 is\_staff = models.BooleanField(default=*False*)  
  
 USERNAME\_FIELD = 'email'

Wszystko jest Ci znane. Nie ma co tłumaczyć chyba. Przystąpmy do tworzenia USER MENAGO:

Tworzymy menagera, a w nim metodę do tworzenia użytkownika. Przy okazji do superusera.

*class* UserManager(BaseUserManager):  
 *"""Manager for our users."""  
  
 def* create\_user(*self*, email, password=*None*, \*\*extra\_fields):  
 *"""Create, save and return a new user."""*  
 user = *self*.model(email=*self*.normalize\_email(email), \*\*extra\_fields)  
 user.set\_password(password)  
 user.save(using=*self*.\_db)  
  
 *return* user  
  
 *def* create\_superuser(*self*, email, password):  
 *"""Create and return a new superuser."""* user = *self*.create\_user(email, password)  
 user.is\_staff = *True* user.is\_superuser = *True* user.save(using=*self*.\_db)  
  
 *return* user

user.set\_password(password)  
user.save(using=*self*.\_db)

Dobra praktyka, kiedy tworzymy nowego usera. Zapisujemy od razu do DB. Idziemy do settings.py gdzie:

AUTH\_USER\_MODEL = 'core.User'

Zapisujemy nasz model który właśnie stworzyliśmy

Robimy migrację.

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

Ale to już nie wejdzie:

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

Musimy wyczyścić DB:

**Docker volume ls** 🡪 **Docker compose down** i usuwamy ostatni lokalny wolumen z naszą bazą danych i :

**docker volume rm rest-app-api\_dev-db-data** i teraz znowu migracja ta co powyży i przejdzie. I na koniec test.

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

48. Normalize Email adres. Zadbamy o to , żeby emaile się zgadzały. Test models dodajemy nowe funkcje

*def* test\_new\_user\_email\_normalized(*self*):  
 *'''Test email is normalized for new users.'''* sample\_emails = [  
 ['test1@EXAMPLE.com', 'test1@example.com'],  
 ['Test2@Example.com', 'Test2@example.com'],  
 ['TEST3@EXAMPLE.com', 'TEST3@example.com'],  
 ['test4@example.COM', 'test4@example.com'],  
 ]  
 *for* email, expected *in* sample\_emails:  
 user = get\_user\_model().objects.create\_user(email, 'sample123')  
 *self*.assertEqual(user.email, expected)

49. Require email address

Zrobimy tak, żeby User miał wymagany email address.

Test\_models.py

*def* test\_new\_user\_without\_email\_raises\_error(*self*):  
 *'''Test creating a user without an email raises a ValueError.'''  
 with self*.assertRaises(ValueError):  
 get\_user\_model().objects.create\_user('', 'test123')

A teraz wracamy do models.py

*def* create\_user(*self*, email, password=*None*, \*\*extra\_fields):  
 *"""Create, save and return a new user."""  
 if not* email:  
 *raise* ValueError("Users need to have email address")

**51. ADD SUPERUSER**

*def* create\_superuser(*self*, email, password):  
 *'''Create a superuser'''* user = *self*.create\_user(email, password)  
 user.is\_staff = *True* user.is\_superuser = *True* user.save(using=*self*.\_db)  
  
 *return* user

*def* test\_create\_superuser(*self*):  
 *'''Test for creating a superuser'''* user = get\_user\_model().objects.create\_superuser(  
 'test@example.com',  
 'testtetsttest1',  
 )

Testy:

Docker compose up. Działa. To teraz create superusera za pomocą dockera:

Działa. Głupia literówka -,-

**SECTION 10.**

SETUP DJANGO ADMIN.

Co mam więcej mówić? Wrzucamy je do pliku admin.py, site register etc. Znasz to. Będziemy też w ramach tego robić konfigurację DJANGO Admin.

54. Zaczniemy od napisania testów dla DJANGO ADMIN(Ciag dalszy TDD).

App/core/tests == > Test\_admin.py.

*"""  
Test for DJANGO ADMIN modifications  
"""  
  
from* django.test *import* TestCase  
*from* django.contrib.auth *import* get\_user\_model  
*from* django.urls *import* reverse  
*from* django.test *import* Client  
  
*class* AdminSiteTests(TestCase):  
 *""" Test for Django admin."""*

Kod który tu dodamy ruszy pierwszy i będzie sprawdzał setup. Ważne setUp a nie set\_up

*def* setUp(*self*):  
 *"""Create user and client."""  
 self*.client = Client()  
 *self*.admin\_user = get\_user\_model().objects.create\_superuser(  
 email= 'admin@example.com',  
 password = 'sample123',  
 )  
 *self*.client.force\_login(*self*.admin\_user)

Używamy FORCE żeby wymusić logowanie na tym setapie. Przy okazji sprawdzimy nam listę użytkowników oraz to czy może utworzyć też zwykłego użytkownika:

*self*.client.force\_login(*self*.admin\_user)  
*self*.user = get\_user\_model().objects.create\_user(  
 email='user@example.com',  
 password='sample123',  
 name='Test User'

Teraz test to listy użytkowników:

*def* test\_users\_list(*self*):  
 *"""Test that user are listed on a page."""* url = reverse('admin:core\_user\_changelist')  
 res = *self*.client.get(url)  
  
 *self*.assertContains(res, *self*.user.name)  
 *self*.assertContains(res, *self*.user.email)

To ma umożliwić DJANGO ADMIN zmienić listę użytkowników. Składania pochodzi prosto z dokumentacji. url = reverse. Następnie wysyłamy żądanie, a że jesteśmy uwierzytelnieni jako force login, to przechodzi to lekko.

Teraz wrzucamy test:

docker-compose run --rm app sh -c " python manage.py test" i ma nam odrzucić. Odrzuca. Jedziemy więc dalej.

