[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)