Django - Wprowadzenie

Zajęcia Prymus 2020

Agnieszka Rudnicka rudnicka@agh.edu.pl

Czym jest Django?

Django to darmowy i open source'owy framework do budowania aplikacji webowych napisany w Pythonie. Innymi słowy to zestaw narzędzi, który przyśpiesza i ułatwia znacząco proces tworzenia stron.

Gdy tworzymy strony internetowe, wiele elementów się powtarza między projektami. Są to przykładowo mechanizmy tworzenia, zarządzania i uwierzytelniania użytkowników, panel zarzadzania treścią czy też mechanizmy wyświetlania i przetwarzania formularzy. Django wychodzi na przeciw tym powtarzającym się wyzwaniom oferując między innymi:

- gotowy system tworzenia, rejestracji i logowania użytkowników
- gotowy system grup i uprawnień do modeli
- mini-framework do tworzenia i przetwarzania formularzy (django.contrib.forms)
- auto-generowany panel administracyjny
- gotowe klasy (Class Based Views) na podstawie których można w kilka linijek kodu tworzyć pełnoprawne widoki CRUD (Create Read Update Delete)
- potężny ORM, czyli narzędzie do operowania na danych w bazach danych bez potrzeby użycia SQL
- wbudowane mechanizmy cachowania, wysyłania maili
- ...i wiele innych

Przygotowanie środowiska

Aby rozpocząć pracę z projektem utwórzmy nowy katalog.

Można to zrobić poleceniem: `mkdir library-project`

Sprawdźmy jeszcze, czy mamy poprawnie zainstalowany język Python w wersji 3 kolejnym poleceniem w terminalu:

```
python -V
```

Naszym oczom powinno się ukazać coś takiego: `Python 3.7.4` (lub inna wersja). W przypadku systemów opartych o jądro Linux trzeba wpisać `python3 -V`, ponieważ samo polecenie `python` może być linkowane do starszego interpretera (2.7).

Tworzenie środowiska wirtualnego

Django jak wiele innych narzędzie jest swego rodzaju dodatkiem/pakietem dodatkowym, który nie jest zainstalowany wraz z podstawowym interpreterem języka Python.

Standardową procedurą przy rozpoczynaniu każdego projektu jest stworzenie "wirtualnego środowiska" (virtual environment - venv) w którym znajdzie się kopia interpretera wraz z wszystkim doinstalowanymi dodatkowymi bibliotekami danego projektu.

Aby stworzyć wirtualne środowisko należy wykonać polecenie:

```
python -m venv moje_srodowisko
# lub krócej
python -m venv venv
```

Powyższe polecenie stworzy w bieżącym katalogu lokalną instalację Pythona w katalogu `moje_srodowisko` albo `venv` zależnie od tego które polecenie postanowimy wykonać. Polecam to krótsze, pierwsze ma jedynie charakter demonstracyjny.

Teraz musimy aktywować środowisko wirtualne. Jeśli używamy IDE takiego jak PyCharm lub VSCode, prawdopodobnie zostanie ono automatycznie aktywowane o czym dowiemy się widząc w nazwę środowiska wirtualnego w nawiasach (na przykładzie systemu Windows):

```
(venv) C:\Users\<user>\PycharmProjects\books>
```

Chodzi oczywiście o nazwę środowiska `(venv)`, która to normlanie się nie pokazuje.

Aby jednak ręcznie aktywować środowisko - np. gdy nie używamy IDE należy wykonać poniższe polecenie:

```
# windows:
venv\Scripts\activate.bat
# linux:
source venv/bin/activate
```

Instalowanie bibliotek/zależności

A teraz właściwa część przygotowania środowiska - czyli instalujemy framework Django:

```
pip install Django
```

Po wykonaniu powyższego polecenia możemy sprawdzić aktualnie zainstalowane pakiety poleceniem:

```
pip list
```

U mnie lista wygląda tak:

```
(venv) $> pip list
Package    Version
---------- -------
asgiref    3.2.7
Django     3.0.5
pip        19.0.3
pytz       2019.3
setuptools 40.8.0
sqlparse   0.3.1
```

Dobrą praktyką jest stworzenie pliku z listą zależności wymaganych do uruchomienia projektu. Najprostszą metodą jest wykonanie polecenia:

```
pip freeze > requiremets.txt
```

Zapisze ono wszystkie biblioteki wraz z dokładnymi ich wersjami do pliku o nazwie `requirements.txt`. Nazwa tego pliku jest pewnego rodzaju konwencją, którą można spotkać w wielu projektach.