56. Tworzymy DJANGO ADMIN user list.

We wnętrzu core app rejestrujemy modele. Admin.py

*""" Django admin customization"""  
  
from* django.contrib *import* admin  
*from* django.contrib.auth.admin *import* UserAdmin *as* BaseUserAdmin  
  
*from* core *import* models  
  
  
*class* UserAdmin(BaseUserAdmin):  
 *""" Defining the admin pages for users"""* ordering = ['id']  
 list\_display = ['email', 'name']  
  
  
admin.site.register(models.User, UserAdmin)

Teraz możemy wejść do bazy za pomocą docker compose up i zobaczyć, że nam się pozmieniało

Kiedy chcemy se klyknąć i zmodyfikować usera wyjedzie nam błąd prawda?

Teraz to naprawimy.

**56.Support for modyfing USERS.**

Zaczniemy od napisania stosownego testu do tego. Idziemy do test\_admin.py

*def* test\_edit\_user\_page(*self*):  
 *""" Test the edit ueser page works."""*

Zrobimy znowu reverse, ale nieco inaczej.

*def* test\_edit\_user\_page(*self*):  
 *""" Test the edit ueser page works."""* url = reverse('admin:core\_user\_change', args=[*self*.user.id])  
 res = *self*.client.get(url)  
  
 *self*.assertEqual(res.status\_code, 200)

Teraz już wiemy do czego podchodzić, aby to ładnie zaszło. Idziemy do admin.py i dokonujemy tutaj paru ciekawych zmian. Dodamy se od razu translate

Tworzymy fieldsety

Najpierw permission:

*class* UserAdmin(BaseUserAdmin):  
 *""" Defining the admin pages for users"""* ordering = ['id']  
 list\_display = ['email', 'name']  
 fieldsets = (  
 (*None*, {'fields': ('email', 'password')}),  
 (  
 translate('Permissions'),  
 {  
 'fields': (  
 'is\_active',  
 'is\_staff',  
 'is\_superuser',  
 )  
 }  
 ),

(translate('Important dates'), {'fields': ('last\_login', )}),

)

Jest to oczywiście połączone z Permission Mixin,

57.Creating users.

Tak jak poprzednio wracamy do test\_admin.py

Nie potrzebujemy dodawać argumentów, bo przecież tworzymy nowego usera. Więc jest to dość zbliżone do tego co poprzednio:

*def* test\_create\_user\_page(*self*):  
 *""" Test the edit ueser page works."""* url = reverse('admin:core\_user\_add')  
 res = *self*.client.get(url)  
  
 *self*.assertEqual(res.status\_code, 200)

I teraz co zaskakujące. Idziemy do admin.py i pod readonly\_fields dodajemy.

*class* UserAdmin(BaseUserAdmin):  
 *""" Defining the admin pages for users"""* ordering = ['id']  
 list\_display = ['email', 'name']  
 fieldsets = (  
 (*None*, {'fields': ('email', 'password')}),  
 (  
 translate('Permissions'),  
 {  
 'fields': (  
 'is\_active',  
 'is\_staff',  
 'is\_superuser',  
 )  
 }  
 ),  
 (translate('Important dates'), {'fields': ('last\_login', )}),  
 )  
 readonly\_fields = ['last\_login']  
 add\_fieldsets = (  
 (*None*, {  
 'classes': ('wide',),  
 'fields': (  
 'email',  
 'password1',  
 'password2',  
 'name',  
 'is\_active',  
 'is\_staff',  
 'is\_superuser',  
 ),  
 }),  
 )

no module named pytz -,-

**11. API DOCUMENTATION**

Dokumentacja API jest krytycznie ważna!

Co powinna zawierać dokumentacja?

--Dostępne endpointy(ścieżki) – w naszym wypadku /api/recies

--Wspierane metody http 🡪 GET, POST, PUT, PATCH, DELETE

--Formaty INPUTÓW(PAYLOADS) 🡪 parametry, Post JSON format

--FORMATY RESPONSÓW 🡪 RESPONSE JSON FORMAT.

--Proces autentyfikacji użytkownika

**My se zrobimy ją zautomatyzowaną, z pomocą AUTO DOCS with DJango Rest Framework** ponieważ pisanie jej jest dramatycznie czasochłonne

59.

Skorzystamy sobie z frameworka DRF-spectacular gdy jest dość prosty i samofunkcyjny, w dodatku troszkę browsable.

Standard OPEN API SCHEMA, Swagger tool. Tego użyjhemy

Najpierw instalujemy DRF-Spectacular. Idziemy do pliku requirements.txt i dodajemy:

drf-spectacular>=0.15.1,<0.16

a następnie rebuild naszego środowiska : docker-compose build, następnie do settings.py gdzie dodajemy do installed APPS

'rest\_framework',  
'drf\_spectacular',

A następnie na samym dole:

REST\_FRAMEWORK = {  
 'DEFAULT\_SCHEMA\_CLASS': 'drf\_spectacular.openapi.AutoSchema',  
}

Teraz konfigurujemy URLs.

Plik URLS

*from* drf\_spectacular.views *import* SpectacularAPIView, SpectacularSwaggerView  
*from* django.contrib *import* admin  
*from* django.urls *import* path  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('api/schema/',SpectacularAPIView.as\_view(), name='api\_schema'),  
 path('api/docs/', SpectacularSwaggerView.as\_view(url\_name='api\_schema'), name='api\_docs',)  
]

Tylko tyle nam potrzeba na to, aby mieć

**12. BUILD USER API**

TWORZYMY USER API

Jakie endpointy wariacie?

