**Django - szkolenie 4**

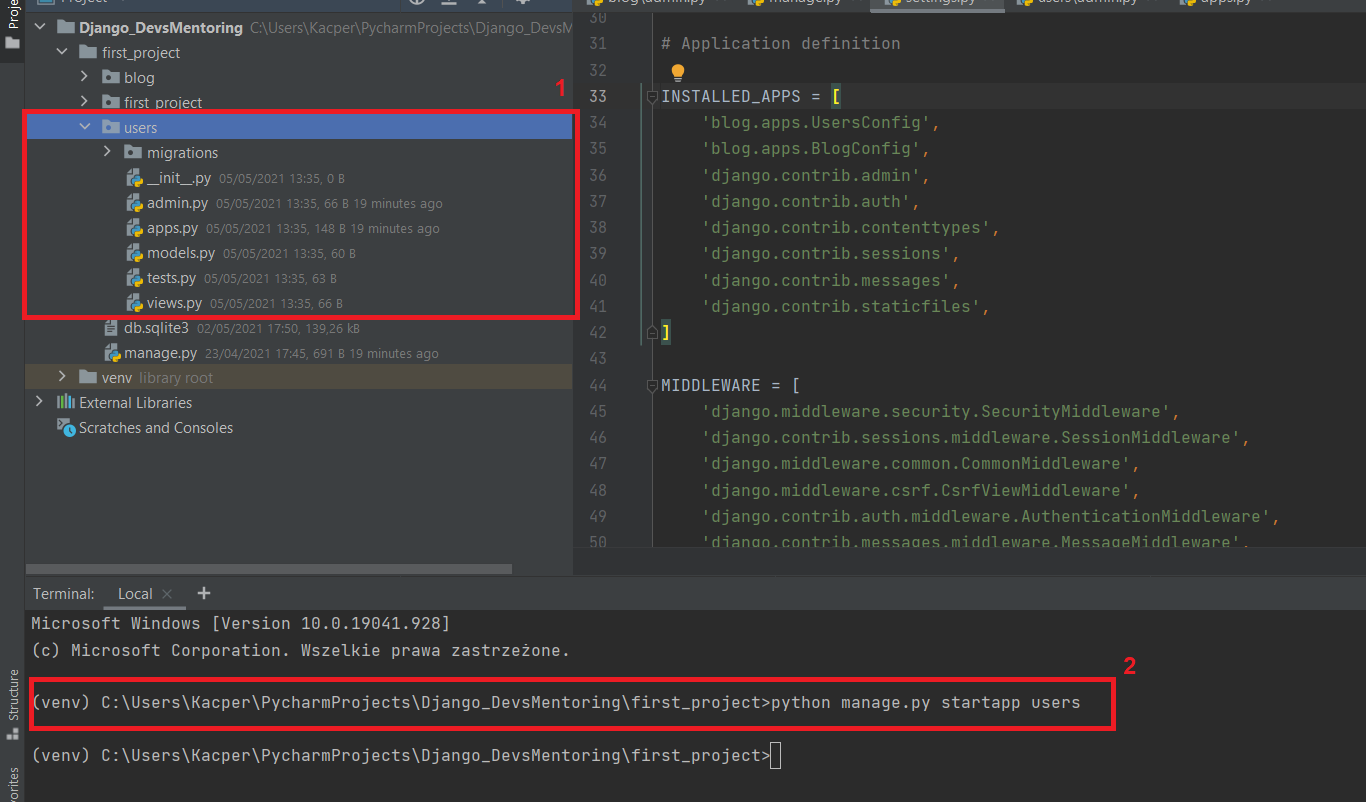
**Panel rejestracji i logowania.**

**Wstęp**

W szkoleniu tym zajmiemy się dodaniem do Naszego projektu nowej aplikacji - users, która odpowiadać będzie za m.in sterowanie logiką związaną z rejestrowaniem/logowaniem użytkowników na Naszym blogu. Dodamy zatem do Naszego serwisu kolejne podstrony - login oraz register. Tak jak wspomniałem, logika tych podstron i całego procesu walidowania użytkownika, zostanie zaprogramowana w nowej aplikacji, którą nazwiemy users. Nie przedłużając zatem, zacznijmy działać!

**Utworzenie nowej aplikacji**

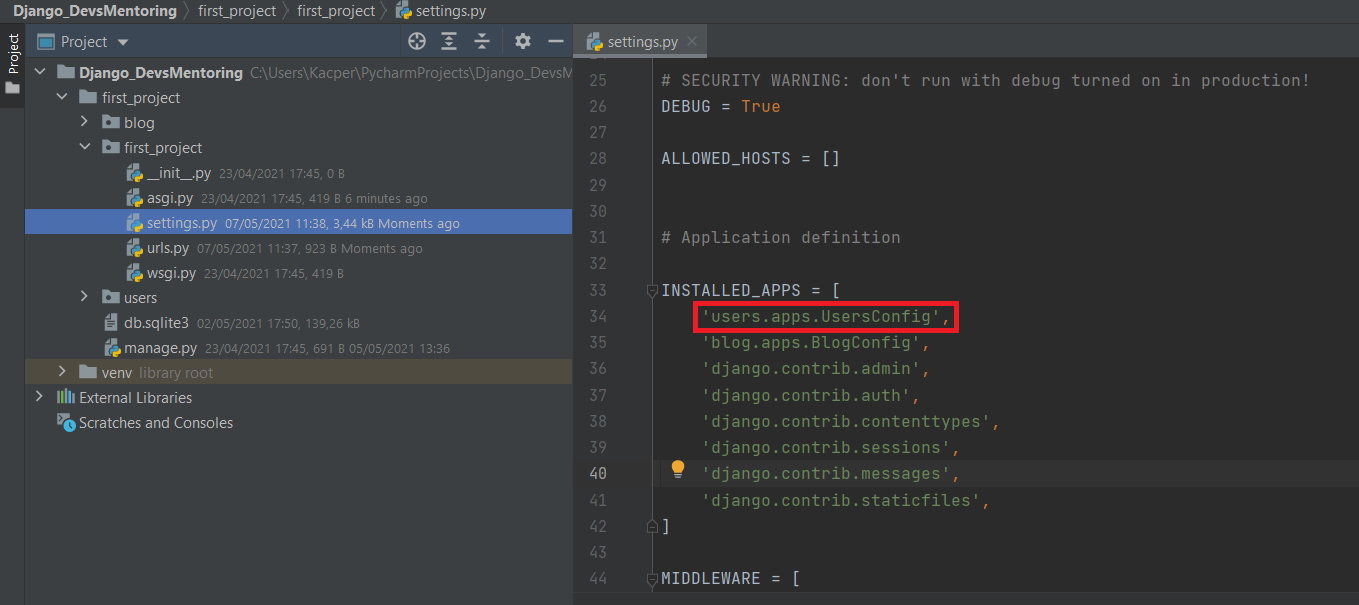
Aby utworzyć aplikację users, w której zarejestrujemy endpointy register oraz login, wykorzystamy znane Nam już polecenie - python manage.py startapp <nazwa\_aplikacji>. Pamiętaj, aby polecenie uruchomić z poziomu głównego katalogu projektu.



Po wpisaniu polecenia (2), w efekcie powinniśmy otrzymać nowo utworzony katalog z aplikacją - users (1).

**Skonfigurowanie aplikacji**

Aby mieć już całą konfigurację z głowy, do listy INSTALLED\_APPS z pliku first\_project/settings.py , dodamy zapis: ‘users.apps.UsersConfig’. Jeżeli nie pamiętasz, czemu służyły te dodatkowe linie kodu, wróć do wcześniejszych szkoleń, w których konfigurowaliśmy aplikację blog.

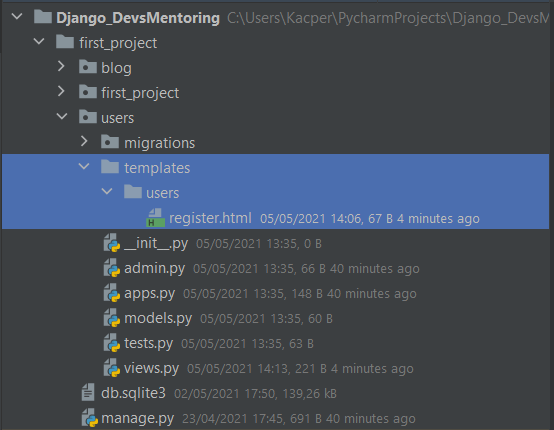


**Zbudowanie pierwszego widoku**

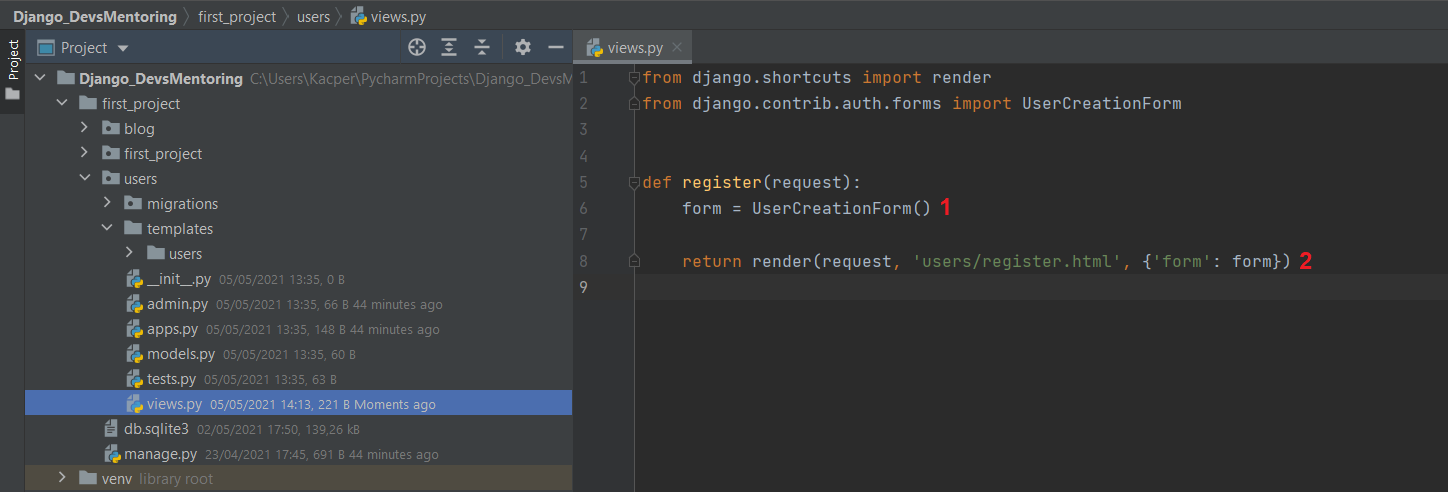
Po dodaniu i skonfigurowaniu aplikacji users, nadszedł czas na zbudowanie pierwszego widoku i powiązanego z nim endpointa. Będzie to podstrona /register, w której wyświetlać się będzie panel rejestracji. Aby zbudować taki panel w szybki i wygodny sposób, wykorzystamy wbudowany w Django szablon UserCreationForm. Dzięki takiemu rozwiązaniu, pozostanie Nam tylko dopracowanie wyglądu tej podstrony bez budowania koła na nowo i tworzenia od początku całego formularza.

Zacznijmy zatem od dodania katalogu templates, w którym trzymać będziemy pliki .html aplikacji users. Przypomnę Ci również, że silnik Django przewiduje, iż oprócz stworzenia samego katalogu templates, konieczne jest utworzenie w nim katalogu o takiej samej nazwie, co Nasza nowo dodana aplikacja (users) i dopiero dodanie do niego plików szablonowych (np. pliku register.html).

Po utworzeniu właściwej struktury, Nasz projekt powinien wyglądać następująco (zaznaczyłem nowo dodane katalogi i pliki):



Póki co zostawmy plik register.html pusty. Wypełnimy go za chwilę, a tymczasem utwórzmy widok register. W tym celu przejdźmy do pliku users/views.py i zapiszmy w nim kod taki jak poniżej:



from django.shortcuts import render

from django.contrib.auth.forms import UserCreationForm

def register(request):

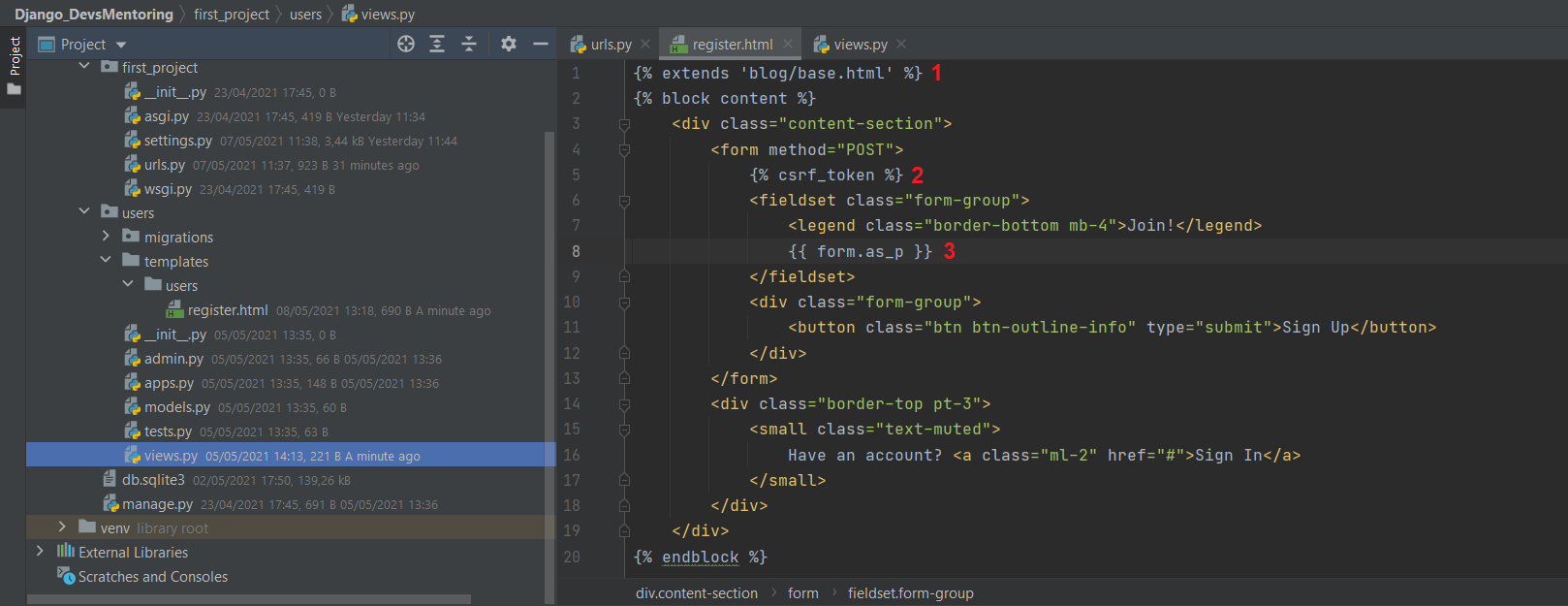
form = UserCreationForm()

return render(request, 'users/register.html', {'form': form})

Tak jak już wspomniałem, aby maksymalnie uprościć tworzenie nowego formularza, wykorzystamy wbudowaną w Django klasę: UserCreationForm. W punkcie (1) widzimy utworzenie jej obiektu, natomiast w (2) przekazanie takiego obiektu do szablonu register.html, w którym za chwilę zakodujemy jego wyświetlanie.

