cHEX APP NOTATKI.

Czego będziemy używać?

Oczywiście: Python

Django(URL MAPPING, Object Relational Mapper, Admin)

DJANGO REST FRAMEWORK

POSTGRE SQL

DOCKER

SWAGGER UI

11.

Dlaczego używać Dockera?

-Dostajemy spójne środowisko rozwoju i produkcji naszej aplikacji.

-Używamy tego samego kodu do produkcji jak i do rozwoju naszej aplikacji. To eliminuje liczbę błędów.

-Łatwiejsza współpraca z innymi programistami. Eliminuje błędy jakie powstają po instalowaniu na innym kompie.

- Przechwytuje wszystkie zależności jako kod. Nie trzeba konfigurować i ściągać, bo to się samo robi

- Łatwiejsze zarządzanie, czyszczenie projektu.

Jak używamy dockera?

Najpierw konfigurujemy plik dockera.

Tworzymy konfigurację docker Compose (wszystkie komendy)

Docker Hub(limit 100 wejść na 6), po zalogowaniu 200 na 6. Jest to strona wspólna. Jak się uwierzytelnimy, to będziemy mogli wykorzystać więcej.

12.

Tworzymy klon do gita. Najpierw repo, potem przechodzimy do tego katalogu, potem git clone i adres do tego repo

Teraz Docker. Settings🡺 Security🡺New access token . Tworzymy token z nazwą apki w który ona będzie. Pamiętamy aby se skopiować token, bo go zobaczymy tylko raz

a28a52f3-5637-42f4-a177-5cdc2f0aa096

Dodajemy go do secretów w GITHUBIE. Także nazwę użytkownika

13-14. Co nam daje DOCKER W DJANGO?

Spójny rozwój i produkcja aplikacji (i wszystkie z 11)

A jakie są wady?

1.Brak dostępu do interpretera w VSC(ale my i tak PYCHARM)

2. Trudniejsze korzystanie ze zintegrowanych funkcyj.

Konfigurejemy Dockera w Django

W git bash piszemy **code .**

Otwiera się VSC, tworzymy tam plik requirements.txt

Django=>4.0.0

djangorestframework=>3.13.1

15. Tworzymy Dockerfile

Plik nazywa się po prostu docker file:

FROM python:3.9-alpine3.13

W pythonie, używamy versji Alpine3.13. Lekka wersja Linuxa,której będziemy używać

LABEL maintainer="firstAPP"

ENV PYTHONBUFFERED 1

Mantainer to nazwa naszej aplikaci. BUFFERED to informacja dla Docker.

To tam wyżej oznacza, że Nie chcesz buforować danych wyjściowych

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

COPY ./app /app

WORKDIR /app

EXPOSE 8000

To zaś, to ustawienia dla wirtualnej maszyny, co se ją będziemy odpalać zaraz. Teraz trochę więcej zaawansowanych:

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

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

USER djang-user

Objaśnienie krok po kroku. To jedna komenda, mająca uruchomić naszą aplikację. 1 linijka tworzy nowe venv(niektórzy go nie tworzą, ale my to zrobiliśmy, żeby zmiejszyc ryzyko konfliktów w plikach), 2 linijka to uaktualnienie pipa, trzecia to zainstalowanie REQUIREMENTSów. Czwarta usuwa TMP, żeby nie był za ciężki. Linijki od 5 do 8 to tworzenie Usera. To najlepsza praktyka nie tworzyć usera. Nie uruchamiajmy aplikacji na ROOTUSERZE. Nie tworzymy hasła, bo nam tu nie poczebne. Nie tworzymy HOME DIRECTION też. Na koniec nazywamy go **DJANGO-USER**(tak to nazwa użytkownika)

DALEJ znajduje się ENV PATH 🡪 Jeżeli uruchamiamy jąkąś rzecz w naszym kodzie, nie chcemy pisać całej długiej linii do kodu, to ją skróci.

Na koniec po prostu user Django-user. Za każdym razem kiedy uruchomimy docker image uruchomi się jako django-user

Tworzymy .dockerignore 🡪 Nie cza tłumaczyć

#Git

.git

.gitignore

#Docker

.docker

#Python

app/\_\_pychache\_\_/

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

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

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

.env/

.venv/

venv/

Takich plików nie chcemy wrzucać na dockera.

GIT to jasne, Docker też, ponieważ czasem zrobi się ukryty. Nie jest potrzebny

No i Python.