User/create/ 🡪 Post 🡺 register a new user

User/token 🡪 Pot --> create new token

User/me/

Put/patch – update profile

Get – view profile

Najpierw tworzymy nową aplikację:

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

rejestrujemy ją w settings.py i usuwamy test,migracje,model etc. Zostaje tylko apps init i views.

Tworzymy też folder tests a w nim plik \_\_init\_\_.py oraz test\_user\_api.py

*"""  
Test for the user API.  
"""  
  
from* django.test *import* TestCase  
*from* django.contrib.auth *import* get\_user\_model  
*from* django.urls *import* reverse  
  
*from* rest\_framework.test *import* APIClient  
*from* rest\_framework *import* status  
  
CREATE\_USER\_URL = reverse('user:create')

Do stworzenie USER URL użyjemy funkcji REVERSE, co myśmy se ją zaimportowaly. Da nam to dostęp do widoku URLA. Zaraz to skonfigurujemy. Appka User będzie jako stwórca a CREATE to Endopint . To zwróci całą ścieżkę URL wewnątrz naszego projektu. Teraz trzeba stworzyć funkcję, która nam pomoże to wykorzystać. Będziemy robili zaraz funkcję, która będzie tworzyła użytkowników, więc aby nie wpisywać tego na okrągło nie?

*def* create\_user(\*\*params):

Takie coś da nam użytkowość taką 🡪 Wszystkie parametry jakie chcemy możemy tutaj zapisać.

*def* create\_user(\*\*params):  
 *""" Create and return a new user. """  
 return* get\_user\_model().objects.create\_user(\*\*params)

teraz klasa testy.

Najpierw Publiki 🡪

*class* PublicUserApiTests(TestCase):  
 *"""  
 Test the public features os the user API  
 """  
  
 def* setUp(*self*):  
 *self*.client = APIClient()  
  
 *def* test\_create\_user\_success(*self*):  
 *"""  
 Test creating a user is successful.  
 """* payload = {  
 'email': 'test@example.com',  
 'password': 'password1',  
 'name': 'Just test',  
 }

Jako Payload mamy po prostu słownik. Teraz cza to wysłać do aplikacji. Użyjemy do tego funkcji res.

res = *self*.client.post(CREATE\_USER\_URL, payload)

Payload mamy napisany przed chwilą, no i odwołujemy się do create user URL. Teraz tylko self.assertEqual

O tak:

*def* setUp(*self*):  
 *self*.client = APIClient()  
  
*def* test\_create\_user\_success(*self*):  
 *"""  
 Test creating a user is successful.  
 """* payload = {  
 'email': 'test@example.com',  
 'password': 'password1',  
 'name': 'Just test',  
 }  
  
 res = *self*.client.post(CREATE\_USER\_URL, payload)  
  
 *self*.assertEqual(res.status\_code, status.HTTP\_201\_CREATED)  
 user = get\_user\_model().objects.get(email=payload['email'])

To samo powtarzamy dla password i dla name.

*self*.assertTrue(user.check\_password(payload['password']))  
*self*.assertNotIn('password', res.data)

teraz na tym samym poziomie tworzymy funkcję dla usera z błędem.

Def test\_user\_with\_email\_exists\_error() . Korzystamy z tego co już stworzyliśmy.

*def* test\_user\_with\_email\_exists\_error(*self*):  
 *"""Test error returned if user with email exists."""* payload = {  
 'email': 'test@example.com',  
 'password': 'password1',  
 'name': 'Just test',  
 }  
  
 create\_user(\*\*payload)  
 res = *self*.client.post(CREATE\_USER\_URL, payload)  
  
 *self*.assertEqual(res.status\_code, status.HTTP\_400\_BAD\_REQUEST)

kolejny test jeżeli hasło jest za krótkie.

*def* test\_password\_too\_short\_error(*self*):  
 *"""Test an error is returned if password is too short(less than 5 chars)"""* payload = {  
 'email': 'test@example.com',  
 'password': 'password1',  
 'name': 'Just test',  
 }  
 res = *self*.client.post(CREATE\_USER\_URL, payload)  
 *self*.assertEqual(res.status\_code, status.HTTP\_400\_BAD\_REQUEST)  
 user\_exists = get\_user\_model().objects.filter(email=payload['email']).exists()  
 *self*.assertFalse(user\_exists)

Używamy tego co wcześniej my mieli znowu przy czym teraz user exist ma wywoływać fail

Teraz robimy teściki. Oblewamy je, czyli idziemy dalej

67. Jedziemy dalej, dodajemy CREATE USER API.

Tworzymy nowy plik serializers.py i za jego pomocą tworzymy to co wcześniej żeśmy napisały

Napierw importy:

*""" Serializers for the user API View."""  
  
from* django.contrib.auth *import* get\_user\_model  
  
*from* rest\_framework *import* serializers

Potem tworzymy klasę, a w niej funkcję.

*class* UserSerialziers(serializers.ModelSerializer):  
 *""" Serializer for the user object."""  
  
 class* Meta:  
 model = get\_user\_model()  
 fields = ['email', 'password', 'name' ]  
 extra\_kwargs = {'password': {'write\_only': *True*, 'min\_length': 5}}

Tak. Meta odpowiada temu co wpisaliśmy w naszych testach.

*def* create(*self*, validated\_data):  
 *"""Create and return a user with encrypted password."""  
 return* get\_user\_model().objects.create\_user(\*\*validated\_data)

To sposób konwertowania danych do i z PYTHONOWYCH obiektów. Biorą JSONa tworzony przez API i walidują go do postaci czytelnej, bezpiecznej i uwierzytelnionej. Następnie konwertują je jako obiekt pytona, którego możemy użyć jako model w naszej DB.

Jest kilka podstawowych klas serializatorki, których możemy sobie użyć. My użyjemy dostarczony przez RESTa ten co powyżej.