**Okodowanie szablonu**

Wróćmy do wcześniej utworzonego przez Nas pliku users/templates/users/register.html - zakodujmy w nim wygląd i zachowanie podstrony. Kod, który za chwilę omówię, znajdziesz w załączniku Szkolenie 4 - kod końcowy.



{% extends 'blog/base.html' %}

{% block content %}

<div class="content-section">

<form method="POST">

{% csrf\_token %}

<fieldset class="form-group">

<legend class="border-bottom mb-4">Join!</legend>

{{ form.as\_p}}

</fieldset>

<div class="form-group">

<button class="btn btn-outline-info" type="submit"> Sign Up </button>

</div>

</form>

<div class="border-top pt-3">

<small class="text-muted">

Have an account? <a class="ml-2" href="">Sign In</a>

</small>

</div>

</div>

{% endblock %}

Struktura Naszego pliku .html wygląda dokładnie tak jak zostało ukazane na powyższym screenshot-cie. Lwią część kodu stanowi typowo struktura Bootstrapowa - czyli wszelkie divy, fieldsety i pola legend z przypisanymi klasami.

Cyfrą (1) oznaczyłem proces dziedziczenia szablonowego - aby zachować spójność w podstronach, stwierdziłem, że warto, aby plik register.html dziedziczył również po blog/base.html. Zapis {% csrf\_token %} (2) jest zabezpieczeniem, które powinno pojawić się przy tworzeniu formularzy. Token ten uchrania Nas przed atakami CSRF (Cross-Site Request Forgery), czyli jednym słowem - możliwością wysyłania “fake-formularzy” na Nasz serwis.

{{ form.as\_p }} (3) natomiast to generowanie widoku formularza utworzonego i przesłanego do HTML-a z poziomu pliku users/views.py (pamiętasz jeszcze obiekt UserCreationForm?). Suffix .as\_p oznacza tyle, iż każde nowe pole w formularzu (np. Login, Password) ma być generowane jako nowy paragraf (w nowej linii). Dzięki temu otrzymamy właściwe rozłożenie elementów.

**Zarejestrowanie widoku**

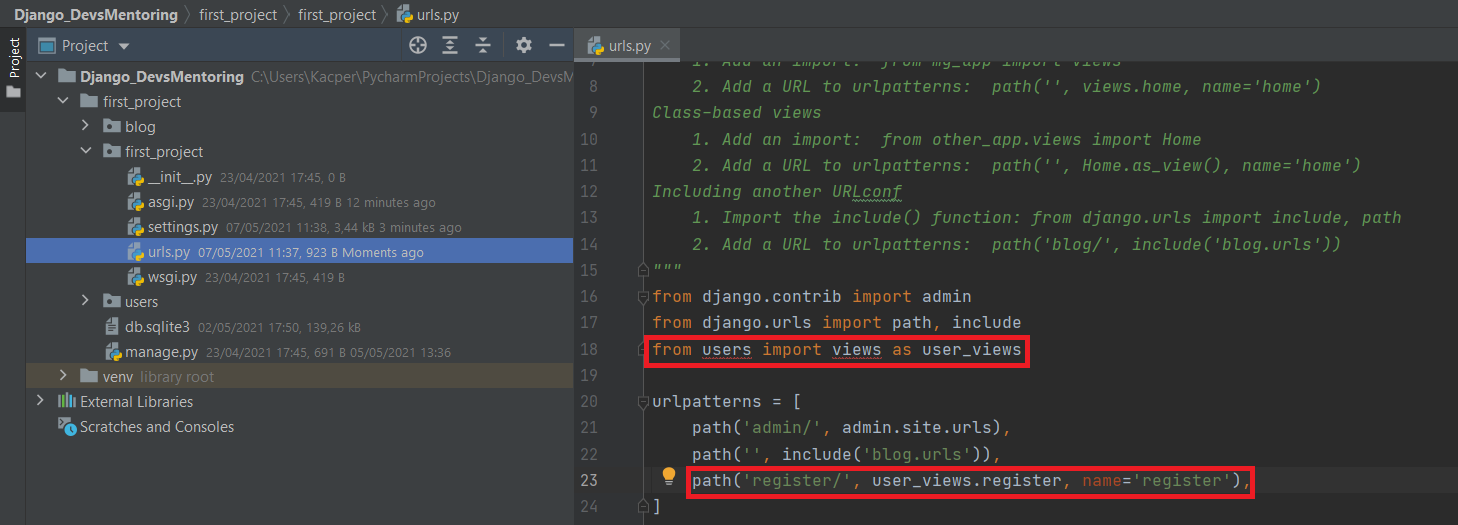
Po dodaniu widoku, skupmy się na określeniu odpowiednich endpointów w pliku first\_project/urls.py. To wszystko po to, aby silnik Django, wiedział do której aplikacji przekierowywać użytkownika (blog czy users) po wpisaniu konkretnego adresu URL.

Nie wiem, czy kojarzysz, ale ze wspomnianego pliku korzystaliśmy już przy rejestrowaniu odnośnika do bloga. Wówczas zapisaliśmy taką linię kodu:

path(‘’, include(‘blog.urls’)), uprzednio oczywiście umieszczając from django.urls import include

Teraz, aby dodać do projektu panel rejestracji, zapiszemy:

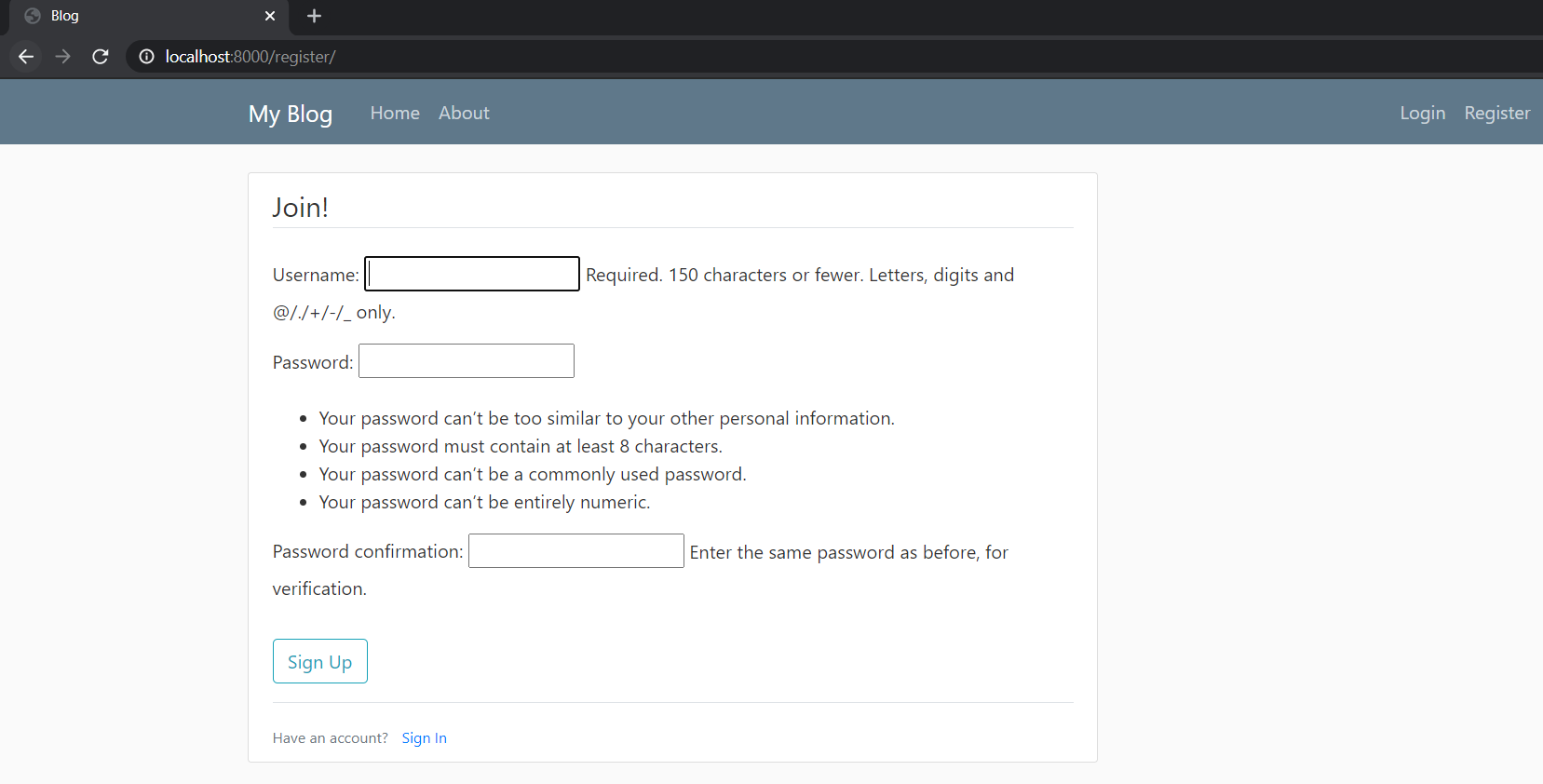
path(‘register/’, user\_views.register, name=’register’) i dodamy from users import views as user\_views



Zapytasz pewnie, dlaczego sposób importowania aplikacji blog i register różni się między sobą (porównaj przytoczone wyżej sposoby na rejestrację ‘’ oraz ‘register/’)...

Odpowiadając - podjąłem się trochę innego sposobu rejestracji endpointa register/, po to aby uzmysłowić Ci, iż istnieją również inne sposoby na dodawanie pewnych ścieżek do konfiguracji tworzonego projektu. W tym przypadku, zamiast tworzyć referencję do całego pliku urls, który znajdowałby się w katalogu users (i w nim bezpośrednio rejestrowalibyśmy konkretne views), przyjmujemy trochę inną strategię i dodajemy jeden pojedynczy view (user\_views.register). Ostateczny efekt jest ten sam, ale nieco inny sposób zapisu w kodzie.

Jeżeli zrobiłeś wszystko zgodnie z założeniami - czyli poprawnie okodowałeś plik .html oraz właściwie skonfigurowałeś nową aplikację, to przejściu do endpointa /register, powinieneś otrzymać efekt taki jak poniżej:



**Jednak to nie wszystko…**

Jeżeli spróbujemy wpisać dane do Naszego formularza i wysłać je po wciśnięciu klawisza Sign Up, to z wiadomych powodów nie otrzymamy żadnego konkretnego rezultatu. To dlatego, iż brakuje Nam zaprogramowanej logiki aplikacji users i jej widoku register. Musimy choćby rozróżniać sytuacje, kiedy użytkownik wyśle formularz (metoda POST), a kiedy będzie chciał po prostu wygenerować jego widok (GET).