Tworzy Kaszę, dlatego chcemy usunąc owe kaszki z naszego pliku. Dodajemy też .env .venv i venv

To nam pomogło:

DOCKER\_BUILDKIT=0 docker build .

16 Docker Compose conf.

Tworzymy plik docker-compose.yml a w nim:

version: '3.9'

services:

  app:

    bulid:

      context: .

    ports:

        - "8000:8000"

      volumes:

        - ./app:/app

      command: >

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

O co w tym wszystkim chodzi?

Services to skonfigurowanie appki.

Context: . oznacza użycie tej lokalizacji w której jesteśmy,

Port 8000: to lokalnie

Volumes to lokalizacja appki

Command to wiadomo, uruchomienie

17.-18.

Linting and testing

Linting to narzędzie do sprawdzania kodu, samo się sprawdza

Flake8 🡪 tej użyjemy **$ docker-compose run --rm sh -c "flake8"**

Wyrzuci nam jako zwrot listę błędów

Tworzymy nowy plik. requirements.dev.txt . Tak, taki plik requirements podczas develepowania naszej appki.

W pliku tym dajemy na razie jedną linijkę:

flake8>=4.0.1

Za to przerabiamy nieco plik docker-compose.yml

 bulid:

      context: .

      args:

        - DEV=True

Oraz Dockerfile dodając taką linijkę pod COPY (pierwszym)

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

Teraz cza przerobić plik go dalsze. O tak, dając instrukcję warunkową:

RUN python -m venv /py && \

    /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 && \

Używamy tego do restartu:

docker-compose up -d

Wewnątrz app tworzymy plik .flake8, a tam wrzucamy pliki, których nie chcemy se spradzać

[flake8]

exclude =

  migrations,

  \_\_pycache\_\_,

  manage.py,

  settings.py

Wreszcie tworzymy DJANGO project, jeno przez docker-compose

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

A jak bo będziem uruchamiać zazwyczaj?

**Docker compose-up**

Zatrzymujemy klasycznie ctrl+c

SEKCJA 2. Configure GitHub Actions

22. Co to jest?

Narzędzie do automatyzacji. Posłuży nam do wyłapywania błędów i do napisania testów.

Pomijamy to:

**Przechodzimy do 33.** Configure DB

Ma konfigurację w momencie kiedy strona jest, jest lepsze wsparcie dla Dockera i pozwala używać zmiennych do konfigurowania.

Architektura?

Docker Compose ma dwa serwisy, DB i APP. Można otworzyć jeden, lub dwa na raz. Komunikują się ze sobą.

Odpowiednią konfigurację za chwilę przeprowadzimy. APP services może użyć DB jako hosta.

Wolumeny:

- Trwałe dane

- Katalog map znajduje się w komputerze lokalnym, w kontenerze -> Dane zawsze będą bo są trwałe. Ale to za chwilę.

Dodamy teraz bazę.

Otwieramy docker-compose.yml na dole, na tej samej indentacji co app dodajemy:

  db:

    image: postgres:13-alpine

poniżej, bez wcięć:

volumes:

  dev-db-data:

Nic nie dodajemy poniżej, docker se sam doda

db:

    image: postgres:13-alpine

    volumes:

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

environment:

      - POSTGRES\_DB=devb

      - POSTGRES\_USER=devuser

      - POSTGRES\_PASSWORD=changeme

Dodajemy to, te 3 enviromnent vars, to są po to, żeby ustawić inicjacyjne uruchomienie naszej DB. To są tylko dane do startu. Raczej nie używać tego w produkcji

Pod Command, w tej samej indentacji dodajemy to:

  environment:

      - DB\_HOST=db

      - DB\_NAME=devdb

      - DB\_USER=devuser

      - DB\_PASS=admin

depends\_on:

      - db

DB\_HOST jest równe DB, bo tak zatytułowaliśmy to poniżej. A zmienne pasują do tych poniżej tu

Na koniec docker-compose up i hulo

35. Konfiguracja DB z Django.

Najpierw musimy powiedzieć Django jak się połączyć. Potem zainstalować adapter, narzędzia które pozwolą się połączyć. Następnie trzeba zainstalować Python Requirements.