Definicja klasy Meta?

Musimy w niej użyć modelu, więc wołamy get\_user\_model.

Fields, używamy w minimalu, tylko te 3, bo te są wymagane. Może być if ofkorzi więcej. IS ACTIVE, IS STAFF etc. Ale lepiej je definiować przez Admina

Extrakwargs to extra metadane, jako słownik wrzucamy tutaj, że hasło jest wymagane, musi być napisane a jego minimalna długość to 5

Skoro mamy serializera, to teraz będziemy tworzyć views, które z niego skorzystają.

*"""  
Views for the user API  
"""  
  
from* rest\_framework *import* generics  
  
*from* user.serializers *import* UserSerializer  
  
*class* CreateUserView(generics.CreateAPIView):  
 *"""  
 Create a new user in the system  
 """* serializer\_class = UserSerializer

Teraz urls.py(też tworzymy )

*"""  
URL mapping for user API  
"""  
from* django.urls *import* path  
  
*from* user *import* views  
  
app\_name = "user"  
  
urlpatterns = [  
 path('create/', views.CreateUserView.as\_view(), name='create'),  
]

I teraz app/urls.py

path('api/user/', include('user.urls')),  
  
]

69. PISZEMY TESTY for TOKEN API.

Wiadomo, tokeny użytkownik zabezpieczon. Piszemy najpierw dla nich test. TEST\_USER\_API z powrotem.

Na górze tworzymy najpierw stałą zmienną

Teraz tworzymy takie coś. Najpierw tworzymy nowego usera, potem generujemy PAYLOAD z powrotem, gdzie bierzemy email i hasło, następnie generujemy RES i sprawdzamy, żeby assert IN był do TOKENA i assert equal do status code 200

*def* test\_create\_token\_for\_user(*self*):  
 *""" Test generates token for valid credentials"""* user\_details = {  
 'name': 'Test Name',  
 'email': 'test@example.com',  
 'password': 'testpass123',  
 }  
 create\_user(\*\*user\_details)  
  
 payload = {  
 'email': user\_details['email'],  
 'password': user\_details['password'],  
 }  
 res = *self*.client.post(TOKEN\_URL,payload)  
  
 *self*.assertIn('token', res.data)  
 *self*.assertEqual(res.status\_code, status.HTTP\_200\_OK)

Teraz tworzymy kolejną funkcję dla bad credentials

*def* test\_create\_token\_bad\_credentials(*self*):  
 *"""Test returns error if credentials are invalid"""* create\_user(email='test@example.com', password='goodpass')  
  
 payload = {'email':'test@example.com', 'password': 'badpass'}  
 res = *self*.client.post(TOKEN\_URL, payload)  
  
 *self*.assertNotIn('token', res.data)  
 *self*.assertEqual(res.status\_code, status.HTTP\_400\_BAD\_REQUEST)

a kolejną troszkie podobne puste hasło

*def* test\_create\_token\_blank\_password(*self*):  
 *""" Test returns error if password is blank"""* payload = {'email': 'test@example.com', 'password':''}  
 res = *self*.client.post(TOKEN\_URL, payload)  
  
 *self*.assertNotIn('token', res.data)  
 *self*.assertEqual(res.status\_code,status.HTTP\_400\_BAD\_REQUEST )

w obydwu wypadkach jest to dość proste. Podajemy payload z błędem i ma wywołać nam owy błąd. Teraz czas na dodanie TOKENA.

70.

Implementujemy TOKEN API

Zaczynamy od wejścia do settings.py gdzie dodajemy w installed apps nową rzecz.

'rest\_framework.authtoken',

Teraz do serializers, bo przecież to jest nam potrzebne. Zaczynamy od importu.

*from* django.contrib.auth *import* get\_user\_model, authenticate

*from* django.utils.translation *import* gettext\_lazy *as* translation

*class* AuthTokenSerializer(serializers.Serializer):  
 *"""Serializer for the user auth token."""* email = serializer.EmailField()  
 password = serializer.CharField(  
 style={'input\_type': 'password'},  
 trim\_whitespace=*False*,  
 )

Wygląda to znajomo nie?

DRF odcina na końcu whitespace, a akurat w przypadku hasła to my tego nie chcemy.

Teraz dodamy metodę walidacji, aby to zaszło.

*def* validate(*self*, attrs):  
 *"""Validate and auth user."""* email = attrs.get('email')  
 password = attrs.get('password')  
 user = authenticate(  
 request=*self*.context.get('request'),  
 username=email,  
 password=password,  
 )  
 *if not* user:  
 msg = translation('Unable to auth with provided credentials.')  
 *raise* serializers.ValidationError(msg, code='authorization')  
  
 attrs['user'] = user  
 *return* attrs

Mamy zdefiniowaną funkcję.

Mamy dane zapisane jako attrs.get . Następnie mamy user które opiera się na wbudowanym systemie w Django. Potem instrukcja warunkowa.

Teraz musimy zrobić views, aby se on zadziałał. Skorzystamy z czegoś co jest zaadaptowane. Zaczynamy od dooimportowania:

*from* rest\_framework.authtoken.views *import* ObtainAuthToken  
*from* rest\_framework.settings *import* api\_settings  
  
*from* user.serializers *import* UserSerializer, AuthTokenSerializer

Mięcho:

*class* CreateTokenView(ObtainAuthToken):  
 *"""Create a new token for user."""* serializer\_class = AuthTokenSerializer  
 render\_classes = api\_settings.DEFAULT\_RENDERER\_CLASSES

A teraz URLE.

urlpatterns = [  
 path('create/', views.CreateUserView.as\_view(), name='create'),  
 path('token/', views.CreateTokenView.as\_view(),name='token'),  
]

i już

71. Piszemy testy dla manage user API.

Dodajemy kolejną rzecz do zarządzania USER API. Najpierw jednak musimy w test\_user\_api dodać kolejną stałą.