Zaglądnijmy więc do pliku `requirements.txt`:

```
asgiref==3.2.7
Django==3.0.5
pytz==2019.3
sqlparse==0.3.1
```

Jak widać, lista jest trochę krótsza niż wynik działania `pip list`. Nie znajdziemy tutaj pakietów `pip` oraz `setuptools`, bo są one niejako "wbudowane" w instalacje Pythona i wymagane do przeprowadzenia jakichkolwiek instalacji pakietów.

Gdybyśmy teraz chcieli na innym komputerze zainstalować wymagane przez nasz projekt pakiety wykonujemy polecenie:

```
pip install -r requirements.txt
```

To polecenie przeczyta sobie plik i zainstaluje pakiety dokładnie w takich wersjach jak wcześniej zostały zapisane.

Oczywiście istnieją sposoby na określanie zakresów wersji pakietów, można też w ogóle nie pisać wymaganej wersji. Tego jednak nie polecam, wraz z biegiem czasu może się okazać, że nasz projekt nie działa z najnowszą biblioteki XYZ albo nie współpracuje z innym wymaganym pakietem.

Dla świętego spokoju warto wiec określać wersję narzędzi w miarę dokładnie.

Nasz pierwszy projekt w Django

Django zaopatruje nas w polecenie `django-admin`, które pozwala na tworzenie nowych projektów, appek i inne działania.

Aby stworzyć nowy projekt wykonajmy polecenie (warto zauważyć kropkę na końcu, która wskazuje na aktualny katalog):

```
django-admin startproject libraryproj .
```

W aktualnym katalogu powinny się pojawić następujące pliki:

```
libraryproj/
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
venv/
manage.py
requiremets.txt
```

- *libraryproj/* - katalog z podstawowymi ustawieniami naszego projektu, najważniejszy plik to `settings.py`, to tam znajdziemy ustawienia projektu. Innym ważnym plikiem jest `urls.py` w którym to podane są URLe do widoków aplikacji - więcej o tym już niebawem.
- *venv/* - katalog z wirtualnym środowiskiem, które wcześniej utworzyliśmy
- *manage.py* - plik do zarządzania projektem, coś w rodzaju manage-scriptu, więcej o nim za chwilę
- *requirements.txt* - plik z wymaganymi zależnościami naszego projektu

Uruchommy więc nasz projekt!

```
python manage.py runserver
```

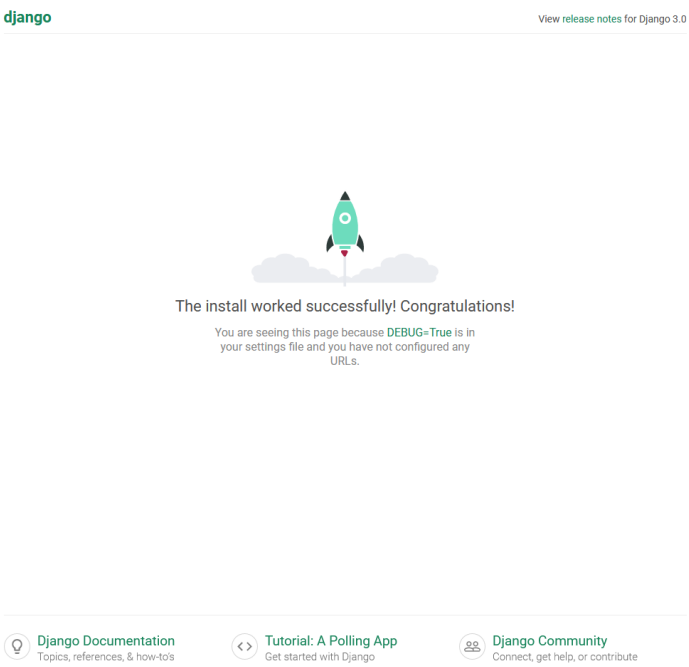
Naszym oczom powinno się ukazać coś takiego:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

Django version 3.0.5, using settings 'libraryproj.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Przejdźmy zatem do przeglądarki jak proponuje wiadomość w terminalu. Adres to: http://127.0.0.1:8000/



Tym samym właśnie napisaliśmy "Hello world" w Django! Nasza aplikacja totalnie nic nie robi, ale czy na pewno? Jeśli ujrzeliście taki widok w przeglądarce, to oznacza, że instalacja zakończyła się pomyślnie i możemy nareszcie przejść do tworzenia aplikacji.