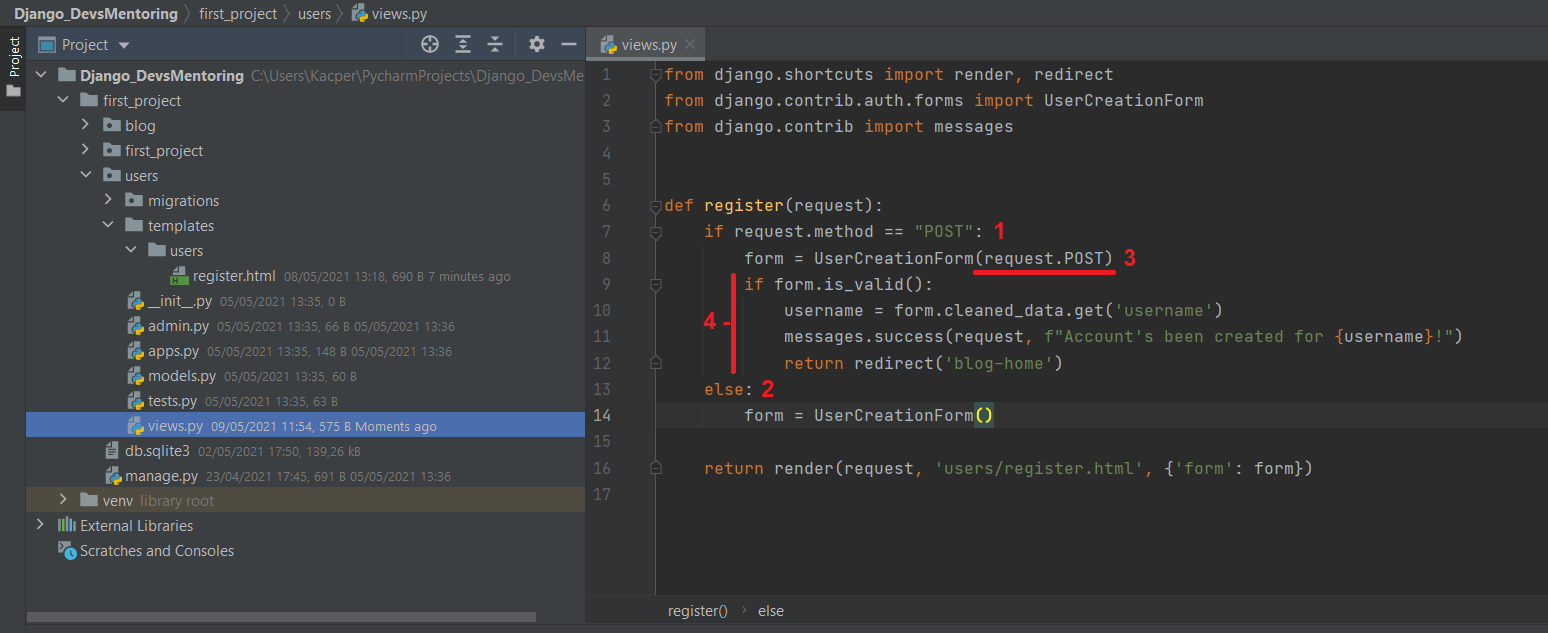
Wróćmy zatem do pliku users/views.py i dodajmy odpowiednią funkcjonalność. Zaczniemy od rozróżnienia metod HTTP wysyłanych do danego endpoint-a oraz pobierania podanych danych z formularza. Następnie dodamy funkcjonalność flashowania wiadomości. Wiadomości będą z różnych kategorii - w zależności od tego, czy będzie to wiadomość o, np. success, failure itd.

Dla bardziej dociekliwych - możemy wyróżnić następujące rodzaje wiadomości

1. DEBUG
2. INFO
3. SUCCESS
4. WARNING
5. ERROR

Określony rodzaj wykorzystywać będziemy w zależności od sytuacji, np. typ SUCCESS wykorzystamy, aby ukazać wiadomość o poprawnym zalogowaniu, ERROR o podaniu błędnych danych itd.

Wprowadźmy wyżej opisane założenia do pliku users/views.py.



from django.shortcuts import render

from django.contrib.auth.forms import UserCreationForm

from django.contrib import messages

def register(request):

if request.method == 'POST':

form = UserCreationForm(request.POST)

if form.is\_valid():

username = form.cleaned\_data.get('username')

messages.success(request, f"Account's been created for {username}!")

return redirect('blog\_home')

else:

form = UserCreationForm()

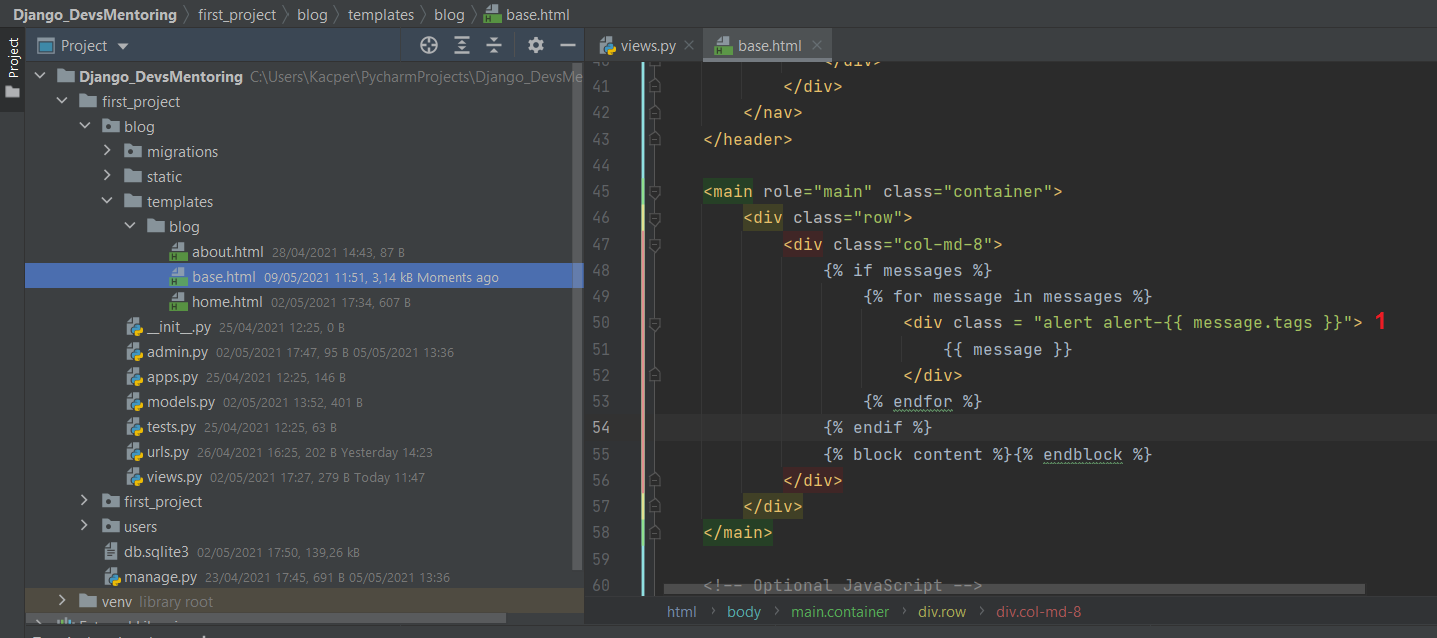
return render(request, 'users/register.html', {'form': form})

# Create your views here.

Punkty (1) i (2) określają, jak zachować ma się strona w zależności od metody wysłanego zapytania HTTP. Warunek if request.method == “POST” zostanie spełniony, gdy użytkownik wyśle formularz. Wówczas tworzymy obiekt UserCreationForm z argumentem request.POST. Następnie sprawdzamy (4), czy formularz ten został poprawnie wypełniony przez użytkownika form.is\_valid() - czyli, czy na przykład, czy hasło spełnia wszystkie podane założenia, czy podane potwierdzenie hasła jest zgodne z właściwym etc.

Jeżeli tak, to wówczas zapamiętujemy podaną w formularzu nazwę użytkownika: username = form.cleaned\_data.get(‘username’) oraz flashujemy wiadomość o succesie: messages.success(request, f”Account’s been created for {username}”). Będąc już przy tej linii kodu, musimy jeszcze na chwilę przejść do szablonu .html, aby wyświetlanie zaflashowanej wiadomości było w ogóle możliwe.

Obsługę wiadomości dodamy w pliku blog/base.html i to właśnie z jego poziomu wyświetlać będziemy wiadomości.



{% load static %}

<!DOCTYPE html>

<html>

<head>

<!-- Required meta tags -->

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap CSS -->

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

<meta charset="UTF-8">

<link rel="stylesheet" type="text/css" href="{% static 'blog/base.css' %}" />

{% if title %}

<title>Blog - {{ title }}</title>

{% else %}

<title>Blog</title>

{% endif %}

</head>

<body>

<header class="site-header">

<nav class="navbar navbar-expand-md navbar-dark bg-steel fixed-top">

<div class="container">

<a class="navbar-brand mr-4" href="/">Django Blog</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarToggle" aria-controls="navbarToggle" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarToggle">

<div class="navbar-nav mr-auto">

<a class="nav-item nav-link" href="{% url 'blog:blog\_home' %}">Home</a>

<a class="nav-item nav-link" href="{% url 'blog:blog\_about' %}">About</a>

</div>

<!-- Navbar Right Side -->

<div class="navbar-nav">

<a class="nav-item nav-link" href="#">Login</a>

<a class="nav-item nav-link" href="register/">Register</a>

</div>

</div>

</div>

</nav>

</header>

<main role="main" class="container">

<div class="row">

<div class="col-md-8">

{% if messages %}

{% for message in messages %}

<div class = "alert alert-{{ message.tags }}">

{{ message }}

</div>

{% endfor %}

{% endif %}

{% block content %}{% endblock %}

</div>

</div>

</main>

<!-- Optional JavaScript -->

<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

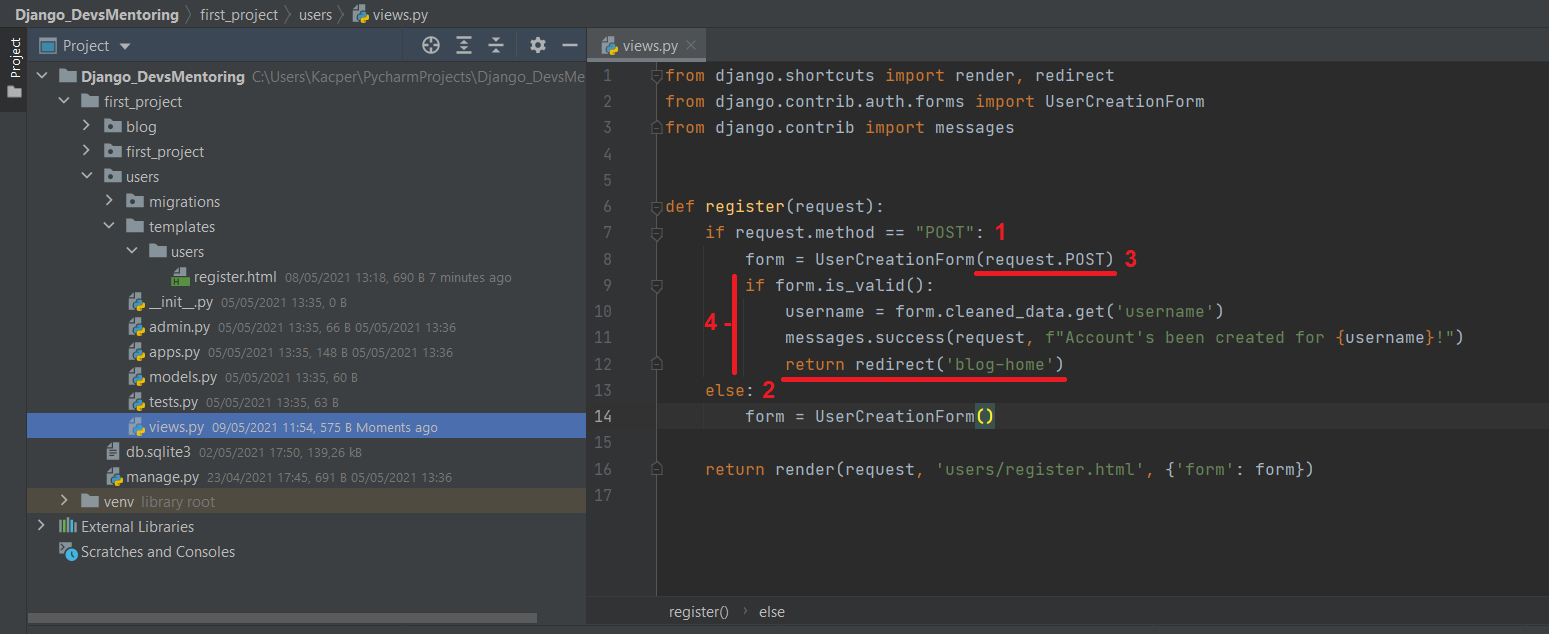
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

</body>

</html>

Sekcję do wyświetlania message dodaliśmy nad blokiem content: {% block content %}{% endblock %}.   
(1) to miejsce, które umożliwia dopasowanie klasy wiadomości w zależności od jej rodzaju. Dzięki temu wiadomości o błędzie zostaną wyświetlone w czerwonym polu, o sukcesie w zielonym itd.