ME\_URL = reverse('user:me')

*def* test\_retrieve\_user\_unauthorized(*self*):  
 *"""Test authentication is required for users"""*

Tutaj musimy sprawdzić czy autentyfikacja jest required.

Prostym testem:

*def* test\_retrieve\_user\_unauthorized(*self*):  
 *"""Test authentication is required for users"""* res = *self*.client.get(ME\_URL)  
  
 *self*.assertEqual(res.status\_code, status.HTTP\_401\_UNAUTHORIZED)

Klasa dla czegoś, co nie potrzebuje mieć autentyfikacji.

*class* PrivateUserApiTests(TestCase):  
 *"""Test API requests that require auth."""  
  
 def* setUp(*self*):  
 *self*.user = create\_user(  
 name='Test Name',  
 email='test@example.com',  
 password='testpass123',  
 )  
 *self*.client = APIClient()  
 *self*.client.force\_authenticate(user=*self*.user)  
  
 *def* test\_retrieve\_profile\_success(*self*):  
 *"""Test retrieving profile for logged user"""* res = *self*.client.get(ME\_URL)  
 *self*.assertEqual(res.status\_code, status.HTTP\_200\_OK)  
 *self*.assertEqual(res.data, {  
 'name': *self*.user.name,  
 'email': *self*.user.email,  
 })

Tutaj zgodnie z nazwą sprawdzamy, czy nasza metoda ME jest not allowed.

*def* test\_post\_me\_not\_allowed(*self*):  
 *"""Test POST is not allowed for the me endpoint"""* res = *self*.client.post(ME\_URL, {})  
  
 *self*.assertEqual(res.status\_code, status.HTTP\_405\_METHOD\_NOT\_ALLOWED)

Sprawdzamy tutaj, upewniamy się, że metoda http POST jest wyłączona dla każdych endpointów. http POST powinno być używane jedynie wtedy, kiedy tworzysz jakiś obiekt w systemie. Dopóki nie jest stworzone, to API powinno być wyłączone dla metody POST w tym endpoincie. Nie możesz więc

Na zakończenie. Test update user profile

*def* test\_update\_user\_profile(*self*):  
 *"""Test updating user profile for the auth user. """* payload = { 'name': 'Update', 'password': 'newpass123'}  
  
 res = *self*.client.patch(ME\_URL, payload)  
 *self*.user.refresh\_from\_db()  
 *self*.assertEqual(*self*.user.name, payload['name'])  
 *self*.assertTrue((*self*.user.check\_password(payload['password'])))  
 *self*.assertEqual(res.status\_code, status.HTTP\_200\_OK)

Nie wydaje się to skomplikowane, najpierw robimy payload, gdzie dajemy nowe dane, potem wywołujemy RES. Następnie odświeżamy bazę danych i sprawdzamy czy te dwa podane misie się zgadzają, na koniec wywalamy res.status\_code 200.

72. Implementing MANAGE USER API.

Aby dodać nową metodę potrzebujemy serializera. Po raz kolejny. Dodamy metodę update user.

Poniżej metody create w klasie USER SERIALIZER tworzymy nową metodę. UPDATE

*def* update(*self*, instance, validated\_data):  
 *"""update and return user"""* password = validated\_data.pop('password', *None*)  
 user = super().update(instance, validated\_data)  
  
 *if* password:  
 user.set\_password(password)  
 user.save()  
  
 *return* user

podobnie dla kreatyny, uaktualniamy obiekt. Używamy specjalnej funkcji update, która ma dwa parametry pop i validated data(z automatu). Wrzucamy w niej dwie rzeczy i instrukcję warunkową.

W password używamy pop, aby usunąc to coś ze słownika, zostawiając. Hasło zostawiamy se opcjonalne, bo może to nie to ma być zmienione. Dajemy NONE i jest bezpieczniej. Następnie jest user w funkcji super, która ma za zadanie apdejtować

Następnie instrukcja warunkowa 🡺 Jeżeli Password będzie true, to ustaw nowe hasło i zwróć usera.

Teraz do VIEWS, gdzie zmienimy kilka rzeczy

*from* rest\_framework *import* generics, authentication, permissions

*class* ManageUserView(generics.RetrieveUpdateAPIView):  
 *"""Manage the autheticated user."""* serializer\_class = UserSerializer  
 authentication\_classes = [authentication.TokenAuthentication]  
 permission\_classes = [permissions.IsAuthenticated]  
   
 *def* get\_object(*self*):  
 *"""Retrive and return the auth user."""  
 return self*.request.user.

RetrieveUpdateAPIView jest jednym z narzędzi DRF do updejtu widoku i na koniec URLsy.

User urls.py

path('me/', views.ManageUserView.as\_view(), name='me'),  
]

Dodajemy i już

Nowego użytkownika możemy dodać przez swagern wchodzimy w user create-> potem try it zmieniamy na multipart/form data i już możemy. Mamy nowego użytkownika. Teraz dodajemy do niego token. Klikamy w TOKENA 🡺 Try it out 🡺 Dane które wpisaliśmy 🡪 Execute 🡪 Zwróci nam token który sobie kopiujemy 🡪 klikamy w authorize i wybieramy token Auth i zapisujemy go koniecznie w takim formacie:

Token **4645a405772416f08d0e48ebd47c490ddfb3e249**

Teraz idziemy do ME klikamy se execute i widzimy czy działa czy nie

**SEKCJA 13. RECIPE API(albo coś podobnego)**

75-76.API VIEWS a VIEWSETS

DRF używa klas zamiast funkcji będąc trochę ponad DJANGO. Jakie są tego zalety?