URL Resolver

Kiedy serwer otrzymuje żądanie (np. z naszej przeglądarki internetowej) przekazuje je do Django.

Zaglądnijmy teraz do pliku `urls.py`, znajdziemy tam coś takiego:

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Django posiada coś co się nazywa *url resolver* - to mechanizm rozpoznawania i rozwiązywania adresów URL. Prościej mówiąc - lista z adresami URL, które dotyczą aplikacji. Jeśli adres URL zapytania pasuje do jakiegoś adresu zadeklarowanego w aplikacji, to żądanie zostaje przekazane do odpowiedniej funkcji (konkretnego **widoku**) wskazanego przez aplikację.

To właśnie widoki są sercem logiki aplikacji w Django. Wykorzystują one warstwę modelu powiązana z bazą danych do zapisywania oraz odczytywania danych. Natomiast szablony HTML są zapełniane odpowiednimi informacjami dostępnymi w widoku i następnie taki wyrenderowany HTML zostaje zwrócony do serwera i wreszcie do użytkownika aplikacji (do przeglądarki).

Jest to opis znacznie uproszczony i prawdopodobnie dla większości niezrozumiały. Nic nie szkodzi. W najbliższym czasie poznamy co to wszystko oznacza w praktyce.

Nasza pierwsza aplikacja

W środowisku Django przyjęło się, że każdy projekt zrzesza wiele mniejszych aplikacji (Django apps). Rolą aplikacji jest zapewnienie konkretnej funkcjonalności, w miarę możliwości niezależnej od innych. Można na to patrzeć jak na swego rodzaju *pluginy*. Takie appki w idealnym świecie można by wydzielać i przenosić między projektami.

Stwórzmy więc pierwsza aplikację, ja na cele tego konspektu będę tworzyła mini-bibliotekę na książki.

```
django-admin startapp books
```

Powyższe polecenie stworzy nowy katalog z kilkoma plikami. Omówimy je poniżej.

```
books/
    migrations/
    admin.py
    apps.py
    models.py
    tests.py
    views.py
libraryproj/
    ...
venv/
manage.py
requiremets.txt
```

- *migrations/* - katalog na migracje bazodanowe, czyli pliki, które opisują jak schemat bazy danych się zmieniał pomiędzy kolejnymi wersjami aplikacji. Tutaj znajdziemy wyłącznie pliki automatycznie generowane przez Django. W praktyce - niekiedy zachodzi potrzeba napisania migracji danych i to tutaj się je tworzy (poza zakresem tego kursu).
- *admin.py* - plik, w którym deklarujemy jak ma wyglądać nasz panel administracyjny, w praktyce opisuje się tutaj które modele i jakie pola modeli mają być edytowalne.
- *apps.py* - plik deklarujący podstawowe informacje o aplikacji, takie jak nazwa "dla ludzi"
- *models.py* - jeden z ważniejszych plików, to tutaj znajdują się informacje jak wyglądają dane przechowywane w bazie danych przez aplikacji. W naszym przypadku będą to książki, więc tutaj znajdziemy model opisujący pojedynczą książkę (spis pól jak tytuł, autor, data wydania, liczba stron...)
- *tests.py* - tak, testy :)
- *views.py* - wspomniane wcześniej *widoki* zamieszczane są w tym pliku. Dzięki logice tutaj umieszczonej Django będzie w stanie wyświetlić nasze strony internetowe.

Nasz pierwszy widok

Otwórzmy plik `views.py` umieszczony w `books/views.py`.

```
from django.shortcuts import render

# Create your views here.
```

Na początek posłużymy się prostą funkcją, która jako argument przyjmuje zapytanie (które przyjdzie z naszej przeglądarki). Funkcja ta powinna jakoś odpowiedzieć.