Po dokonaniu takich zmian, wróćmy jeszcze na chwilę do pliku users/views.py. Została Nam do przeanalizowania ostatnia rzecz - przekierowywanie użytkownika do danego endpoint-u po poprawnej rejestracji:



from django.shortcuts import render, redirect

from django.contrib.auth.forms import UserCreationForm

from django.contrib import messages

def register(request):

if request.method == 'POST':

form = UserCreationForm(request.POST)

if form.is\_valid():

username = form.cleaned\_data.get('username')

messages.success(request, f"Account's been created for {username}!")

return redirect('blog:blog\_home')

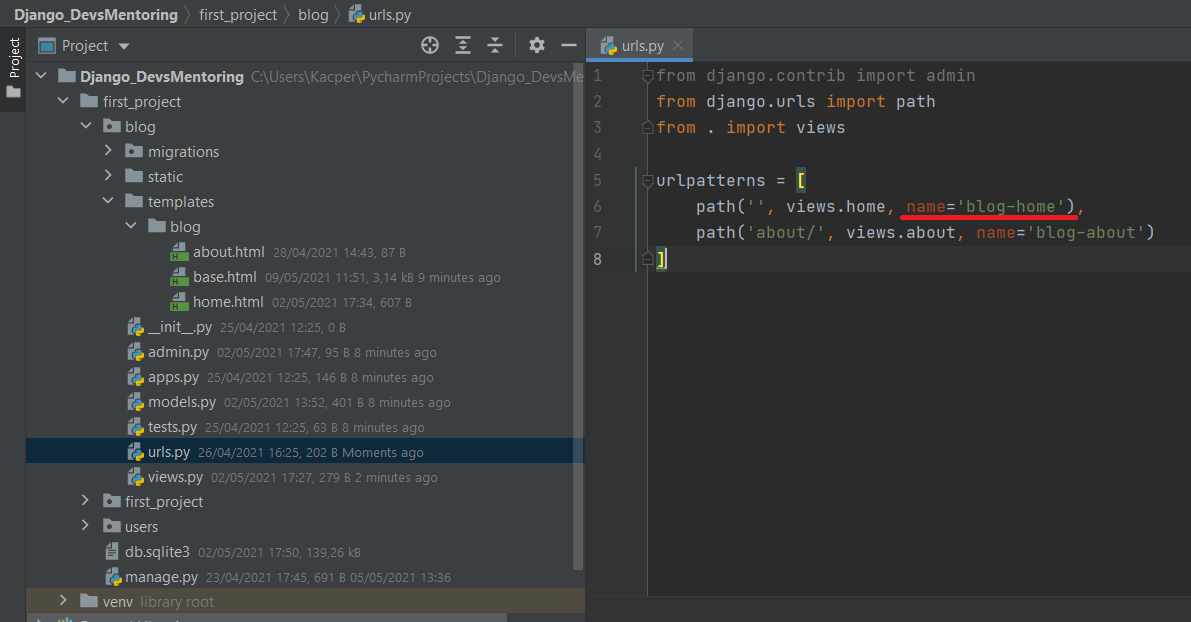
else:

form = UserCreationForm()

return render(request, 'users/register.html', {'form': form})

# Create your views here.

W celu przekierowywania zarejestrowanego usera na podstronę Home, użyliśmy metody redirect. Wewnątrz niej podaliśmy natomiast nazwę widoku, do którego odsyłamy zarejestrowaną osobę. Przypominam Ci, że nazwa blog-home nie wzięła się znikąd - spójrz na zawartość pliku blog/urls.py. Po raz kolejny sprawdza się praktyczność nadawania nazw pewnym ścieżkom w aplikacji:



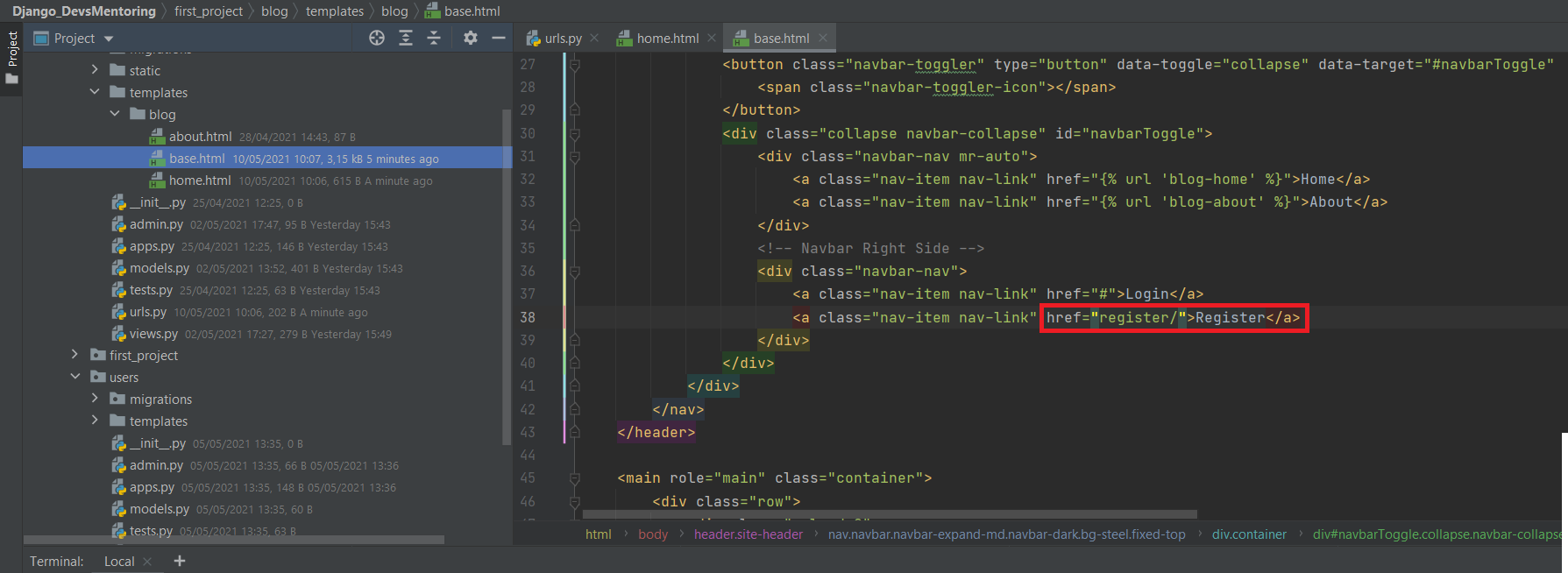
*Plik blog/users.py*

**Uruchomienie**

Po tak przeprowadzonych zmianach, możemy zweryfikować, jaki efekt otrzymamy (dodatkowo zmieniłem href przycisku Register na podstronie blog-home - tak, aby przekierowywał do odpowiedniego endpointa):