- Logika wielokrotnego użytku. Możemy jej potem uzyć ponownie wewnątrz aplikacji

- Nadpisywanie zachowywania funkcji.

DRF wspiera również dekoratory funkcji

**APIVIEW** i **VIEWSETS** to dwie podstawowe klasy Django Rest Framework, które pozwalają nam implementować funkcjonalnośc

APIViews 🡪 Używaliśmy ich do tej pory, aby postawić bebeszki naszej aplikacji.

* Te są sfokusowane na metodach http
* Klasy metod dla http. Wszystkie są wspierane: GET, POST,PUT,PATCH,DELETE
* Używanie API VIEWS zapewnia większą elastyczność logiki i URL
* Użyteczne dla NON CRUD APIs

VewSets 🡪 Tego będziemy od teraz używać.

🡺 Skoncentrowane wokół akcji (retrieve, list, update, partial update i destroy)

* Zmapowane do DJANGO models
* Świetne dla CRUD (tak jak ta aplikacja)

**77. WRITE TEST FOR RECIPE MODEL**

Wewnątrz core/tests dodajemy testy bo tam mamy model (test models.py)

*from* decimal *import* decimal  
  
*from* core *import* models

Doimportowuję to i poniżej piszemy teścik. Proste po prostu.

*def* test\_create\_recipe(*self*):  
 *""" Test creating a recipe is successful."""* user = get\_user\_model().objects.create\_user(  
 'test@example.com',  
 'sample123',  
 )  
 recipe = models.Recipe.objects.create(  
 user=user,  
 title='Sample ',  
 time\_minute=12,  
 cost=Decimal('5,50'),  
 description='sample description'  
 )  
  
 *self*.assertEqual(str(recipe), recipe.title)

Dodajemy sobie model TRAVELI(ex Recipe)

*class* Travel(models.Model):  
 *"""Travel destination object."""* user = models.ForeignKey(settings.AUTH\_USER\_MODEL,on\_delete=models.CASCADE,)  
 title = models.CharField(max\_length=255)  
 description = models.TextField(blank=*True*)  
 days\_spent = models.IntegerField()  
 hours\_spent = models.IntegerField()  
 cost = models.DecimalField(max\_digits=5, decimal\_places=2)  
 link = models.CharField(max\_length=255, blank=*True*)  
  
 *def* \_\_str\_\_(*self*):  
 *return self*.title

i rejestrujemy nasz model w admin.py

admin.site.register(models.Travel)

i robimy migrację. I elo.

79. Tworzymy aplikację Book.

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

tworzymy folder tests, tam \_\_init\_\_.py usuwamy wszystko co nie jest nam potrzebne i rejestrujemy aplikację.

80. Testy:

*""" Test for recipe APIs."""  
  
  
from* decimal *import* Decimal  
  
*from* django.contrib.auth *import* get\_user\_model  
*from* django.test *import* TestCase  
*from* django.urls *import* reverse  
  
*from* rest\_framework *import* status  
*from* rest\_framework.test *import* APIClient  
  
*from* core.models *import* Book  
  
*from* recipe.serializers *import* BookSerializer

Już wrzucamy. Częśc z nich posłuży nam później. Teraz tworzymy sobie testy.

Ta pierwsza pomoże nam stworzyć nasze wpisy(dynamiczne)

*def* create\_book(user, \*\*params):  
 *""" Create and return a sample entry, record"""* defaults = {  
   
 }

Wrzucamy

*def* create\_book(user, \*\*params):  
 *""" Create and return a sample entry, record"""* defaults = {  
 'title':'Sample entry',  
 'category' : 'Drama',  
 'number\_of\_pages' : 121,  
 'language' : 'Hindi',  
 'cost' : Decimal('5.50'),  
 'description' : 'sample description',  
 'link' : 'http://example.com/entry.pdf',  
 }  
 defaults.update(params)  
  
 book = Book.objects.create(user=user, \*\*defaults)  
 *return* book

Jest to jako słownik, słownik jest niezmienny, więc tworzymy nowy słownik, który to apdejtujemy. Nadpisujemy, wołamy i zwracamy. Teraz tworzymy klasę testów dla nieautoryzowanych wejść do aplikacji.

*class* PublicBookAPITests(TestCase):  
 *"""Test unauthenticated API requests"""  
  
 def* setUp(*self*):  
 *self*.client = APIClient()  
  
 *def* test\_auth\_required(*self*):  
 *"""Test auth is required to call API"""* res = *self*.client.get(BOOKS\_URL)  
  
 *self*.assertEqual(res.status\_code, status.HTTP\_401\_UNAUTHORIZED)

standarodowo setUp, wołamy, odwołując się do tego co tworzymy powyżej:

BOOKS\_URL = reverse('books:books-list')

Na koniec assertEqual, rest status code i http unaothirezed. Nie robimy tego publicznie. Jedynie dajemy możliwość zalogowania się.

Podobnie robimy dla uwierzytelnionego użytkownika.

*class* PrivateBookAPITests(TestCase):  
 *"""Test authenticated API requests"""  
 def* setUp(*self*):  
 *self*.client = APIClient()  
 *self*.user = get\_user\_model().objects.create\_user(  
 'test@example.com',  
 'sample123',  
 )  
 *self*.client.force\_authenticate(*self*.user)

najpierw wołamy api, potem tego naszego użytkownika, tworzymy go, i siłowo przepychamy auth.