Na górze pliku zaimportujmy generyczną odpowiedź HTTP:

```
from django.http import HttpResponse
```

Następnie napiszmy prostą funkcję, która zwraca jedynie napis "Witaj świecie":

```
def hello_world(request):
    return HttpResponse("Witaj świecie!")
```

Nasz plik `views.py` powinien wyglądać następująco:

```
from django.http import HttpResponse
from django.shortcuts import render


# Create your views here.
def hello_world(request):
    return HttpResponse("Witaj świecie!")
```

Jednak to nie wszystko. Jeśli wejdziemy http://127.0.0.1:8000/ nie ukaże się nam powitanie świata.

Aby nasza aplikacja była uwzględniona przez Django trzeba ją dodać do zainstalowanych. W tym celu otwieramy plik `settings.py`, o którym była mowa wcześniej. To tutaj znajdują się ustawienia projektu, w tym lista zainstalowanych aplikacji.

Odszukujemy fragmentu z listą o nazwie `INSTALLED_APPS`:

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'books',  # <-- dopisane teraz :)
]
```

I dopisujemy naszą aplikację, w moim przypadku to jest `books`.

W tej chwili nasz aplikacja została dołączona do projektu. Jednakże widok, który napisaliśmy nie został podpięty pod żaden adres URL, więc nie da się go jeszcze wyświetlić. Naprawmy to.

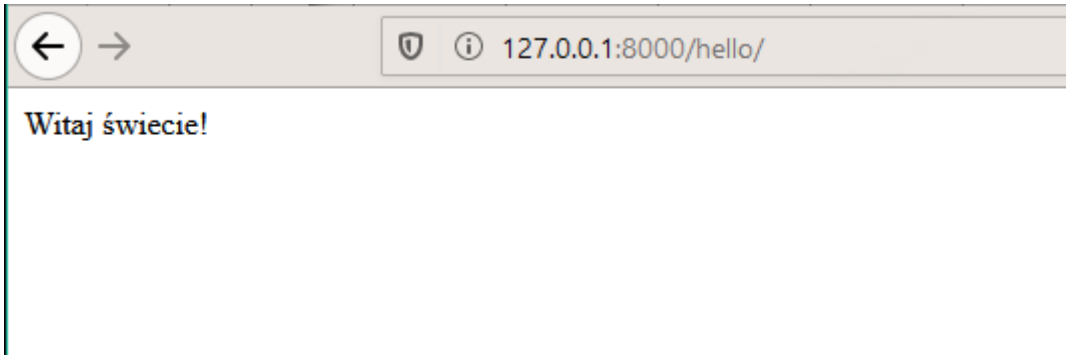
Otwórzmy `urls.py` w katalogu projektu i dodajmy naszą funkcję widoku:

```
from books import views  # NEW

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', views.hello_world),  # NEW
]
```

W powyższym kodzie `# NEW` zaznaczyłam 2 dodane linijki. Reszta powinna zostać bez zmian.

Następnie udajmy się do przeglądarki pod adres http://127.0.0.1:8000/hello/:



Działa! 😊

Automatyczne przeładowanie aplikacji

W międzyczasie mogliście zauważyć (jeśli aplikacja była uruchomiona), ze doszło do automatycznego przeładowania aplikacji:

```
.../views.py changed, reloading.
Performing system checks...

Watching for file changes with StatReloader
```

Jest to mechanizm automatycznego przeładowania, który jest domyślnie włączony i "nasłuchuje" na zmiany w plikach projektu i aplikacji.

Dzięki temu nie musimy ręcznie zatrzymywać i uruchamiać serwera od nowa po każdej wprowadzonej zmianie. Prawda, że przydatne?

Nasz pierwszy "pełnoprawny" widok

Ten poprzedni pierwszy widok był tylko nudnym tekstem, spróbujmy tym razem użyć odpowiedzi HTML.

Wróćmy do `views.py`:

```
from django.shortcuts import render


# Create your views here.
def hello_world(request):
    return render(request, template_name="hello.html")  # NOWE
```

Tym razem użyjmy funkcji `render`, która już była wcześniej zaimportowana. Przyjmuje ona 2 argumenty wymagane:

- *request* - czyli przychodzące zapytanie o stronę
- *template_name* - nazwę szablonu HTML, który ma być wyrenderowany jako odpowiedź

Oprócz tego można też przekazać kontekst, ale do tego wrócimy za chwilę.

Oczywiście plik `hello.html` nie istnieje, więc musimy go wpierw utworzyć.

W katalogu naszej aplikacji stwórzmy katalog `templates/`, to tutaj będziemy umieszczać pliki HTML dotyczące aplikacji `books`. W tym katalogu utwórzmy plik `hello.html` i zamieśćmy tam dowolny HTML, np `<h1>Witaj świecie</h1>`