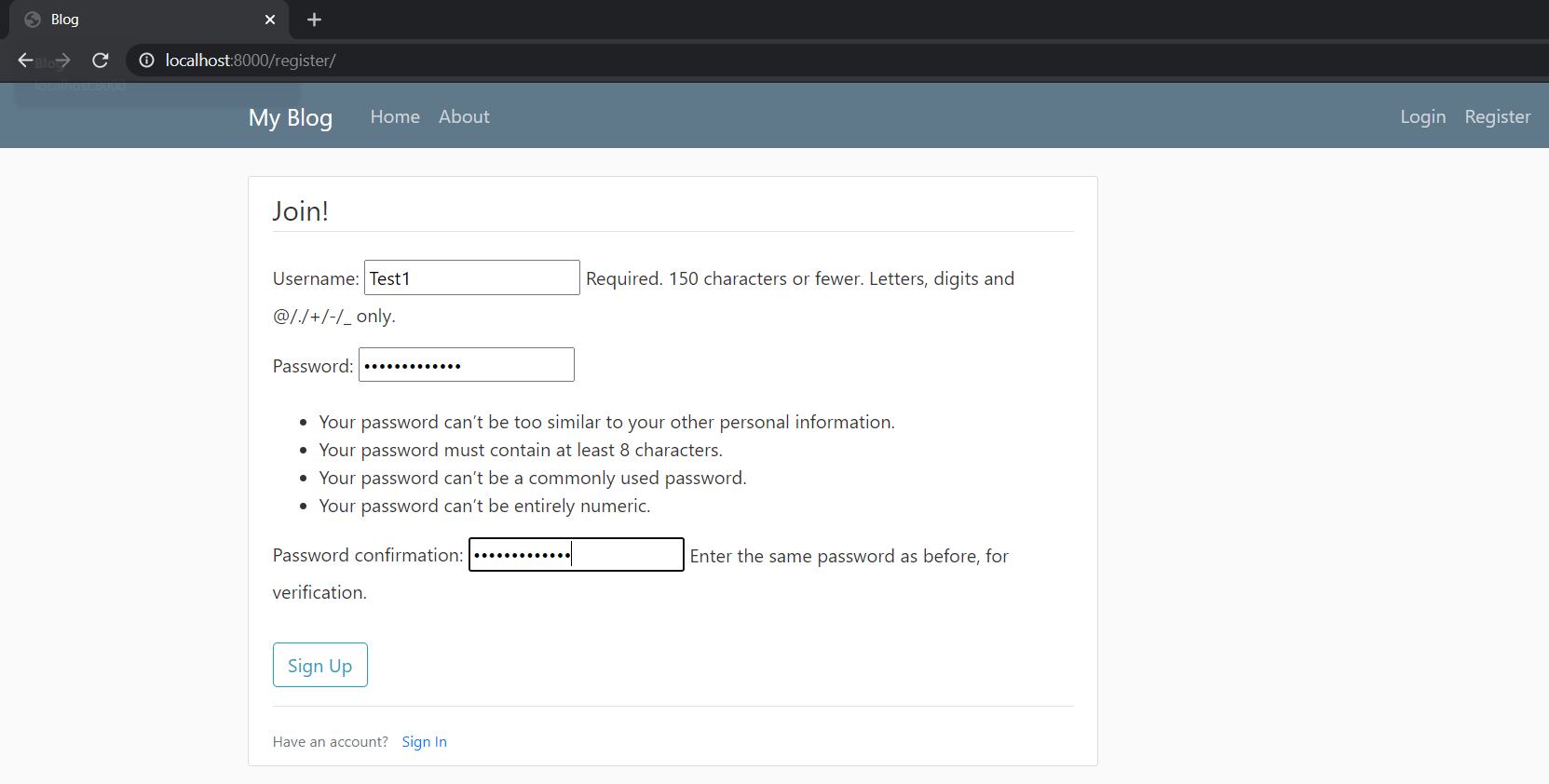


*Navbar*

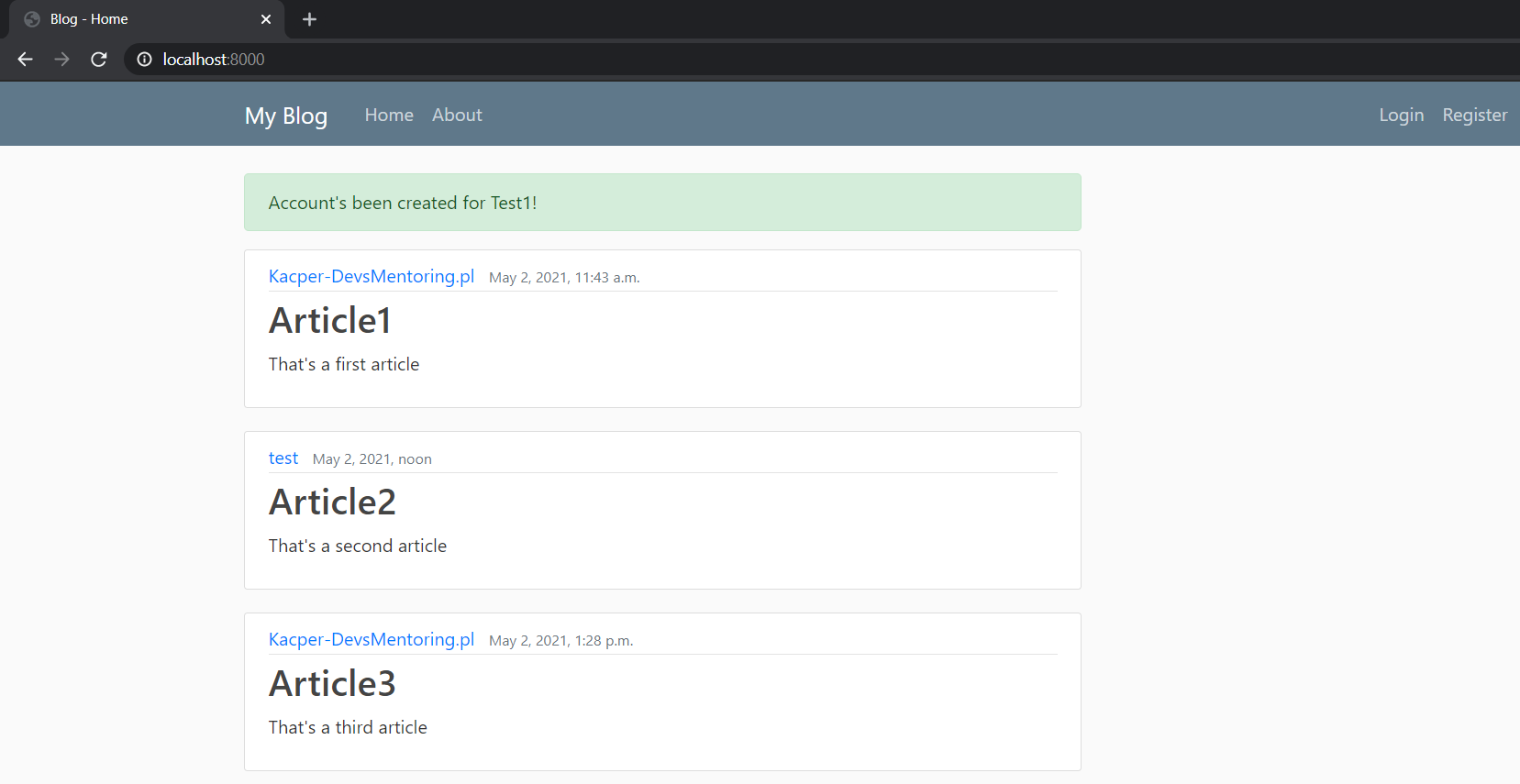


*Plik blog/base.html*

<a class="nav-item nav-link" href="register/">Register</a>



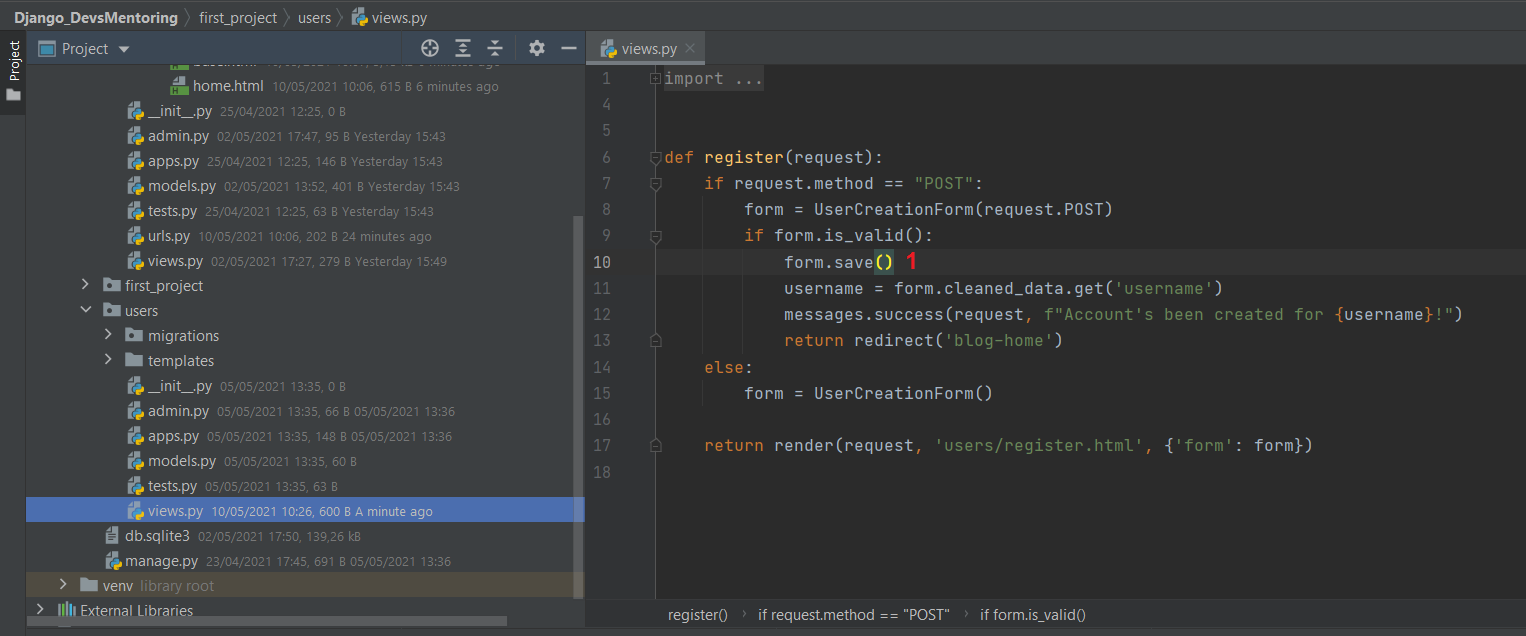
*Efekt zmiany*



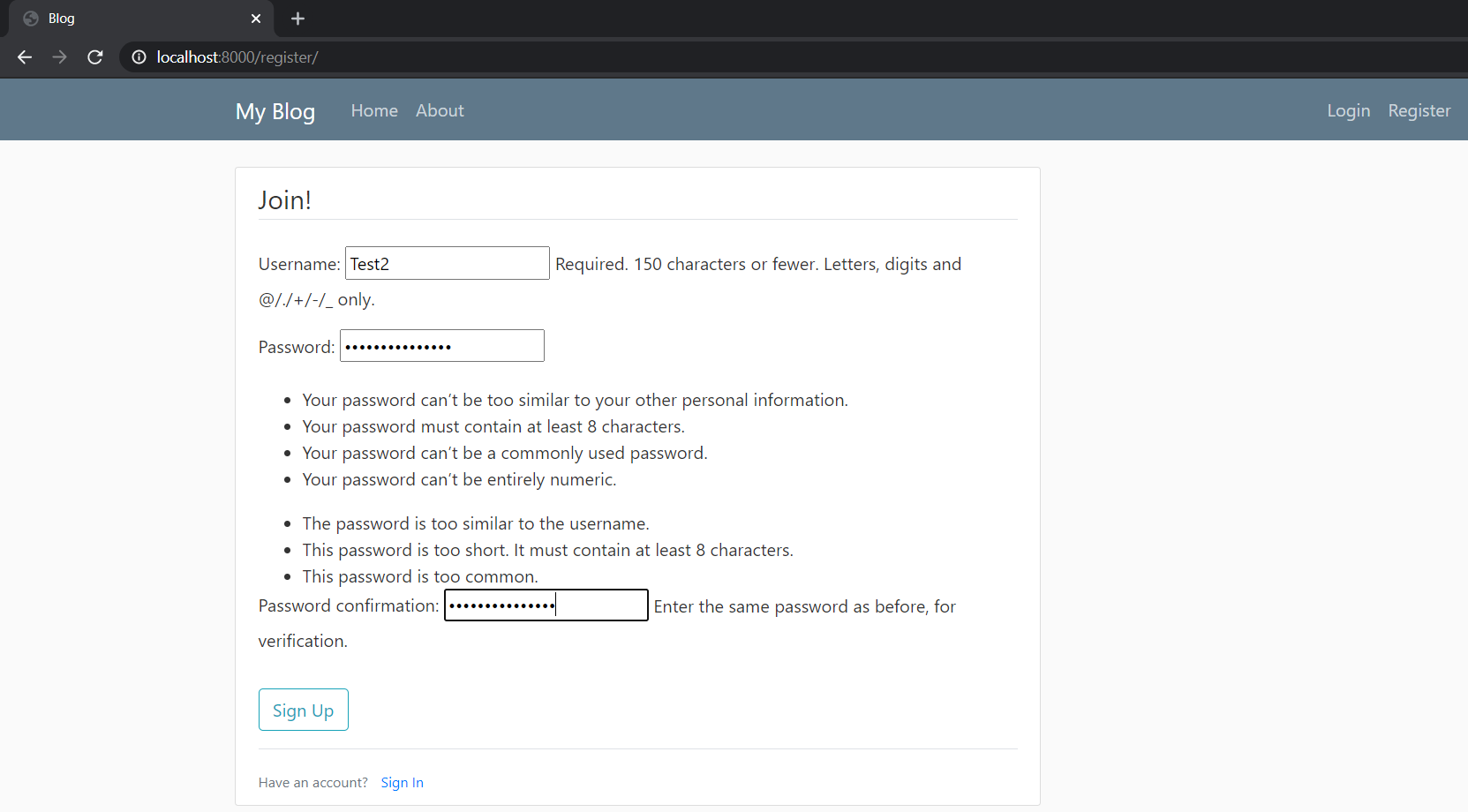
*Efekt po zarejestrowaniu*

**Zapis użytkowników do bazy**

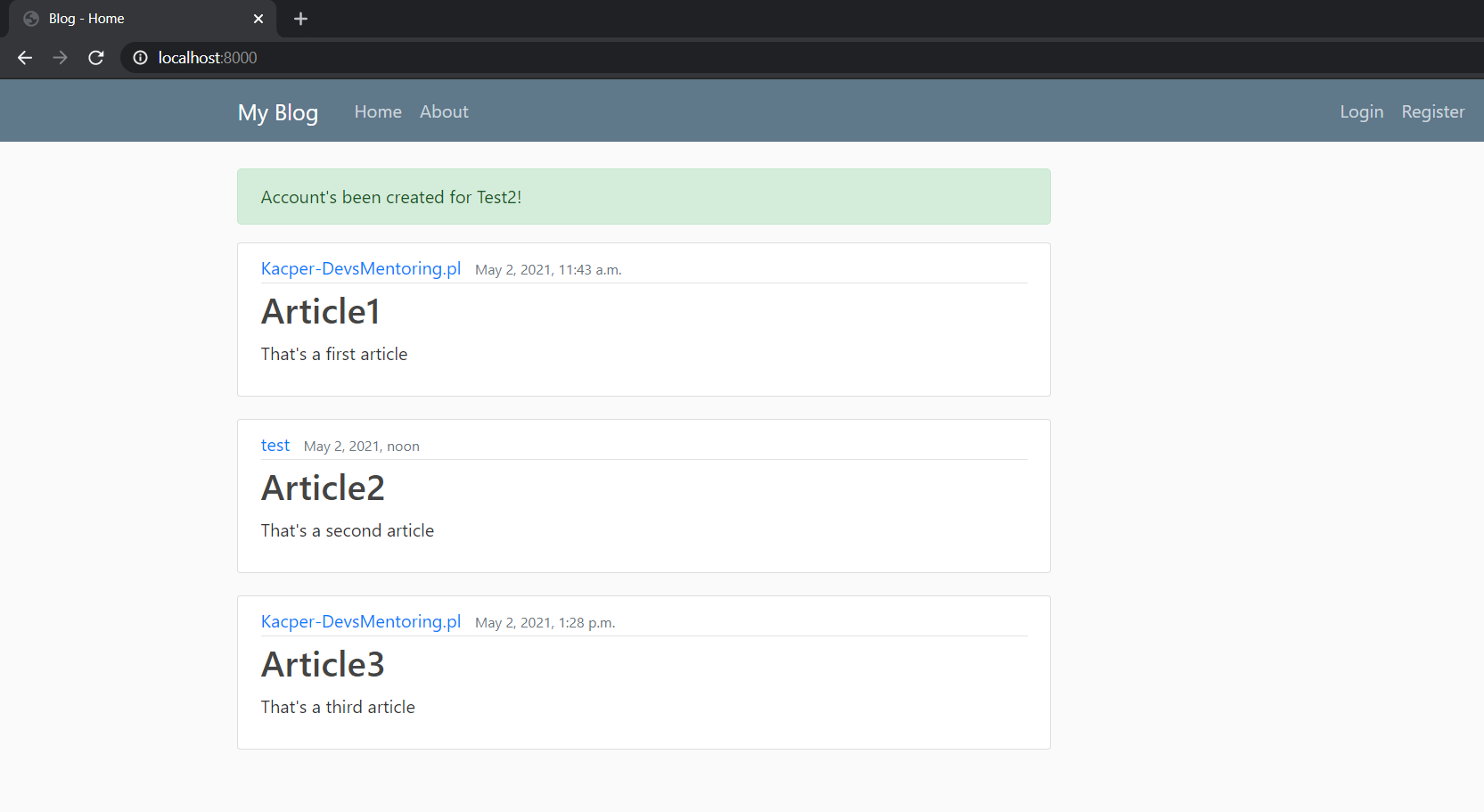
Super, udało Nam się rozpracować funkcjonalność formularzy, ale nadszedł czas na jego integrację z bazą danych. Otóż chcielibyśmy, aby każdy nowo zarejestrowany użytkownik, był automatycznie dodawany do tabeli User w bazie danych. Aby to zrealizować wystarczy tak naprawdę jedna linia kodu - form.save() umieszczona w odpowiednim miejscu.



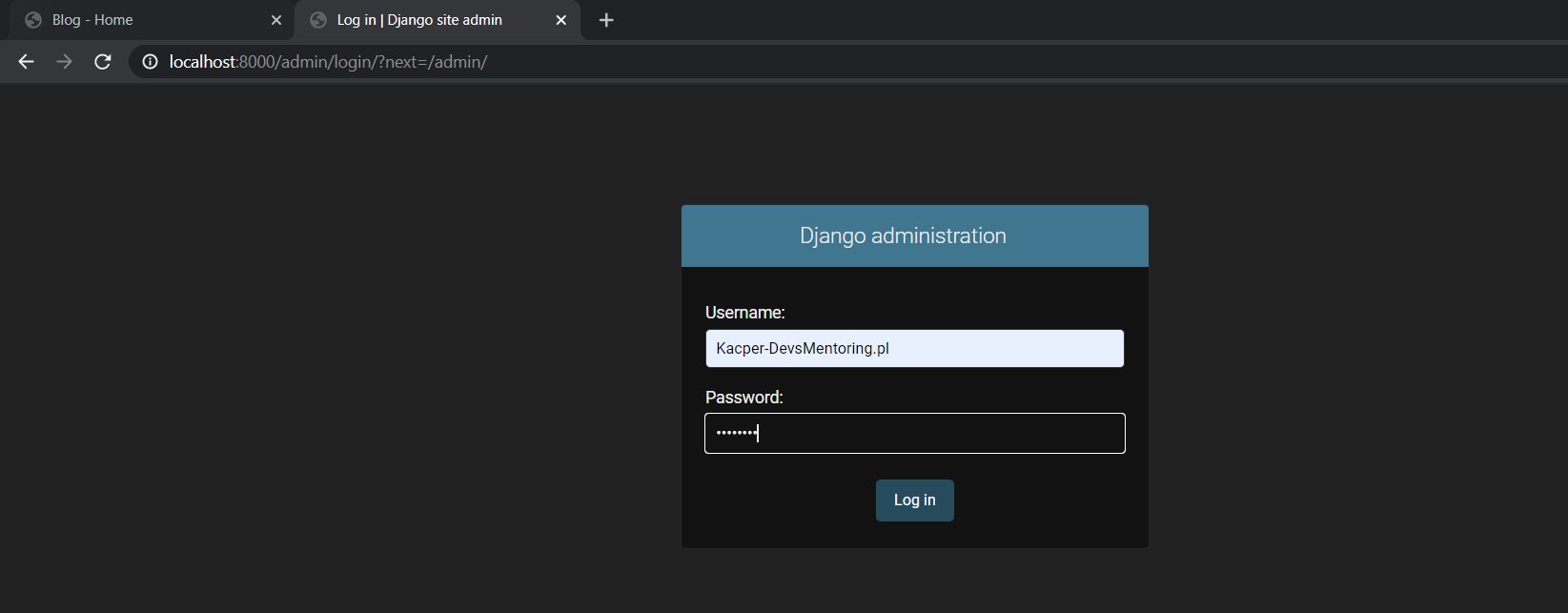
Sprawdźmy, czy po zarejestrowaniu przykładowego użytkownika: Test2 i haśle TestPassword123, faktycznie zostanie on dodany do DB. Zawartość tabeli User, zweryfikujemy standardowo na endpoint-cie /admin Naszej aplikacji.