Teraz możemy dodać testy, zbliżone do tych z usera

Najpierw ten, gdize odwołyujemy się do pierwszej testowej funkcji, create book

*def* test\_retrieve\_books(*self*):  
 *""" Test retrieving a list of recipes."""* create\_book(user=*self*.user)  
 create\_book(user=*self*.user)  
  
 res = *self*.client.get(BOOKS\_URL)  
  
 books = Book.objects.all().order\_by('id')  
 serializer = BookSerializer(books, many=*True*)

Tak tworzymy dwie książki, mamy dwie w bazie, wrzucamy response. Spodziewamy się, że wrzuci dwie, sprawdzając ich porządek po **minus ID**. To nam pokaże książkę z najwyższym ID bo przecież minus ID. Teraz tworzymy serializera w niej serializer.

Spodziewamy się, że więcej niż jedna pasuje, dlatego many jest bezpieczniejsze bo pokaże tego więcej, na koniec assertEqual i dwuseta. Odwołamy się w drugim do serialziera.

*self*.assertEqual(res.status\_code, status.HTTP\_200\_OK)  
*self*.assertEqual(res.data, serializer.data)

Teraz powinniśy widzieć obydwa booksy jako nasze jako APIs

Napiszemy teraz test taki.

Bo nie wiemy czy np. API nie wrzuci nam wszystkich, które nie chcemy. Może chcemy tylko te dla auth usera, który to jest zalogowany? Wygląda bardzo podobnie do poprzedniego. Sprawdzamy czy jest user i czy już istnieje

*def* test\_recipe\_list\_limited\_to\_user(*self*):  
 *""" Test list of recipes is limited to auth user."""* other\_user = get\_user\_model().objects.create\_user(  
 'another@example.com',  
 'sample1232',)  
 create\_book(user=other\_user)  
 create\_book(user=*self*.user)  
 res = *self*.client.get(BOOKS\_URL)  
 books = Book.objects.filter(user=*self*.user)  
 serializer = BookSerializer(books, many=*True*)  
 *self*.assertEqual(res.status\_code, status.HTTP\_200\_OK)  
 *self*.assertEqual(res.data, serializer.data)

W tym teście wyrzucamy pojedyncze. Więc nie martwimy się innymi takimi ciekawymi, bo wylecą jako pojedynczy respons. W tym teście mamy dwie książki metodki, jedna jest przyklejona do innego użytkownika, a druga nie. I wołamy je, ale chcemy zobaczyć tylko tą co jest dla auth usera. Przechodzimy przez Serializer, sprawdzamy status i porównujemy go do serializer.data ModuleNotFoundError: No module named 'book.serializers'. Idziemy dalej

00

81. Implement Book listing API.

Mamy już testy, dodajmy sobie więc kod, żeby to mogło działać.

W appce books tworzymy serializers.py a w nim dokonujemy o to takiego kodu.

*"""  
Serializers for book APIs.  
"""  
from* rest\_framework *import* serializers  
*from* core.models *import* Book  
  
*class* BookSerializer(serializers.ModelSerializer):  
 *"""Serializer for books"""  
 class* Meta:  
 model = Book  
 fields = ['id', 'title', 'author', 'number\_of\_pages', 'category', 'language', 'link']  
 read\_only\_fields = ['id']

Tworzymy klasę dla serialziera. Meta to wiadomo, z tego chcemy korzystać, a fields, to te co mają być wyświetlone. Mamy kolejny błąd, ale nieco inny. Mamy no reverse match, bo nie dodaliśmy URLA czy tam VIEWSA. Teraz czas dodać views we views.py

Robimy viewsety:

*"""  
Views for books APIs.  
"""  
from* rest\_framework *import* viewsets  
*from* rest\_framework.authentication *import* TokenAuthentication  
*from* rest\_framework.permissions *import* IsAuthenticated

Pojedziemy na viewsetach, ściągamy też auth, bo będziemy jej używać i permissions, aby zabronić nie auth użytkownikowi wiadomo co.

Tworzymy viewseta. Jest kilka ich modeli, bierzemy najbardziej podstawowy 🡪 Ten jest stworzony do działania z Modelem, a to akurat tutaj mamy

*class* BookViewSet(viewsets.ModelViewSet):  
 *"""View for manage book APIs"""* serializer\_class = serializers.BookSerializer  
 queryset = Book.objects.all()  
 authentication\_classes = [TokenAuthentication]  
 permission\_classes = [IsAuthenticated]

Odwołujemy się do stworzonego wcześniej bookserilizera, jako queryset bierzemy wszystkie obiekty z BOOK, a potem formułka z auth

Aby wszystko było grejt wołamy teraz ten queryset:

*def* get\_queryset(*self*):  
 *"""Retrieve books fot auth user."""  
 return self*.queryset(user=*self*.request.user).order\_by('-id')

Wyświetlamy po negatywnym ID, Get queryset to wywołanie obiektu.

Teraz trzeba zrobić parę zmian . Tworzymy w appce book plik urls.py

*"""  
URL mapping for this app.  
"""  
  
from* django.urls *import* path, include  
*from* rest\_framework.routers *import* DefaultRouter  
*from* book *import* views  
  
router = DefaultRouter()  
router.register('books', views.BookViewSet)  
  
app\_name = 'book'  
  
urlpatterns = [  
 path('', include(router.urls)),  
]

Tworzymy swój domyślny router. Mamy tutaj path i include, czyli domyślne funkcje dla CRUD

No i coś nie działa.(once again)

*def* test\_retrieve\_books(*self*):  
 *""" Test retrieving a list of books."""* create\_book(user=*self*.user)  
 create\_book(user=*self*.user)  
  
 res = *self*.client.get(BOOKS\_URL)  
  
 books = Book.objects.all().order\_by('-id')  
 serializer = BookSerializer(books, many=*True*)  
 *self*.assertEqual(res.status\_code, status.HTTP\_200\_OK)  
 *self*.assertEqual(res.data, serializer.data)

Ta funkcja powodowała problemy, cholera wie dlaczego xD. Jedziemy dali

82. Piszemy testy for Book detail API.

To jest dość w sumie proste. Dodajemy nowy serializer:

*from* book.serializers *import* BookSerializer, BookDetailSerializer

zaraz za tym tworzymy funkcję o taką:

*def* detail\_url(book\_id):  
 *"""Create and return a book detail URL."""  
 return* reverse('book:book-detail', args=[book\_id])

dlaczego tworzymy funkcję detail\_url skoro mamy jako zmienną BOOKS\_URL ? Bo musimy wrzucić BOOK ID do URLa. Każdy detal będzie inny, jest unikalne ID dla każdej z książek, którą chcemy testować. Ten detal jest zawarty w funkcji.

