**Django**

**First Lesson:**

Django-admin startproject mypage – Komenda tworząca projekt w Django.

* Jeśli terminal będzie informował o błędzie możliwe, że nie ma ścieżki do lokalizacji, gdzie jest zainstalowane django

Python manage.py startapp „własna nazwa” – komenda tworząca bloki projektu, które nazywamy „app”ką.

Python manage.py runserver – Komenda tworząca serwer,

**Second Lesson:**

Ps. trzeba importować plik django do pliku pycharm przez przystąpieniem do runserver

Ciekawostka kiedy robimy import z innego pliku .py możemy zrobić coś takiego: „from . import urls.py”. Pozwala to na pobranie z tego samego położenia co nasz plik wskazany plik.

**Proces działania django:**

W scieżce głównego projektu, w urls.py w liście urlpatterns, django wyszukuje

1. W stworzonym bloku APP, w pliku **views.py** tworzymy nasza funkcję, która ma być wezwana w momencie wpisania odpowiedniego url.

Z biblioteki django potrzebujemy: from django.shorcuts import render i from django.http import HttpResponse

def index(request):

return HttpResponse(“Hello World!”)

1. Kolejnym krokiem jest w pliku urls.py stworzyć listę w której mamy poszczególne ścieżki dla danego bloku APP

Z bibliotek potrzebnych tutaj mamy from django.urls import path

urlpatterns = [

path(„nasza nazwa ścieżki np. january”, views.nazwa\_funkcji np. january)

]

1. Trzecim krokiem jest w pliku głównym projektu urls.py dodać do istniejącego urlpatterns listy nową ścieżkę

Z bibliotek django potrzebujemy: from django.contrib import admin I from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('challenges/', include('challenges.urls'))

]

**Dynamiczne wybieranie segmentów I wychwytywanie ich wartości**

Cały trik polega na wskazaniu w pliku z urls.py dla danego bloczku APP identyfikatora, który jest wsadzony w dynamic path segments < >:

path(„**<**month**>**”, views.monthly\_challenge)

Do tego jest stworzona pojedyncza funkcja w views.py, która robi coś co następnie wyświetla dane na stronie.

def monthly\_challenge(request, month):

display = dict[month]

return HttpResponse(display)

Django ma dokładnie taki sam sposób wpływania na typ danych, które są wychwytywane s url jak Flask:

path("<**str:**month>", views.monthly\_challenge)

Istotna rzecz, jeśli ten sam argument o tej samej nazwie jest wykorzystywany w dwóch funkcjach (pomimo tego, że wzywasz dwie różne funkcje), kolejność w urls.py ma znaczenie:

urlpatterns = [  
 path("<int:month>", views.monthly\_challenge\_by\_num),  
 path("<str:month>", views.monthly\_challenge)  
]

Najpierw sprawdza czy argument może zostać integer, jeśli nie to przechodzi do dalszej części funkcji.

**Redirect**

W przypadku Django redirect działa na zasadzie wykorzystania classy HttpResponseRedirect”

try:  
 redirect\_month = list(monthly\_challenges.keys())[month-1]  
 print(redirect\_month)  
except IndexError:  
 return HttpResponseNotFound(f"There is no data for {month} input")  
  
return HttpResponseRedirect("/challenges/" + redirect\_month)

Musimy wskazać lokalizqacaję bloku APP, który nas interesuje w tym przypadku „challenges” i dodać argument, który jest wymagany przez funkcję, w tym wypadku jest to nazwa miesiąca.

Zasada działania jest taka: wysyłamy ansz request do serwera, gdzie wychwytuje, że szukamy challenges, następnie w urls dla challenges sprawdza input argument jaki wprowadziliśmy. Jako, że mamy rozpoznanie argumentu przez dwie funkcje, to zaczyna od tej która jest na początku urlpatterns. Jako, że wprowadziliśmy miesiąc w postaci cyfry, to odsyła nas serwer do funkcji monhtly\_challenge\_by\_num, gdzie nasze dictionary jest zamieniane w liste z imionami i z tej listy wybieramy po indeksie, który jest anszym argumentem. Kiedy już mamy nazwę miesiąca, przy pomocy HttpRedirect odsłyamy informacje serwerowi, że szukamy czegoś o nowym argumencie, który tym razem jest rozpoznany jako string nazwy miesiąca.

\*pamiętać trzeba o znaku /

**Reverse function and named URLs**

Stosuje się w momencie wykorzystania redirect.