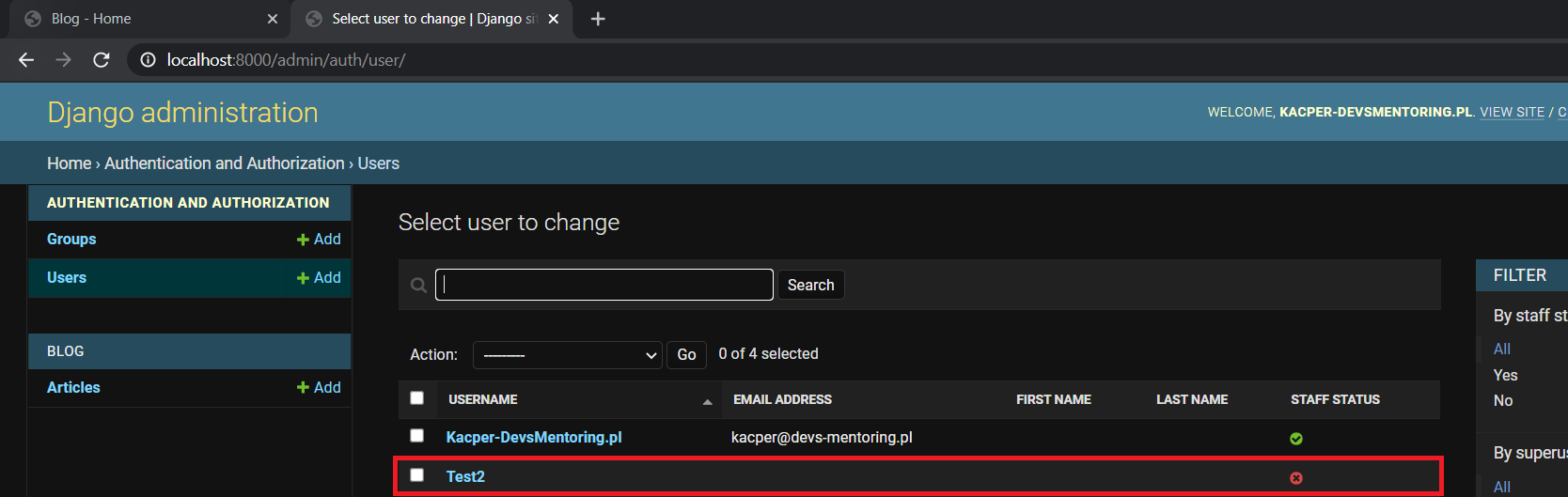
*Podanie danych przy rejestracji*



*Po wysłaniu formularza*



*Zalogowanie do panelu /admin*



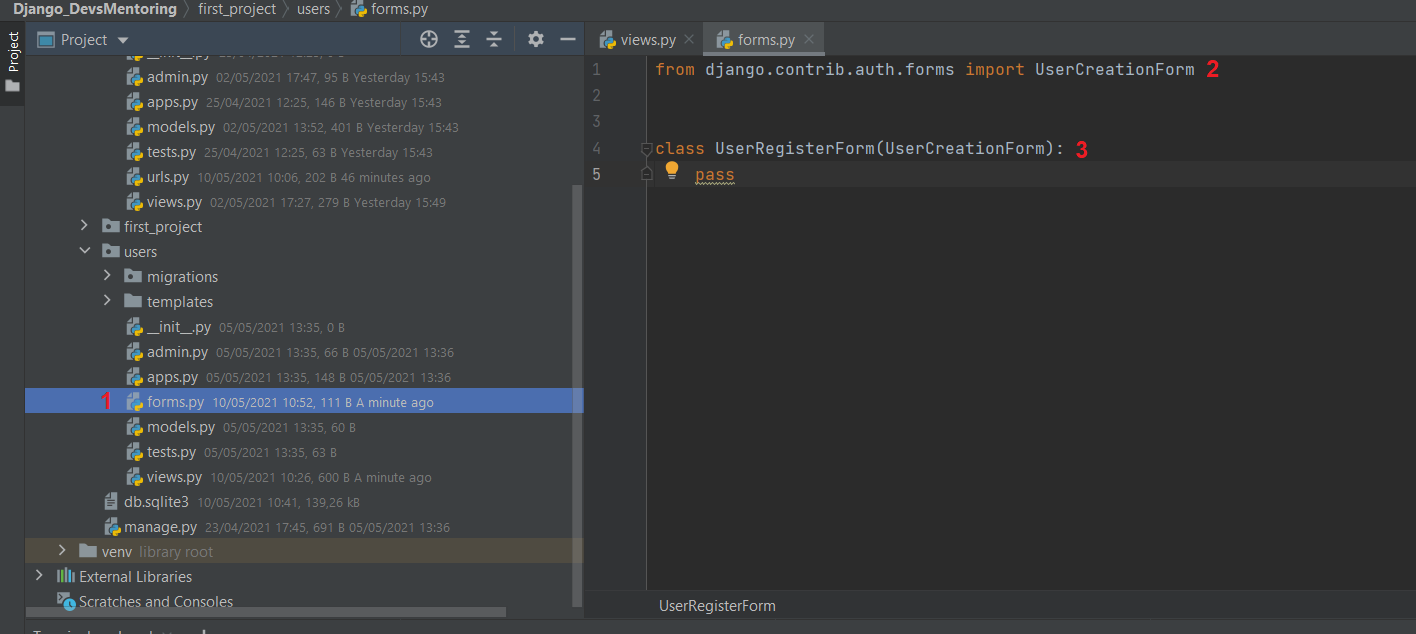
*Po przejściu do tabeli User*

Udało się! Zgodnie z tym, co widzisz na powyższym zrzucie ekranu, dodaliśmy nowego użytkownika - Test2 - do tabeli User w bazie danych. Pojawia się jednak pewien szkopuł… Zauważ, że przy tworzeniu usera, formularz UserCreationForm nie zapewnia możliwości podania adresu e-mail. Jedyne pola, jakie możemy uzupełnić to: Username, Password oraz Password Confirmation. Wynika to z tego, iż domyślnie klasa UserCreationForm standardowo dostarcza właśnie te 3 pola. Musimy zatem rozważyć, jak go zmodyfikować, aby dodać zupełnie nowe pole i móc tworzyć nowych użytkowników z pełnym zestawem informacji.

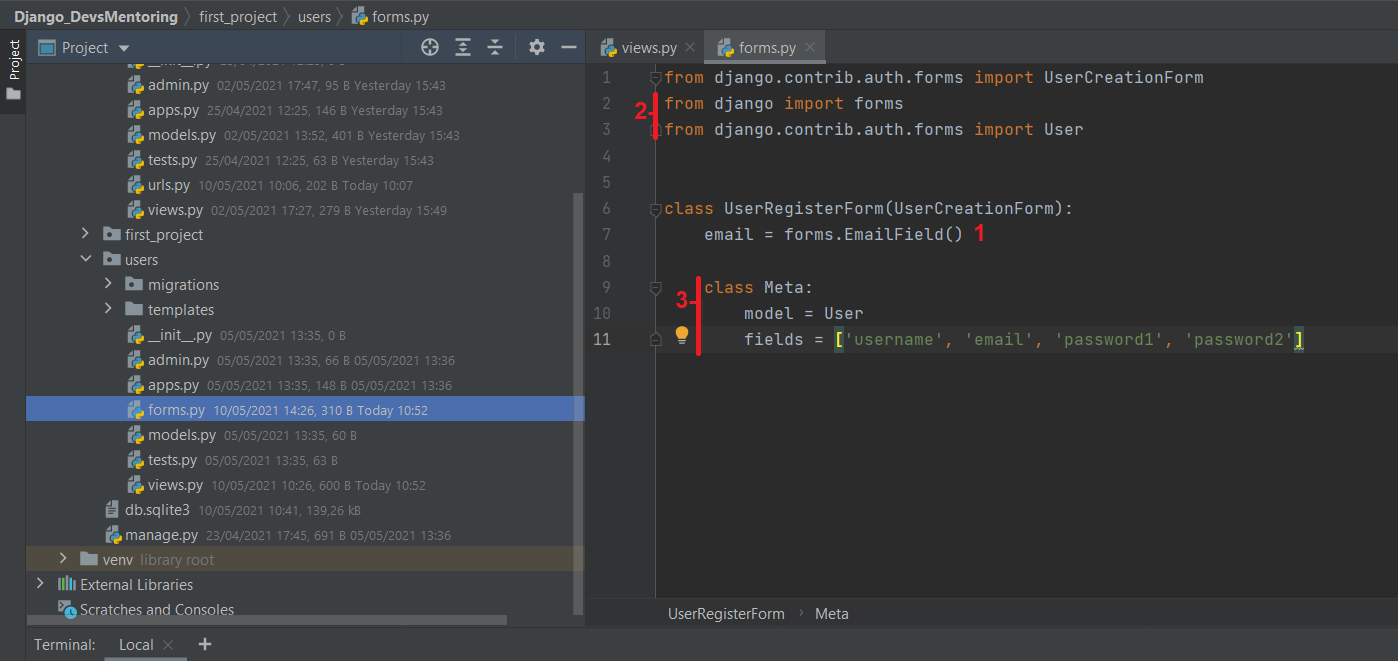
Rozwiązanie tego problemu jest dość proste - otóż musimy stworzyć zupełnie nową klasę, która reprezentować będzie Nasz personalizowany formularz. Zasada tworzenia wygląda następująco:

* Musimy w katalogu aplikacji users utworzyć nowy plik o nazwie forms.py
* Następnie powinniśmy zbudować nową klasę, reprezentującą formularz, o nazwie, np. UserRegisterForm. Klasa ta dziedziczyć będzie po formularzu bazowym - UserCreationForm.
* Taką klasę będziemy musieli uzupełnić właściwymi polami, jak np. zagnieżdżoną klasą Meta (o której więcej za chwilę), a także pojedynczymi polam reprezentującymi kolejne składniki formularza (w tym pole do podania adresu email).

1. Utwórzmy zatem nowy plik users/forms.py (1) i dodajmy do niego pustą klasę UserCreationForm (3). Pamiętaj również o odpowiednich importach (2).



1. Mając już taką sygnaturę pliku, przejdźmy do implementacji klasy UserRegisterForm. Umieścimy w niej pola takie jak: email (1) - reprezentujące nowo dodaną rubrykę do formularza oraz klasę Meta, w której zawrzemy: tabelę - User - do której będziemy chcieli zapisywać przechwytywane dane (3) oraz listę fields, która określa, które pola mają zostać wyświetlone w ramach budowanego formularza. Pamiętaj o odpowiednich importach (2).



from django.contrib.auth.forms import UserCreationForm

from django import forms

from django.contrib.auth.forms import User

class = UserRegisterForm(UserCreationForm):

email = forms.EmailField()

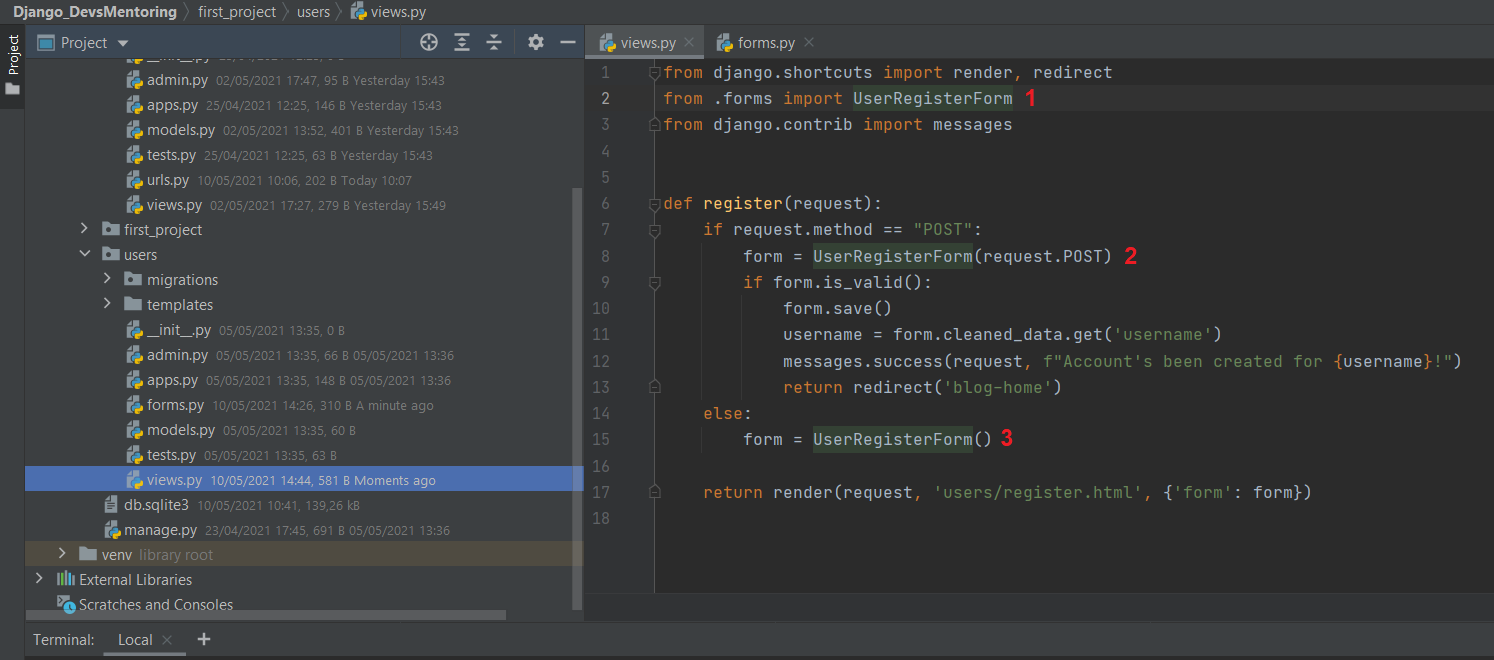
class Meta:

model = User

fields = ['username', 'email', 'password1', 'password2']