Zanim Django się połączy musi wiedzieć jaki ma engine wyszukiwarki, Hostname(IP i domena), numer portu, nazwę DB, Username, hasło. Wiadomo, idziemy do settings.py. Zastosujemy tutaj ENVIRON – Environment Vars.(za chwilę)

Paczka Psycopg2 -> najbardziej popularny adapter do Pytona, oficjalnie wspierany przez Django.

Jest kilka opcji instalacyjnych:

**psycopg-2 binary.** Jest okej przy rozwoju, gorsza w produkcyjnej wersji, nie polecana jak chcemy to gdzieś wrzucić

**psycopg2** – jest instalowana z kodu źródłowego, albo z pipa. Jest kompilowana specjalnie dla kompa, którego używamy. Można ją łatwiej zainstalować za pomocą Dockera.

Zawiera kilka paczek ekstra. C compiler, python3-dev, libpq-dev. Nie wiem po co są, ale będą. Alpine 13 ma swoje

Przed instalacją warto uporządkować zależności.

Dockerfile, nad PIP UPGRADE PIP dodajem

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

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

bulid-base posgresql-dev musl-dev &&\

a pod fi &&\ dodajemy:

rm -rf /tmp && \

apk del .tmp-bulid-deps && \

tam powyżej, jako postgresql- client instalujemy clienta do obsługi paczki, poniżej konfigurujemy wersję virtualną.

Tam poniżej, zwróćmy uwagę, że usuwamy tą paczkę. Jak ruszymy w produkcji, to jej nie chcemy. Teraz w requirements, dodajemy nową linijkę

psycopg2==2.9.3

no i na koniec docker-compose build

'C:\ProgramFiles\Docker\Docker\DockerCli.exe' –SwitchDaemon 🡪 Gdyby docker się nie włączył.

37. Konfiguracja Django Settings(w końcu)

Otwieramy settings.py i dokonujemy sobie małych zmian. (Ps. Da się mieć kilka baz danych w DJANGO)

Zmieniamy SQLa na POSTGRESa

TO:

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': BASE\_DIR / 'db.sqlite3',

    }

}

Na to(najpierw importujemy OS)

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'),

    }

}

DB HOST,USER,NAME,PASS pasują do tych, które ustaliśmy w pliku docker-compose.yml

Naprawimy bazę danych, bo zawsze ma ona jakieś problemy. **38-40.**

Docker Compose upewnia się, że DB rusza pierwsza, ale to nie oznacza, że uruchomi się do czasu, w którym ruszy aplikacja DJANGO, może być tak, że DJANGO się wywali, bo wyśle zapytanie do DB, a ona dalej będzie wstawać. Może się zdarzyć zarówno gdy jest lokalne, lub już na serwerze. Musiemy dodać do django **wait for db**. Najpierw musimy zrobić nową apkę w DJANGO

**$ docker-compose run --rm app sh -c "python manage.py startapp core"**

Usuwamy z niej tests i views, bo nam nie będą poczebne, tworzymy też folder tests, a w nim plik \_\_init\_\_ oraz dodajemy core do settings.py

**40. Testy do WAIT\_FOR\_DB.**

Jest ona solidnie opisana w dokumentacji DJANGO, ale przejdziemy przez to razem.

Zaczynamy od stworzenia w core folderu menagement, a tam pliku \_\_init\_\_ . W folderze menagement tworzymy folder COMMANDS, a w nim kolejne init i wait\_for\_db.py

Ów plik waitfor:

from django.core.management.base import BaseCommand

class Command(BaseCommand)

    '''Django command to wait for db '''

    def handle(self, \*args, \*\*options):

        pass

Wracamy do folderu tests. Wewnątz tworzymy plik test\_commands.py, a wnim takie importy:

''' Test custom Django menago commands'''

from unittest.mock import patch

from psycopg2 import OperationalError as Psycopg2Error

from django.core.menagement import call\_command

from django.db.utils import OperationalError

from django.test import SimpleTestCase

Tych potrzebujemy. Mock jest do tego, aby baza nam zwracała czy jest lub nie

Symulujemy zachowanie naszej DB, więc nie są nam potrzebne migracje i takie tam

''' Test custom Django menago commands'''

from unittest.mock import patch

from psycopg2 import OperationalError as Psycopg2OpError

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 CommandTests(SimpleTestCase):

    """Test commands"""

    def test\_wait\_for\_db\_ready(self, patched\_check):

        patched\_check.return\_value = True

        call\_command('wait\_for\_db')

        patched\_check.assert\_called\_once\_with(databases=['default'])