Zasada działania jest taka, że z funkcja reverse bierze argument „name” z bloku APP URLs do stworzenia dynamicznego URL dla naszego redirecta oraz wprowadzamy argumenty (ilość jest zależna od składowych dynamicznych segmentów URL)

redirect\_path = reverse("month-challenge", args=[redirect\_month]) #Tworzy ścieżkę /challenges/miesiąc  
return HttpResponseRedirect(redirect\_path)

**Dodawanie I rejestrowanie templatów**

Tak samo jak w przypadku Flaska, tworzymy directory o nazwie templates w lokalizacji naszej APPki i wewnątrz jej tworzymy kolejny directory o nazwie naszej APPki. Robi się to po to aby uniknąć problemów, gdy posiada się kilka APPek, gdzie sią templaety o tej samej nazwie np. index.html.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, multimedia

Opis wygenerowany automatycznie

Aby móc korzystać z templatów należy wykonać zarejestrować APP:

W settings.py projektu można wprowadzić direction DIR do lokalizacji poszczególnych APP temaplatów:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Nie jest to jednak najlepsza metoda dla APPek, ponieważ za każdym razem, gdybyśmy tworzyli nową APP, musielibyśmy dokładać nową lokalizację. Można korzystać z tej metody w przypadku, gdy mamy templaty wykorzystywane przez wiele APPek. Natomiast, w przypadku templates, które są wykorzystywane przez wiele APPek jest ot najlepsze rozwiązanie.

Lepszym rozwiązaniem jest wykorzystanie APP\_DIR i wprowadzenie w INSTALELD\_APPS nazwy naszej APPki. Opcja ta jest bardzo dobra w przypadku wykorzystywania templatów przeznaczonych dla naszej App.

Obraz zawierający tekst, zrzut ekranu, Czcionka, design

Opis wygenerowany automatycznie

**Rendering Templates**

Aby móc renderować templaty, wpierw musimy importować z django.shorcuts import render.

Wzywając render, pierwszy argument to request, kolejny to string z lokalizacją naszego templata a kolejne to key argguments wprowadzone w postaci dictionary:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

DTL – Django Template Language. Dziala na tej samej zasadzie co Jinja z varaible interpolation {{ }} oraz {% %}

Django Filtry oraz tags – elementy, które pozwalają na wpływanie na template np.:

{{ month|title }} – filtr zamienia pierwsze litery variable na duża literę.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Strona z poszczególnymi filtrami i elementami:

<https://docs.djangoproject.com/en/4.2/ref/templates/builtins>

SQLite – Baza danych, której konstrukcja przypomina tabele jak w excelu.

Przykładowa konstrukcja danych, do utworzenia bazy danych w SQLite. Tak samo jak w przypadku templatów, trzeba dodać naszą APPkę do listy z Appkami, aby Django wiedział, gdzie znajdzie bazę danych.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Nie Trzeba dodawać ID, Id jest samo tworzone podczas dodawania danych do bazy danych.

Utworzenie bazy danych jest podzielone na dwa etapy: makemigrations oraz migrate.

Obydwie komendy wykorzystuje się z pola terminalu.

Makemigrations:

W polu terminalu wpisujemy:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Tworzy to w folderze projektu migrations.py plik, który zawiera instrukcję jak ma zostać stworzona/zmodyfikowana baza danych etc.

Migration:

Wprowadza instrukcję z folderu migrations do bazy danych.

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

W ten o to sposób mamy zmodyfikowaną bazę danych.

**WAŻNE KAŻDA ZMIANA W KONSTRUKCJI DODAWANEGO OBIEKTU MUSI PRZEJŚĆ MAKEMIGRATION ORAZ MIGRATE.**

Aby wprowadzić dane do naszej bazy danych, możemy wykorzystać terminal z interpretatorem naszego języka. Aby to zrobić należy wpierw odpalić terminal komendą: python manage.py shell.



Dodawanie danych do bazy danych:

Wpierw musimy importować classę do stworzenia obiektu, który zostanie wprowadzony do bazy danych: from nazwa\_appki.models import nazwa\_classy



Następnie tworzymy obiekt do dodania do danych i zapisujemy obiekt:



Można również wykorzystać methodę create, bez konieczności zapisywania:



Modyfikacja danych

Tak jak w przypadku listy python:

Tworzymy variable Harry\_potter = Book.objects.all()[0]

I wybieramy atrybut do zmiany Harry\_potter.rating = 5 i zapisujemy varibale Harry\_potter.save()

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Usuwanie danych

Działa na podobnej zasadzie co modyfikacja.



Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Wyszukiwanie danych

GET:

Books.objects.get(id=5) – zwróci nam pozycję dla której id jest równe 5. UWAGA get zwraca tylko jedną pozycję. Dodatkowo, ta pozycja jest tylko dla obiektów które mają unikatowy atrybut np. id albo nazwę. W przypadku kilku obiektów o np. raiting 5 dostaniemy ERROR.