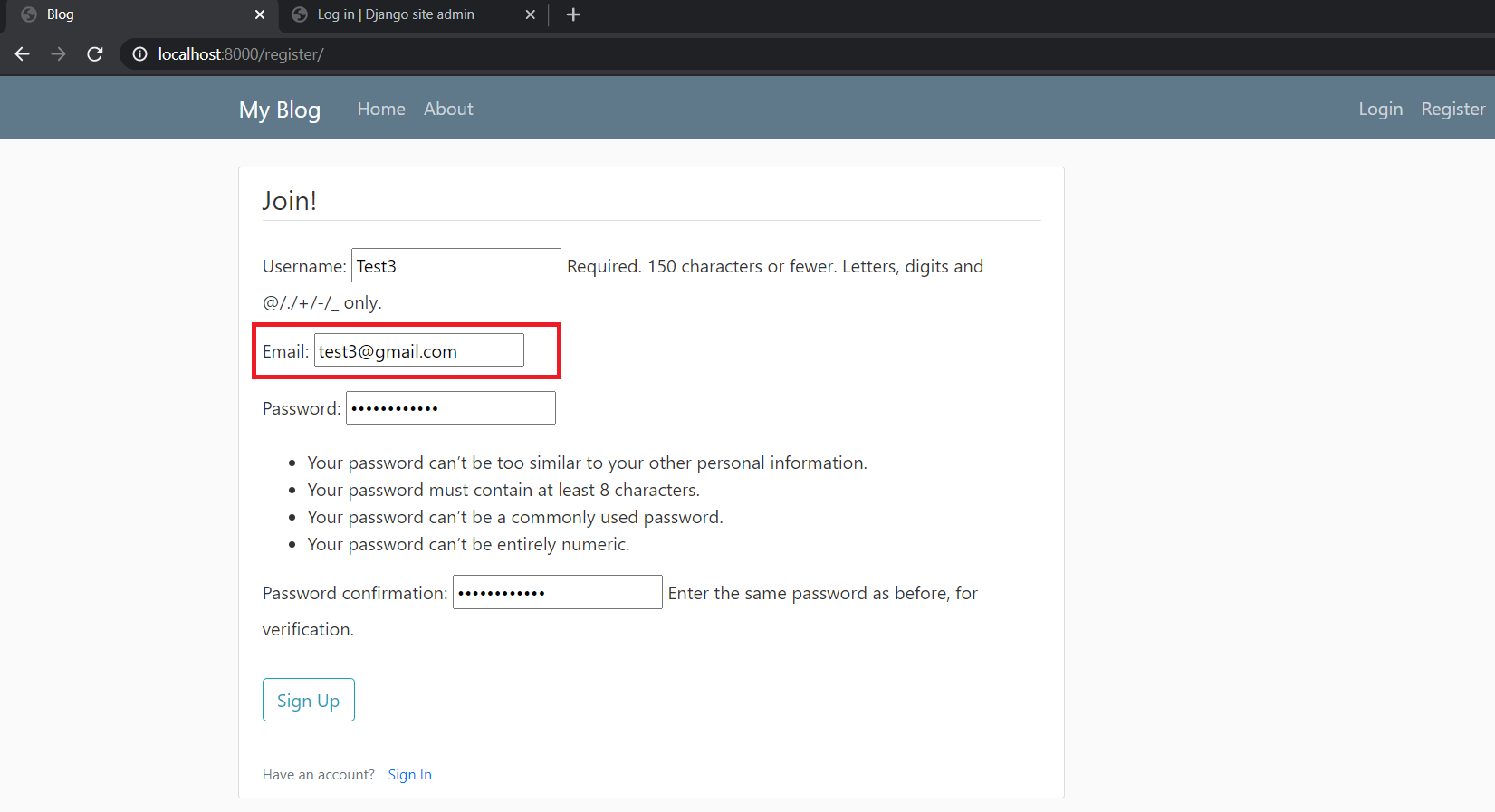
Na pewno w tym momencie zastanawia Cię zagnieżdżona klasa Meta, ponieważ wprowadza dość niestandardowy zapis do Naszego kodu. Musisz się jednak do niego przyzwyczaić, gdyż często będziemy wykorzystywali taką strukturę przy tworzeniu własnych Django-wych struktur. Meta to klasa - kontener, który zawiera pewne opcje (metadane) definiujące Model, w którym się znajduje. Może ona określać choćby takie właściwości jak: zakres uprawnień modelu, powiązaną tabelę, do której mają być automatycznie zapisywane dane, pola wchodzące w skład formularza etc.

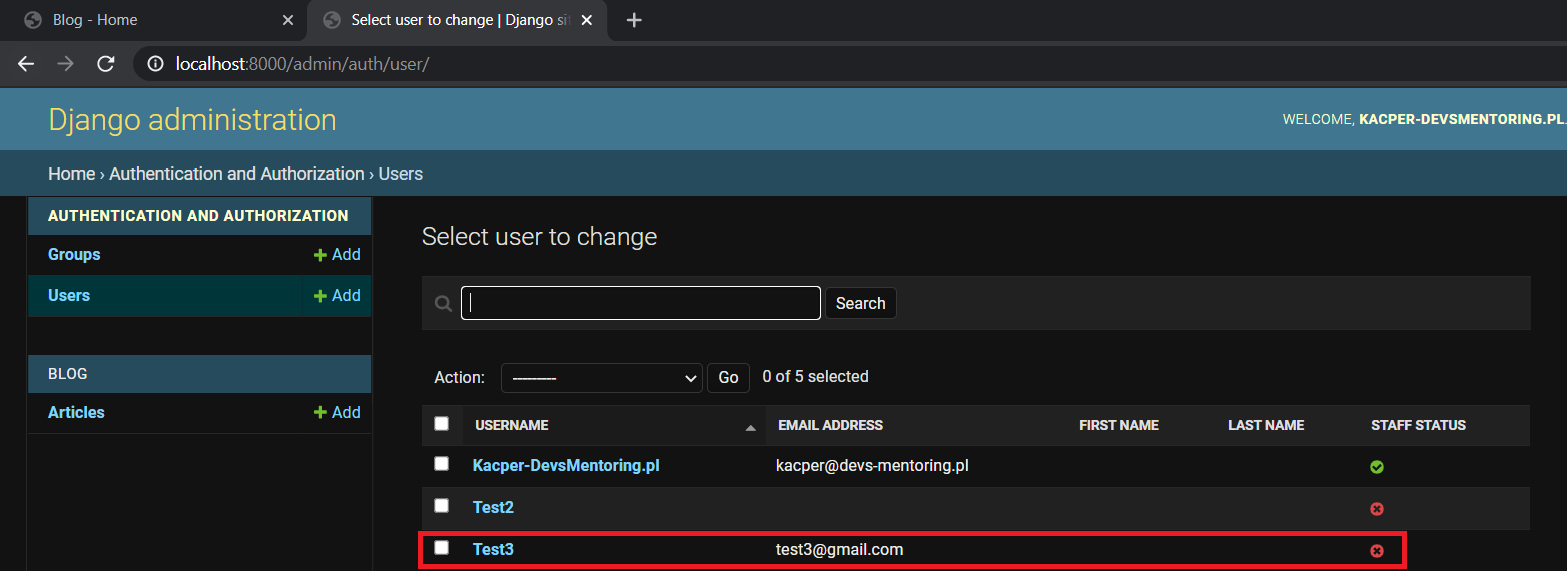
Po tym, jak wprowadziliśmy powyższe zmiany do pliku users/forms.py, nadszedł czas na podmianę obecnych obiektów UserCreationForm w pliku users/views.py na obiekty UserRegisterForm.



Na powyższym zrzucie widać, jak linię importującą obiektu UserCreationForm zastąpiłem importem utworzonej uprzednio klasy UserRegisterForm (1), a także jak dodałem tworzenie jej dwóch obiekt (2 i 3).

Od tego momentu, po przejściu na podstronę /register, ujrzysz dodatkowe pole email przy zachowaniu tej samej funkcjonalności - przenoszenia wpisanych danych do tabeli w bazie danych.

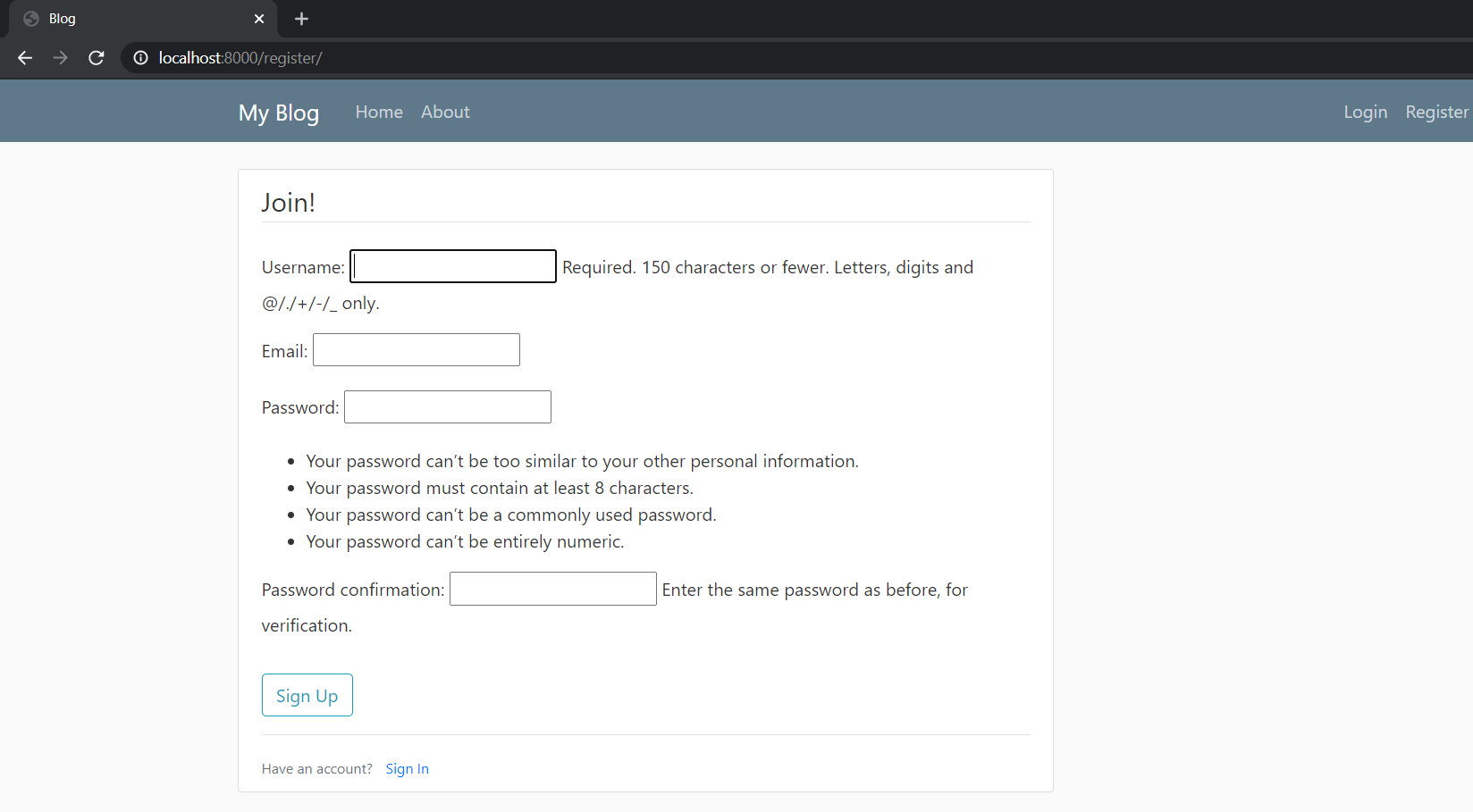




*Widoczny adres email [test3@gmail.com](mailto:test3@gmail.com) przy rekordzie Test3*

**Na zakończenie…**

Ostatnim krokiem będzie dopracowanie wyglądu obecnego formularza. Przyjrzyjmy się mu raz jeszcze.



Przyznaj, że nie wygląda on idealnie. Sporo do życzenia pozostawiają choćby punkty takie jak: *Your password can’t be too similar to your other personal information*, wygląd rubryk do wpisywania danych i ogólne formatowanie czcionki.

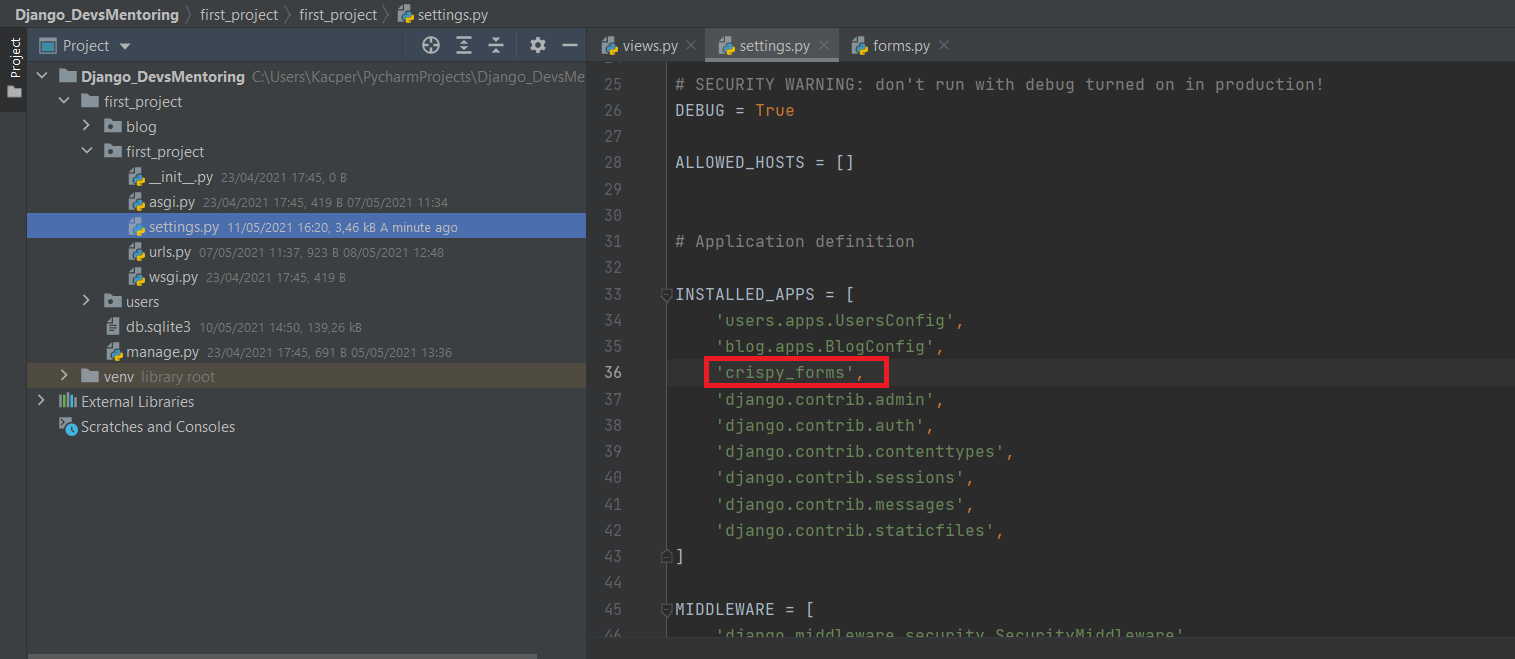
Wygląd ten możemy poprawić, modyfikując kod Bootstrapa lub w o wiele łatwiejszy i sprytny sposób - przez wykorzystanie, tzw. crispy forms. To biblioteka, która umożliwia łatwe zarządzanie formularzami Django. Pozwala dostosowywać ich właściwości (takie jak wygląd rubryk, przycisków, ogólnie CSS-a) bez wprowadzania modyfikacji do stworzonego szablonu HTML. Brzmi obiecująco! Przystąpmy zatem do instalacji.