```
books/
    migrations/
    templates/
        hello.html
    ...
libraryproj/
    ...
venv/
manage.py
requiremets.txt
```

Następnie odświeżmy stronę w przeglądarce (http://127.0.0.1:8000/hello/ jeśli ktoś zamknął kartę).

Powinniśmy ujrzeć napis, ale tym razem jako nagłówek `<h1>`.

Bazy danych

Wróćmy na chwilę do pliku `settings.py`. To tutaj znajdują się kluczowe ustawienia projektu.

Wśród nich jest również informacja o bazie danych:

```
# Database
# https://docs.djangoproject.com/en/3.0/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Jak może zauważyliście, po uruchomieniu projektu w katalogu głównym pojawił się plik `db.sqlite3`. To właśnie nasza baza danych. Django domyślnie przechowuje dane w pliku `'db.sqlite3'` w formacie `SQLite3`.

Gdybyśmy w przyszłości chcieli używać innego silnika bazy danych, trzeba będzie zmienić powyższe ustawienie.

Jednak na potrzeby tego projektu zostaniemy przy SQLite - jest proste w obsłudze i nie wymaga instalowania dodatkowego oprogramowania. Django wspiera jednak również inne serwery baz danych, jak na przykład: PostgreSQL i MySQL.

Podczas uruchomiania projektu można było zauważyć w terminalu poniższą wiadomość:

```
System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until you apply
the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
```

Django informuje nas, że mamy niezaaplikowane migracje. W skrócie oznacza to, że stan naszej bazy danych nie odzwierciedla tego, czego oczekuje aplikacja. Przykładowo aplikacja `django.contrib.auth` dostarcza model użytkownika, którego użyjemy za chwilę.

Model użytkownika ma takie pola jak *username*, *imię*, *nazwisko*, *hasło*. Django oczekuje, że tak zdefiniowany model będzie miał odpowiednią tabele w bazie danych z kolumnami odpowiedniego typu (na dane tekstowe - tekstowe, na liczby - kolumnę typu liczbowego, na pliki inny itp.).

Żeby lepiej zrozumieć konsekwencje spróbujmy wejść pod adres http://127.0.0.1:8000/admin/

Zobaczymy formularz logowania. Jednak nie mamy przecież ani loginu, ani hasła potrzebnych do przejścia dalej. Te informacje byłyby zapisane w bazie danych.

Migracja bazy danych

Zmigrujmy zatem naszą bazę danych (tym samym stworzone zostaną tabele na użytkowników i nie tylko).

Wszelkie akcje związane z zarządzaniem aplikacją można wykonać używając skryptu `manage.py` (wcześniejsze tworzenie appki też):

```
(venv) > python manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying sessions.0001_initial... OK
```

Tworzenie super-użytkownika

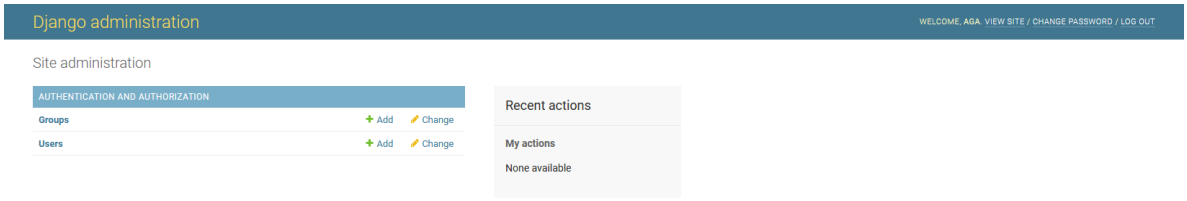
Aby stworzyć super-użytkownika, który będzie miał wszystkie uprawnienia w aplikacji możemy użyć polecenia pomocniczego:

```
(venv) > python manage.py createsuperuser
Username (leave blank to use 'admin'): aga
Email address:
Password:
Password (again):
Superuser created successfully.
```