Teraz test dla db, gdy nie jest gotowa

@patch('time.sleep')

    def test\_wait\_for\_db\_delay(self, patched\_sleep, patched\_check):

        """Waiting test for db when OperationalError."""

        patched\_check.side\_effect = [Psycopg2OpError] \* 2 + [OperationalError] \* 3 + [True]

        call\_command('wait\_for\_db')

        self.assertEqual(patched\_check.call\_count, 6)

        patched\_check.assert\_called\_with(databases=['default'])

Dodajemy komendę czekaj na DB.(41)

import time

from psycopg2 import OperationalError as Psycopg2OpError

from django.db.utils import OperationalError

from django.core.management.base import BaseCommand

class Command(BaseCommand)

    '''Django command to wait for db '''

    def handle(self, \*args, \*\*options):

        """Entrypoint for commands"""

        self.stdout.write('Waiting for db...')

        db\_up = False

        while db\_up is False:

            try:

                self.check(databases=['default'])

                db\_up = True

            except (Psycopg20pError, OperationalError):

                self,stdout.write('DB unvailable, waiting 2 sec.')

                time.sleep(2)

        self.stdout.write(self.style.SUCCESS('DB is available'))

42. MIGRACJE DB.

Otwieramy plik docker-compose

    command: >

      sh -c "python manage.py wait\_for\_db &&

             python manage.py migrate &&

             python manage.py runserver 0.0.0.0:8000"

i robimy tak. To za każdym razem uruchomi naszą funkcję czekającą na DB oraz wrzuci migracje, żeby uaktualnić nasze DB.

'C:\Program Files\Docker\Docker\DockerCli.exe' –SwitchDaemon 🡪 chyba cza to za każdym razem uruchamiać

hexapp-db-1 | 2022-05-21 09:14:46.936 UTC [33] FATAL: database "devdb" does not exist ☹

**USER MODEL**

Mam nadzieję, że to nie wpłynie na poprawność funkcjonowania tego programu. A jak nie to duuuuu. Konfigurujemy USER\_MODEL.

Najpierw napiszemy test.

W core/tests tworzymy plik test\_models.py

Zaczynamy od takiego importu :

from django.test import TestCase

from django.contrib.auth import get\_user\_model

Get\_user\_model jest najbardziej uniwersalny:

class ModelTests(TestCase):

    '''Testing models'''

    def test\_create\_user\_with\_email\_successful(self):

        email = 'test@test.com'

        password = 'testtest'

        user = get\_user\_model().objects.create\_user(

            email=email,

            password=password,

        )

        self.assertEqual(user.email, email)

        self.assertTrue(user.check\_passowrd(password))

Nie ma tu nic skomplikowanego. Najpierw tworzymy funkcję, a niej domyślne dane, a potem zapomocą assert.Equal sprawdzamy czy jest takie same. Do tego samego służy assertTrue do hasła

**Dodajemy USERMODEL (48)**

Zaczynamy od models.py:

from django.db import models

from django.contrib.auth.models import AbstractBaseUser, BaseUserManager,PermissionsMixin

class User(AbstractBaseUser, PermmisionsMixin):

    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)

    basic = models.BooleanField(default=False)

    premium = models.BooleanField(default=False)

    enterprise = models.BooleanField(default=False)

    USERNAME\_FIELD = 'email'

Nie ma tu nic skomplikowanego. Wszystko jest raczej jasne, przechodzimy do stworzenia ModelMenagera.

Umieszczamy go nad poprzednio stworzoną klasą:

class UserManager(BaseUserManager):

    def create\_user(self, email, password=None, \*\*extra\_field):

        """Tworzymy,zapisiujemy i zwracamy użytkownika"""

        user = self.model(email=email, \*\*extra\_field)

        user.set\_password(password)

        user.save(using=self.\_db)

        return user

Krytycznie ważne jest dobre nazewnictwo, funkcja powinna się nazywać create\_user. Ważna jest też wcięciografika

Keyword arguments są całkiem ciekawym rozwiązaniem na uniwersalne dodywanie kolejnych method klasy.

Nie umieszczamy tu hasła. Dlaczego? To zapewni mu dodatkowe szyfrowanie.

user.save(using=self.\_db) 🡪 a to da mu możliwość pracy z wieloma bazami danych.

Teraz wystarczy podłączyć user\_menaga do klasy USER.