Bibliotekę tę zainstalujemy przez wykorzystanie polecenia pip install django-crispy-forms.

Jeszcze crispy-bootstrap5



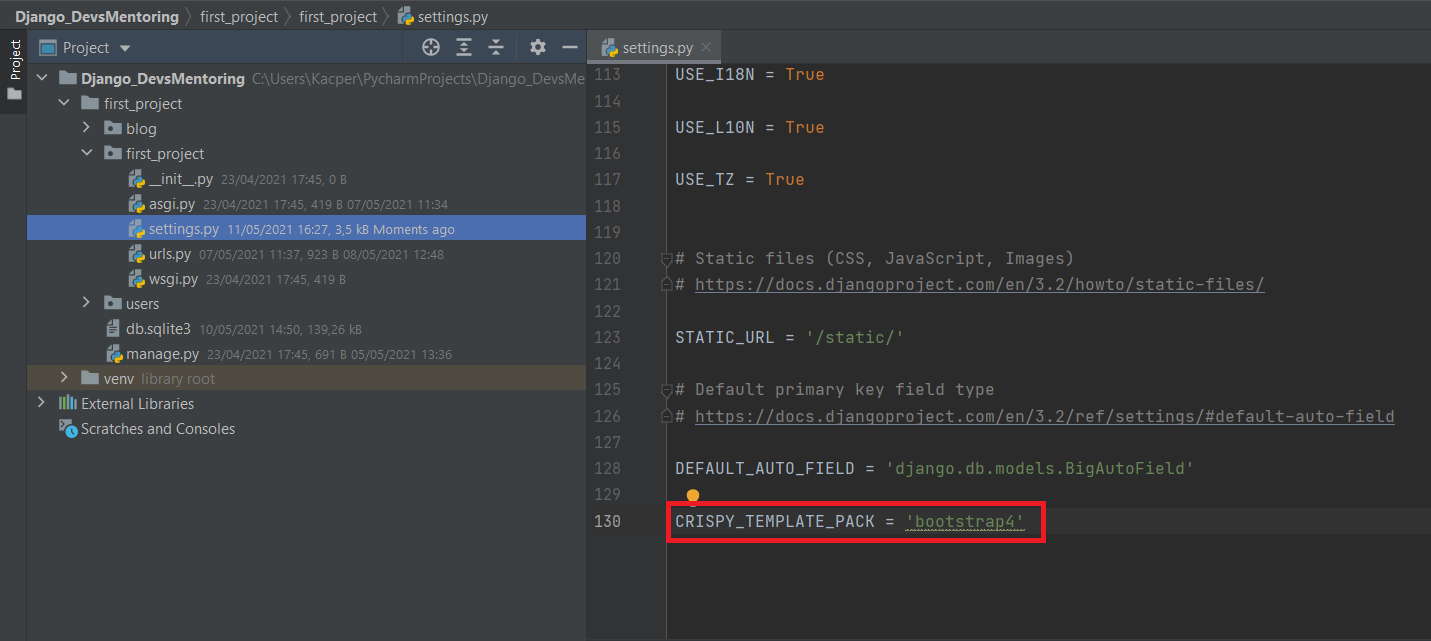
Następnie, musimy dodać tak zainstalowaną aplikację do znanego już Nam pliku first\_project/settings.py. To wszystko po to, aby prawidłowo zintegrować tworzoną aplikację Django z zewnętrznymi rozszerzeniami.



"crispy\_forms",

"crispy\_bootstrap5",

Następnie musimy doprecyzować, którą wersję silnika Bootstrap, crispy będzie wykorzystywało. Domyślnie wykorzystuje ona wersję Bootstrap, ale chcielibyśmy ją zmodyfikować do wersji najnowszej, czyli Bootstrap. Aby to osiągnąć, wystarczy na końcu pliku first\_project/settings.py umieścić poniższy zapis: CRISPY\_TEMPLATE\_PACK = ‘boostrap4’

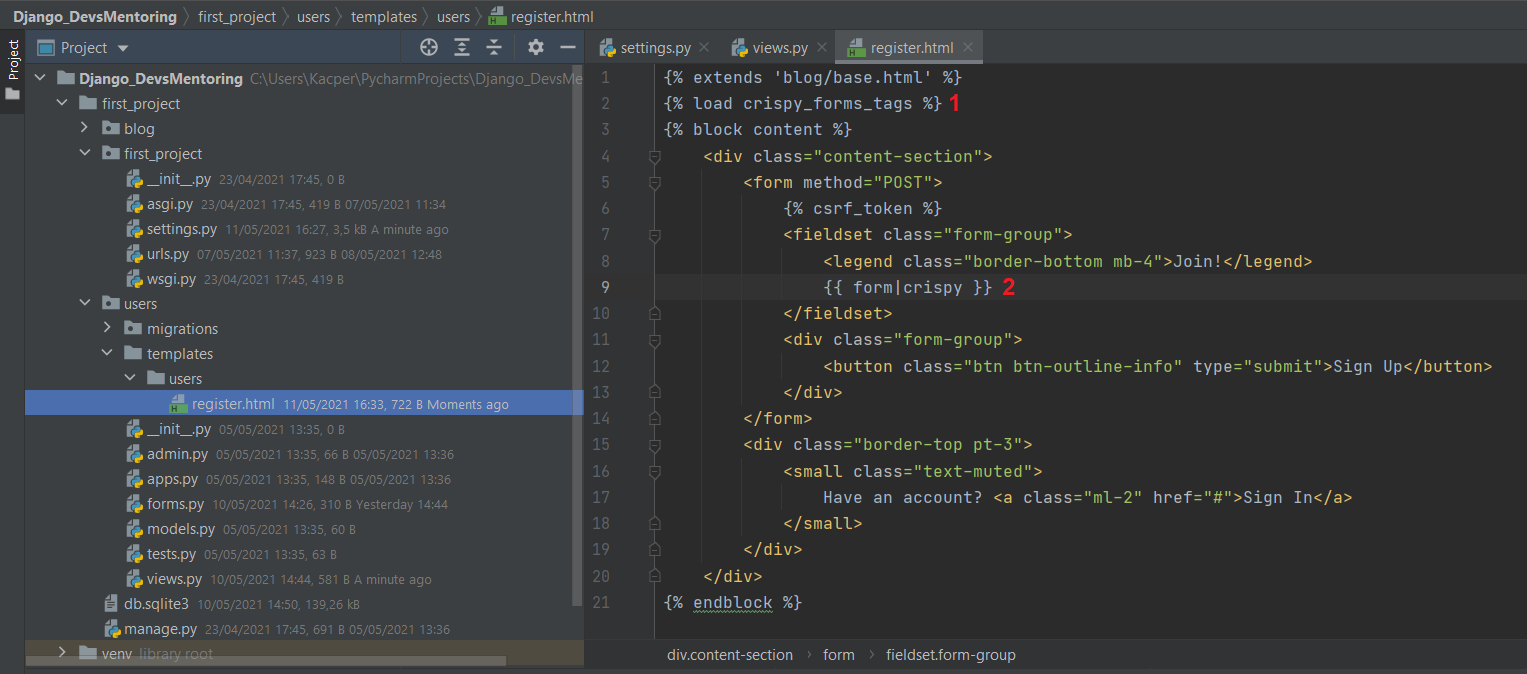


CRISPY\_ALLOWED\_TEMPLATE\_PACKS = "bootstrap5"

CRISPY\_TEMPLATE\_PACK = "bootstrap5"

**Dodanie formatowania z crispy**

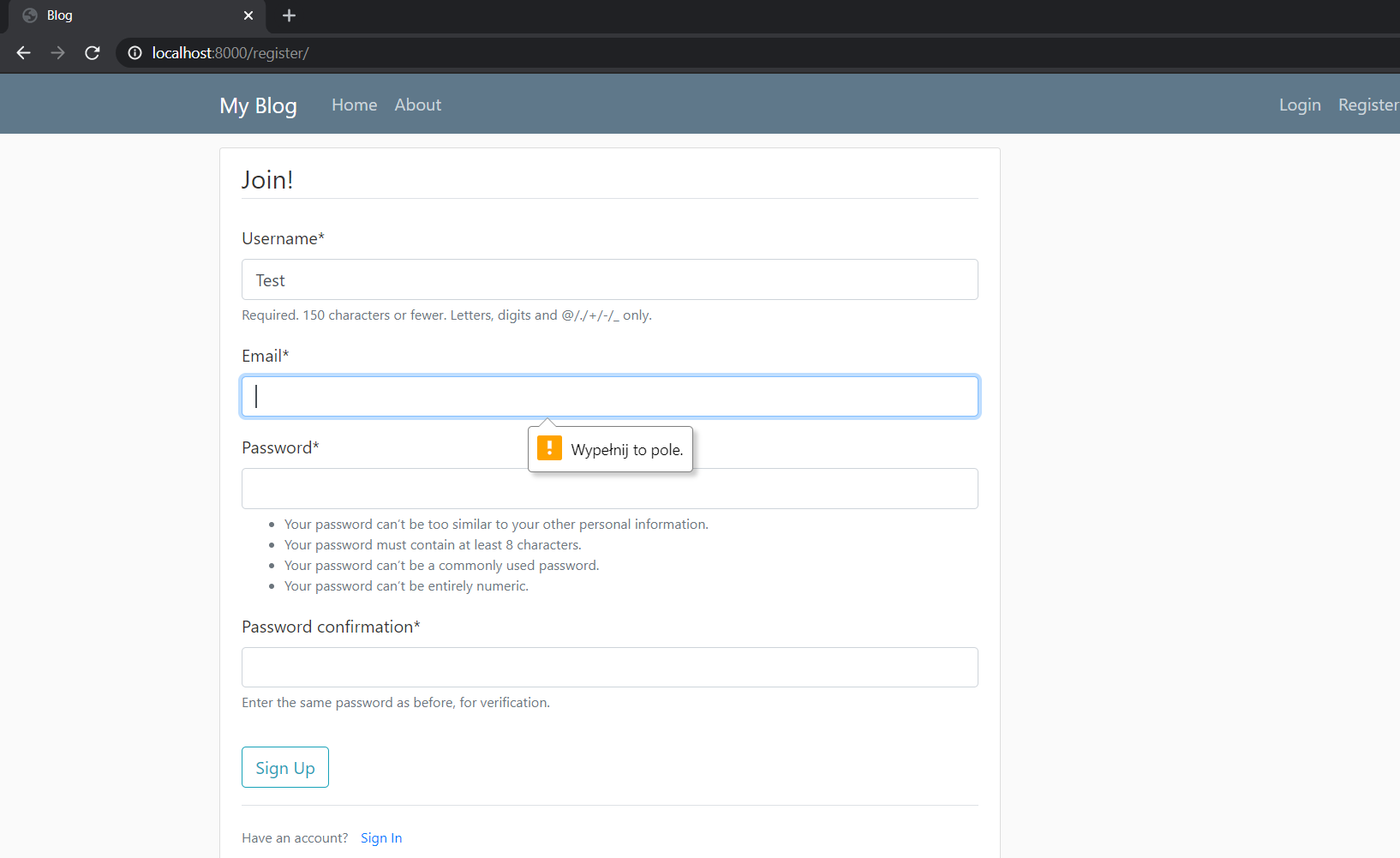
Ostatnim krokiem dzielącym Nas od wykorzystania crispy w praktyce jest dodanie właściwego już formatowania Naszego formularza w pliku users/templates/users/register.html. Poniżej ukazałem wszystkie (aż 2 :)) przeprowadzone zmiany.



{% load crispy\_forms\_tags %}

Pierwsza zmiana (1) dotyczyła dodania możliwości ładowania paczki crispy do pliku .html. (2) natomiast wprowadza już samo formatowanie zgodnie z dostarczonym szablonem przez crispy do Naszego formularza.

Czas na uruchomienie tak zmienionej aplikacji i sprawdzenia, czy wszystko działa zgodnie z oczekiwaniami.



Voilà! Spójrz, jak piękny stał się Nasz formularz rejestracji. Automatycznie dodana jest w nim także walidacja poprawności wpisywanych pól. Uważam, że epizod związany z tworzeniem własnego panelu rejestracji możemy zamknąć. Przed Nami dodanie kolejnych funkcjonalności - jak, np. umożliwienie procesu logowania na stronie. Ale to w następnych szkoleniach...