Pod spodem krótki test:

*def* test\_get\_book\_detail(*self*):  
 *"""Test get book detail."""* book = create\_book(user=*self*.user)  
  
 url = detail\_url(book.id)  
 res = *self*.client.get(url)  
 serializer = BookDetailSerializer(book)  
 *self*.assertEqual(res.data, serializer.data)

Tworzymy kolejny wpis, książkę, następnie tworzymy detal URL, potem za pomocą RES wołamy naszą metodę, tworzymy serializer a następnie za pomocą Asser equal sprawdzamy czy są takie same

**83. Teraz trzeba napisać Serializer, aby to się ładnie wyświetliło**

Definiujemy nową klasę.

*class* BookDetailSerializer(BookSerializer):  
 *"""Serializer for book detail view"""*

A dlaczego korzystamy z BookSerializer? Bo będzie w prostej linii rozszerzeniem tego powyżej BookSerializera. Przeniesiemy całą funkcjonalność z głównego serializera do tego i dodamy se kilka rzeczy za pomocą.

*class* BookDetailSerializer(BookSerializer):  
 *"""Serializer for book detail view"""  
  
 class* Meta(BookSerializer.Meta):  
 fields = BookSerializer.Meta.fields + ['description']

Nie jest to skomplikowane. Teraz musimy zmodyfikować viewset w views.py w taki sposób, aby przywoływał DETAIL endpoint zamiast BOOK, który zdefiniowaliśmy w naszym Serializerze. Chcemy użyć detalu, gdy wywołamy jego endpoint. Osiągniemy to w sposób ODWRACAJĄC metodę GET\_SERIALIZER\_CLASS

Modyfikujemy też BookViewset:

*class* BookViewSet(viewsets.ModelViewSet):  
 *"""View for manage book APIs"""* serializer\_class = serializers.BookDetailSerializer  
 queryset = Book.objects.all()  
 authentication\_classes = [TokenAuthentication]  
 permission\_classes = [IsAuthenticated]  
  
 *def* get\_queryset(*self*):  
 *"""Retrieve books fot auth user."""  
 return self*.queryset.filter(user=*self*.request.user).order\_by('-id')  
  
 *def* get\_serializer\_class(*self*):  
 *""" Return the serializer class for request."""  
  
 if self*.action == 'list':  
 *return* serializers.BookSerializer  
  
 *return self*.serializer\_class

Jeżeli akcja to list, używamy BOOK SERIALIZER, inaczej używamy DETAIL. Czyli de facto odwróciliśmy to sobie

**84. Piszemy test dla tworzenia BOOK API**

Całym klu jest zrobienie tego przez API, a nie z pomocą funkcji pomocniczej. Chcemy przekazać ładunek użyteczny do API z kontentem. Musimy mieć zapewnienie, że jest to stworzone poprawnie w bazie danych.

*def* test\_create\_book(*self*):  
 *"""Test creating a new entry"""* payload = {  
 'title': 'Sample entry',  
 'author': 'Johny Bravo',  
 'category': 'Drama',  
 'number\_of\_pages': 121,  
 'language': 'Hindi',  
 'cost': Decimal('5.50'),  
 'description': 'sample description',  
 'link': 'http://example.com/entry.pdf',  
 }  
  
 res = *self*.client.post(BOOKS\_URL, payload)

Teraz logicznie oczekujemy http 200

*self*.assertEqual(res.status\_code, status.HTTP\_201\_CREATED)  
book = Book.objects.get(id=res.data['id'])

Tak więc bierzemy obiekt po ID, a następnie musimy teraz zrobić pętlę

*def* test\_create\_book(*self*):  
 *"""Test creating a new entry"""* payload = {  
 'title': 'Sample entry',  
 'author': 'Johny Bravo',  
 'cost': Decimal('5.50'),  
 }  
  
 res = *self*.client.post(BOOKS\_URL, payload)  
  
 *self*.assertEqual(res.status\_code, status.HTTP\_201\_CREATED)  
 book = Book.objects.get(id=res.data['id'])  
 *for* k, v *in* payload.items():  
 *self*.assertEqual(getattr(book, k), v)  
 *self*.assertEqual(book.user, *self*.user)

Zaczynamy definiując payload, potem wzywamy clienta post, http POST request, potem sprawdzamy czy mamy 201, co oznacza, że sukces. Potem uzyskujemy specyficzne ID tego utworzonego misia i teraz powinno być już utworzone.

Teraz tylko pętla przez słownik k – KEY, v – VALUE payload.items(). Teraz używamy funkcji assertEqual oraz GETATTR dostarczonej przez Pythona czy klucz i wartość są takie same. Na koniec sprawdzamy czy user który to dodał jest równy temu, którego żeśmy uwierzytelninili

85. Implementujemy create BOOK API

Musimy tylko dodać jedną metodę do istniejącego Viewu.

*def* perform\_create(*self*, serializer):  
 *"""Create a new book entry"""* serializer.save(user=*self*.request.user)

Tak się zapisuje metodę tworzenia w DRF. Jeden z nich musi być zwalidowanym serializatorem a drugi to self.

Nic skomplikowaneog