Ano tak:

    objects = UserManager()

A w settings.py dodajemy autentyfikację.

AUTH\_USER\_MODEL = 'core.User'

Teraz możemy zrobić sobie migracje taką linijką kodu:

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

następnie:

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

Teraz musimy zatrzymać dockera i wyczyścić dane z naszej DB, więc **docker-compose down**, następnie wpisujemy **docker volume ls**.

Wybieramy stamtąd nasze api i wpisujemy(w tym przypadku): **docker volume rm hexapp\_dev-db-data**

TO NIEZWYKLE WAZNE JAK CI SIĘ WYKRZACZY. To naprawiło problem z piątku.

Odpalamy test, żeby sprawdzić czy działa:

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

49. NORMALIZACJA USER\_ADDRESU.

Do naszego test\_models.py dodajemy nowinkę.

Upewnimy się, że nasz program wspiera dodawanie unikalnych maili z różnymi kapitalizacjami.

    def test\_new\_user\_email\_normalized(self):

        """Testing normalized users 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, 'sample221')

            self.assertEqual(user.email, expected)

tworzymy tą funkcyję. Dajemy kilka przykładowych maili, a następnie jedziemy pętelkę.

To da nam aszybkę. Żeby naprawić błąd. I już.

Błąd od konta bez maila, również banalnie prosty:

    def test\_new\_user\_whitout\_email(self):

        with self.assertRaises(ValueError):

            get\_user\_model().objects.create\_user('', 'bobofrut')

Ale nie zadziała, jeżeli nie zmienimy naszej clasy użytkownik

class UserManager(BaseUserManager):

    def create\_user(self, email, password=None, \*\*extra\_field):

        """Tworzymy,zapisiujemy i zwracamy użytkownika"""

        if not email:

            raise ValueError('Email is mandatory!')

        user = self.model(email=self.normalize\_email(email), \*\*extra\_field)

        user.set\_password(password)

        user.save(using=self.\_db)

        return user

Dodamy se superusera

W models.py, w klasie UserMenager dodajemy nową funkcję:

 def create\_superuser(self,email,password):

        user = self.create\_user(email, password)

        user.is\_staff = True

        user.is\_superuser = True

        user.save(user=self.\_db)

        return user

A do tego test w test\_models

   def test\_create\_superuser(self):

        user = get\_user\_model().objects.create\_superuser(

            'test@test.com',

            'test223',

        )

        self.assertTrue(user.is\_superuser)

        self.assertTrue(user.is\_staff)

Nie ma błędów. Można iść dalej. Przetestujemy nasze modele, jeżeli wszystko jest okej, to widzimy się na następnej stronie tworząc DJANGO\_ADMIN.

Docker-compose up. Jeżeli wszystko poszło w porządku to możemy wejść na lokalny serwer w przeglądarce i wejść do DJANGO. Tak samo powinien zadziałać ADMIN. Tworzymy sobie superusera z poziomu CMD:

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

**SETUP DJANGO ADMIN**

Zaczniemy od napisania testów, bo jak, że to tak. Ta metoda pracy jest całkiem ciekawa, to trzeba przyznać.

Tworzymy plik w core/tests/test\_admin.py, a wnim takie importy:

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):

    def setUp():

Achtung, UnitTest używa innej konwencji, CammelCase a nie Snake\_case to zapisywania nazw funkcji.

Tworzymy sobie dwóch użytkowników, regulara i superusera.

 def setUp(self):

        self.client = Client()

        self.admin\_user = get\_user\_model().objects.create\_superuser(

            email='admin@test.com',

            password='test223',

        )

        self.client.force\_login(self.admin\_user)

        self.user = get\_user\_model().objects.create\_user(

            email='user@test.com',

            password='test232',

            name='TestUsa'

        )

A teraz sprawdzamy czy są na liście stworzonych userków:

    def test\_users\_list(self):

        '''Testing users are on page'''

        url = reverse('admin:core\_user\_changelist')

        res = self.clientget(url)

        self.assertContains(res, self.user.name)

        self.assertContains(res, self.user.email)

Bierzemy reverse, aby dostać listę użytkowników, następnie za pomocą RES wywołujemy clienta, a on zostanie wyrzucony jako użytkownik.