FILTER:

Books.objects.filter(rating=5) – Zwróci nam wszystkie obiekty, które posiadają rating 5. Można dać jako kilka warunków do wyszukiwania.

Jeśli chcemy wyszukać np. mniejsze równe to składnia wygląda tak: Books.objects.filter(rating\_\_lt=3)

FILTER CONTAINS

Books.objects.filter(title\_\_contains=”story”) – Sprawdza daną kolumnę bazy danych, czy w komórce znajduję się „story” w tytule.

Strona, która zawiera wszystkie składnie i filtry zapytań:

<https://docs.djangoproject.com/en/4.2/ref/models/querysets/#field-lookups>

OR Conditions

Aby skorzystać z warunku OR najpierw należy zimportować moduł: **from django.db.models import Q**

**AND Condition**

**Używamy znaku przecinka ,**



**Ważne!**

**GET zwraca nam classę modelu, gdzie wystarczy podać atribute name.**

**W przypadku FLTER uzykujemy serię zapytań (Query Set), aby uzyskać wartości dla danej klasy, musimy dostać się do pierwszej pozycji poprzez np. book[0].title**

W naszym modelu możemy nadpisać metodę get\_absolute\_url z wykorzystaniem funkcji return

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Co pozwala nam na szybkie stworzenie URL w anchor tag.

Obraz zawierający tekst, Czcionka, zrzut ekranu

Opis wygenerowany automatycznie

Również ciekawą opcją jest napisanie w classie modelu methody save aby utworzyła automatycznie uzupełniła atrybut naszej classy w tym przypadku „slug”:

Wymaga to zimportowania methody:

from django.utils.text import slugify

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Aggregation emthods:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

**Order\_by**

Można również uporządkować rosnąca lub malejąco ( wstawiając znak – przy parametrze po którym ma być segregowane) używając:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Admin

W celu skorzystania z opcji admina, wpierw musimy stworzyć superusera.

Obraz zawierający tekst, Czcionka, zrzut ekranu

Opis wygenerowany automatycznie

Dzięki temu możemy się zalogowac na stronie dla admina aplikacji.

Aby widzieć poszczeólne bazy danych na naszej stronie admina należy wpierw poinformować Django o tym w folderze Admin:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Configurate Admin Settings

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Możemy konfigurować ustawienia np. w bazie danych jak tworzony jest sług poprzez stworzenie classy BookAdmin, dobry zwyczaj aby tak nazywać klasy, super od ModelAdmin. Dzięki prepopulated\_fields możemy wprowadzić, że atrybut z naszej klasy modelu, jest tworzony na bazie poszczególnych pól np. title. Konstrukcja jest jak w dictionary z tym, że value to tuple. Pamiętać aby w tuple zawsze po 1 wartości był przecinek.

Dzieki stworzeniu klasy Bookadmin i wprowadzeniu prepolutated mogliśmy usunąć metodę z klasy Book z save.

Jeden z atrybutów do naszej klasy BookAdmin list\_filter, pozwala nam wprowadzić w tuple atrybuty z klasy Book, po których możemy filtorwac na stronie admina.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, oprogramowanie, System operacyjny

Opis wygenerowany automatycznie

Relathionship in models

One to Many – where we use the foreignkey. One author has many books

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

Przy zapytaniach np. filter stosujemy dwa podkreślnik Book.objects.filter(author\_\_last\_name=”Rowling”)

Obraz zawierający tekst, Czcionka, zrzut ekranu, linia

Opis wygenerowany automatycznie

Gdy chcemy zrobić zapytanie, gdzie chcemy sprawdzić ile autor ma książek to robi się to tak:

Tworzymy variable z autorem jkr = Author.objects.get(last\_name=”Rowling”)

Wyszukujemy po book\_set - jkr.book\_set.all()

book\_set jest automatycznie generowane przez Django w momencie tworzenia relacji pomiędzy dwoma modelami, w tym wypadku Book i Author.

Jeśli chcemy zmienić ta nazwę to możemy to zrobić poprzez zmianę parametru w klasie Book, gdzie mamy foregin\_key przy pomocy related\_name



One to One – with usage of OneToOneField

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

Jeśli chcemy sprawdzić kto jest przypisany do danego adresu, robimy to w następujący sposób.

Adress.objects.get(id=1).author.first\_name

Tutaj jako, że jest to konstrukcja one to one nazwa po której dostajemy się do przypisanebazy jest nazwą modelu: author

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Test