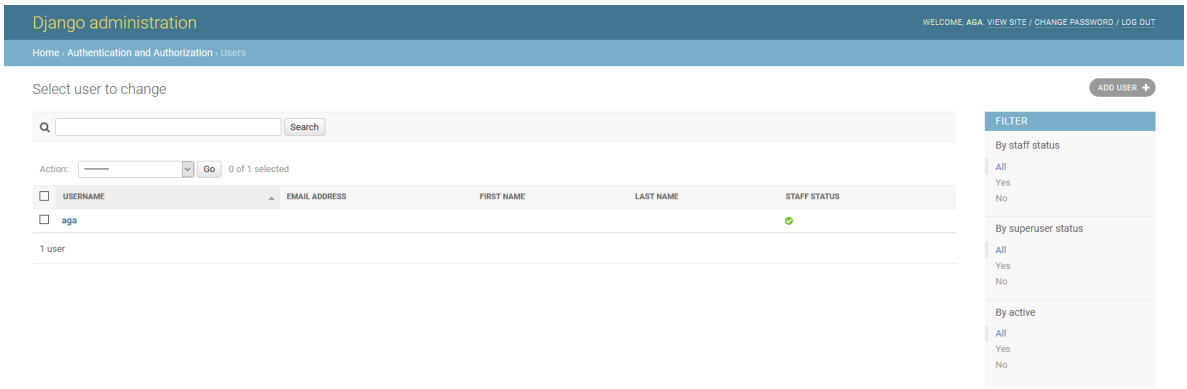
Teraz będzie można się zalogować do panelu administracyjnego.

Panel administracyjny

Przejdźmy więc na stronę: http://127.0.0.1:8000/admin/login/



Można tu miedzy innymi zarządzać użytkownikami (klikając w `Users` przejdziemy do listy użytkowników):



Wszystkie znajdujące się tutaj widoki są automatycznie generowane. Na następnych zajęciach zajmiemy się dodawaniem modeli i podepniemy je pod ten panel.

Kontekst w szablonach HTML

Wróćmy na chwilę do naszego przykładu z "Witaj świecie" w HTML. Gdybyśmy chcieli wyświetlić informacje o aktualnie zalogowanym użytkowniku możemy umieścić w HTML taki oto kod:

```
<p>
    Aktualny użytkownik to: {{ request.user }}
</p>
```

W przeglądarce powinniśmy zobaczyć:

```
 Aktualny użytkownik to: AnonymousUser
```

lub jeśli się zalogowaliśmy przez panel admina - login naszego użytkownika zamiast `AnonymousUser`.

Generalnie za pomocą `{{ zmienna }}` podwójnych nawiasów klamrowych możemy wypisywać dane dostępne w kontekście HTML. Co to znaczy? Myślmy o kontekście jak o słowniku, czyli strukturze klucz-wartość. Pod każdym kluczem kryją się jakieś dane. Wiele z nich jest zapewnionych domyślnie przez framework, jak np. wyżej wykorzystane `request.user`.

Jednak to nie wszystko. Załóżmy, że chcemy dodać do kontekstu własne dane, np. aktualną godzinę.

Wróćmy zatem do naszej funkcji widoku w `views.py`:

```
from datetime import datetime  # NEW

from django.shortcuts import render


# Create your views here.
def hello_world(request):
    our_context = {"time": datetime.now()}  # NEW
    return render(request, template_name="hello.html", context=our_context)  #NEW
```

W powyższym kodzie stworzyliśmy słownik `our_context`, w którym umieściliśmy aktualną datę i godzinę. Następnie przekazaliśmy ten słownik jako `context` do funkcji `render()`.

Efekt?

W HTML możemy teraz wypisać zmienną `{{ time }}` używając podwójnych nawiasów klamrowych.

```
<p>Aktualny czas: {{ time }}</p>
```

To właśnie prze kontekst będziemy przekazywać informacje z tak zwanego "backendu" do "frontendu". Będą to np. zapisane w naszej bazie danych książki i inne informacje.

Kontynuacja

Praca samodzielna:

- wykorzystaj HTML z poprzednich zajęć i podepnij do tej aplikacji
- można zrobić kilka podstron w ramach ćwiczenia
- warto zrobić "menu" do poruszania się między nimi
- można wykorzystać informacje o użytkowniku z `{{ user.is_authenticated }}`
- jeśli wystąpią problemy z plikami statycznymi (CSS, JS): https://docs.djangoproject.com/en/3.0/howto/static-files/