Na dole mamy dwie assercję, pierwsza sprawdza usera, druga email

class AdminSiteTests(TestCase):

    def setUp(self):

        self.client = Client()

        self.admin\_user = get\_user\_model().objects.create\_superuser(

            email='admin@test.com',

            password='test223',

        )

        self.client.force\_login(self.admin\_user)

        self.user = get\_user\_model().objects.create\_user(

            email='user@test.com',

            password='test232',

            name='Test Usa'

        )

    def test\_users\_list(self):

        '''Testing users are on page'''

        url = reverse('admin:core\_user\_changelist')

        res = self.clientget(url)

        self.assertContains(res, self.user.name)

        self.assertContains(res, self.user.email)

Gdy zainstalujemy userów, testy nie będą failowały. Ale jest tak jak powinno być. Dodamy listę użytkowników i przestanie.

56. Tworzymy listę użytkowników Django

Wewnątrz Core wchodzimy do admin.py

Do rejestracji modeli użyjemy nieco zmodyfikowanego modelu podstawowego z Django. Najpierw importy

from django.contrib import admin

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

from core import models

class UserAdmin(BaseUserAdmin):

    order =['id']

    list\_display= ['email', 'name']

admin.site.register(models.User)

To zarejestruje podstawowy model, ale teraz czas go nieco zmodyfikować. Wołamy naszą custom klasę:

admin.site.register(models.User, UserAdmin)

I pojawiła się nowa lista użytkowników, ale jeszcze nie działa. Idziemy do następnego filmu, gdzie to naprawimy.

57. Modyfikowanie użytkowników.

Zaczniemy od dodania testu.

    def test\_edit\_user\_page(self):

        url = reverse('admin:core\_user\_change', args=[self.user.id])

        res = self.client.get(url)

        self.assertEqual(res.status\_code,200)

całkiem podobny test co do tego co był nieco wyżej, chcemy, żeby wyrzucił nam status\_code = 200. Ale na razie oblejemy, bo jeszcze tego nie mamy. Czas to zmienić nie?

Musimy zrobić dalszą kustomizację, aby móc robić to co chcemy sobie zrobić. Ściągamy też translatora, aby aplikacja ta była bardziej uniwersalna.

from core import models

class UserAdmin(BaseUserAdmin):

    ordering =['id']

    list\_display= ['email', 'name']

    fieldsets = (

        (None, {'fields': ('email', 'password')}),

        (\_('Personal Info'), {'fields': ('name',)}),

        (

            \_('Permissions'),

            {

                'fields': (

                    'is\_active',

                    'is\_staff',

                    'is\_superuser',

                )

            }

        ),

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

    )

    readonly\_fields = ['last\_login']

admin.site.register(models.User, UserAdmin)

Co tutaj jest zrobione? Owa kastomizacja fields. Stworzyliśmy nowy model powiązany z naszym modelem, który wcześniej żeśmy napisały. Dodajemy też informację, kiedy się ów człowiek zalogował ostatni raz, oraz readonly\_fields = żeby dało się sprawdzić tylko to. Teraz nasze add usser działa.

58. Dodamy Creating Users.

Tradycyjnie zaczniemy od testu, bardzo podobny do poprzedniego, ale w sumie nie potrzeba nam user ID, bo go jeszcze nie mamy:

    def test\_create\_user\_page(self):

        url = reverse('admin:core\_user\_add')

        res = self.client.get(url)

        self.assertEqual(res.status\_code,200)

teraz do admin.py

Ciekawy sposób pisania formularza do Django. Na to bym nie wpadł. Tak to wygląda.

  add\_fieldsets = (

        (None, {

            'classes': ('wide',),

            'fields': (

                'email',

                'password1',

                'password2',

                'name',

                'is\_active',

                'is\_superuser',

                'is\_basic',

                'is\_premium',

                'is\_enterpirse',

            ),

        }),

    )

**KOLEJNA SEKCJA: (11 w Kursie)**

59. TWORZENIE DOKUMENTACJI API/

Dlaczego robimy dokumentację?

Ponieważ API są tworzone do użytku przez deweloperów. Muszą oni wiedzieć, jak używać tego coś napisał.

**API jest tak dobre jak jego dokumentacja.** Bez dobrych doców, API jest trudne w użyciu.

Co tam powinno być?

1. Wszystko potrzebne, aby używać API.
2. Dostępne punkty końcowe(endpointy) np. /api/users
3. Wspierane metody(GET,POST,PUT,PATCH,DELETE itp.)
4. Format użytecznych danych( inputs) Takie jak parametry, które można wpisać, query. POST JSON Format(jeżeli jest) 🡪 Jest to potrzebne aby skutecznie używać owych
5. Format odpowiedzi(Danych wyjściowych – outpost). Takie jak znowu JSON
6. Procesy autentykacji użytkownika.

Opcje dla dokumentacji:

Manual 🡪 Np. Markdown, albo .doc w Wordzie. Trzeba manualnie wpisywać zmiany w programie(także to trochę słabe)

Zautomatyzowana 🡪 Uzyjemy metadanych z kodów, komentarzy, żeby automatycznie stworzyć dokumentację.

W tej sekcji właśnie takowe poznamy,

60. Tworzymy takową automatyczną dokumentację wraz z REST Framework.

Użyjemy biblioteki trzeciego rzędu, drf-spectacular. Ta jest ponoć najlepsza, świetna i łatwa w użyciu. Stworzy przy okazji automatycznie dokument, schemat jako JSON lub YML. Będzie dostępna z poziomu przeglądarki. Pozwoli tworzyć testowe requesty, posiada też moduł authentication.

Jak to działa?

1. Najpierw trzeba zrobić plik schematu.
2. Parsowanie schematu w GUI 🡪 Stworzy nam interface dla użytkownika

OpenAPI Schema jest standardem, jest popularne w środowisku, ma wsparcie i wspierane przez największe API. Używa tych samych YAML I JSON co powyższa.

Zaserwujemy Swaggera z API.

**61. Instalujemy DRF-SPECTACULAR.**

Najpierw wrzucamy go do requirements.txt

drf-spectacular==0.22.1

Aby wprowadzić zmiany w naszym containerze w Dockerze, wrzucamy sobie *docker-compose build*

Po zainstalowaniu wchodzimy w app/settings.py i do settings dodajem takowe rzeczy:

    'rest\_framework',

    'drf\_spectacular'

Rest był zainstalowany, a drfu teraz będziemy używać, a teraz skonfigurujemy RESTa, żeby używał schematów z DRF:

REST\_FRAMEWORK = {

    'DEFAULT\_SCHEMA\_CLASS': 'drf\_spectacular.openapi.AutoSchema',

}

Teraz działa

**62. Konfigurujemy URLe.**

Wchodzimy do app/app/urls.py i robimy tylko tyle:

from drf\_spectacular.views import SpectacularAPIView,  SpectacularSwaggerView

from django.urls import path

urlpatterns = [

    path('admin/', admin.site.urls),

### API ###

    path('api/schema/',SpectacularAPIView.as\_view(), name='api-schema'),

    path('api/docs/',SpectacularSwaggerView.as\_view(url\_name='api-schema'),name='api-docs',

    )

]

Testujemy. Teraz gdy wrzucimy w bęben te tutaj, dostaniemy albo plik YAML(ten pierwszy – schema) albo piękny rozkład naszego API – (api/docs).

**SEKCJA GŁÓWNA. BUDUJEMY CLUE PROGRAMU**

Obejrzymy parę filmów i zobaczymy co i jak: (65)

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

Tworzymy sobie tą apkę, w której będziemy się urządzać nie? Usuwamy niepotrzebne pliki:

Migrations, bo te robimy w CORE,

Admin, jak wyży

Models, z tego samego powodu

Test, bo nie używamy testów w tej postaci, a teraz tylko settings i zainstalować ją

77.

Różnice pomiędzy APIview a Viewsets.

Obydwie to podstawowe funkcje do stworzenia widoku. Różnice między nimi?

APIView:

- Skupiają się na metodach http.

- Specjalne metody dla http(GET, POST,PUT,PATCH,DELETE)

- zapewniają elastyczność w stosunku do adresów URL i logiki.

- Użyteczne dla NON CRUD APIs.

- Lepiej unikać ich dla prostych aplikacji CRUD.

- „Logika na zamówienie” dobre dla aplikacji: autentyfikacja, szukanie pracy albo jakieś zewnętrzne API.

VIEWSETS:

- Skupione na akcji, nie na metodach. Dobre na metody jak: pobieranie, lista, aktualizacja, częściowa aktualizacja, zniszczenie.

- zmapowane do modeli DJANGO.

- używa ROUTES do wygenerowania adresu URL

- Świetne do modeli CRUD, bo mnóstwo predefiniowanych rozwiązań.