**Django - szkolenie 7**

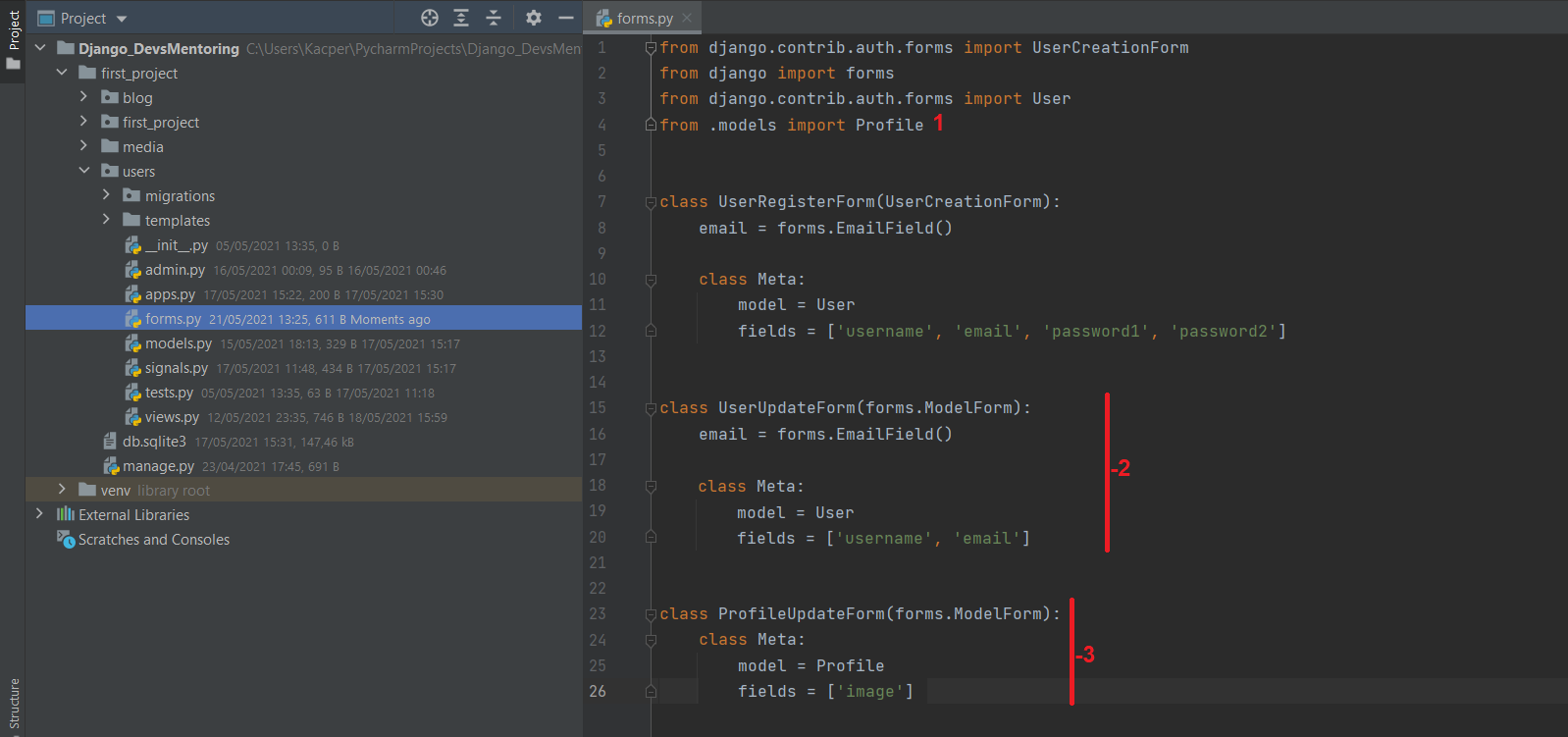
**Zarządzanie profilem**

**Zarządzanie profilem**

Po ostatnim szkoleniu, w Naszej aplikacji znalazł się poprawnie funkcjonujący profil każdego z użytkowników. Mogą oni w nim podejrzeć swoją nazwę oraz zuploadowane zdjęcie profilowe. Brakuje jednak w Naszej aplikacji możliwości zmieniania danych z poziomu strony profilowej. Przypomnę Ci tylko, że do tej pory, aby wprowadzić jakiekolwiek modyfikacje związane z użytkownikiem, konieczne było przejście do panelu administracyjnego i to właśnie z poziomu niego dokonanie zmian. Pomijając już fakt, że nie chcielibyśmy, aby każdy użytkownik miał dostęp do takiego panelu, to przyznaj, że jest to niezbyt optymalne rozwiązanie.

**Utworzenie formularzy zmiany**

Chcąc zatem umożliwić użytkownikowi wprowadzanie zmian z poziomu jego profilu dostępnego pod endpointem /profile, musimy dodać w sumie 2 nowe klasy: UserUpdateForm oraz ProfileUpdateForm. Przypomnę Ci tylko, że taki rodzaj formularza tworzyliśmy już przy zapewnianiu użytkownikowi panelu rejestracji. Wówczas utworzyliśmy nowy plik users/forms.py i umieściliśmy w nim klasę UserRegisterForm. Aby wdrożyć wspomniane formularze zmiany, przejdźmy do tego samego pliku i utwórzmy kolejne struktury.



from django.contrib.auth.forms import UserCreationForm

from django import forms

from django.contrib.auth.forms import User

from .models import Profile

class UserRegisterForm(UserCreationForm):

email = forms.EmailField()

class Meta:

model = User

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

class UserUpdateForm(forms.ModelForm):

email = forms.EmailField()

class Meta:

model = User

fields = ['username', 'email']

class ProfileUpdateForm(forms.ModelForm):

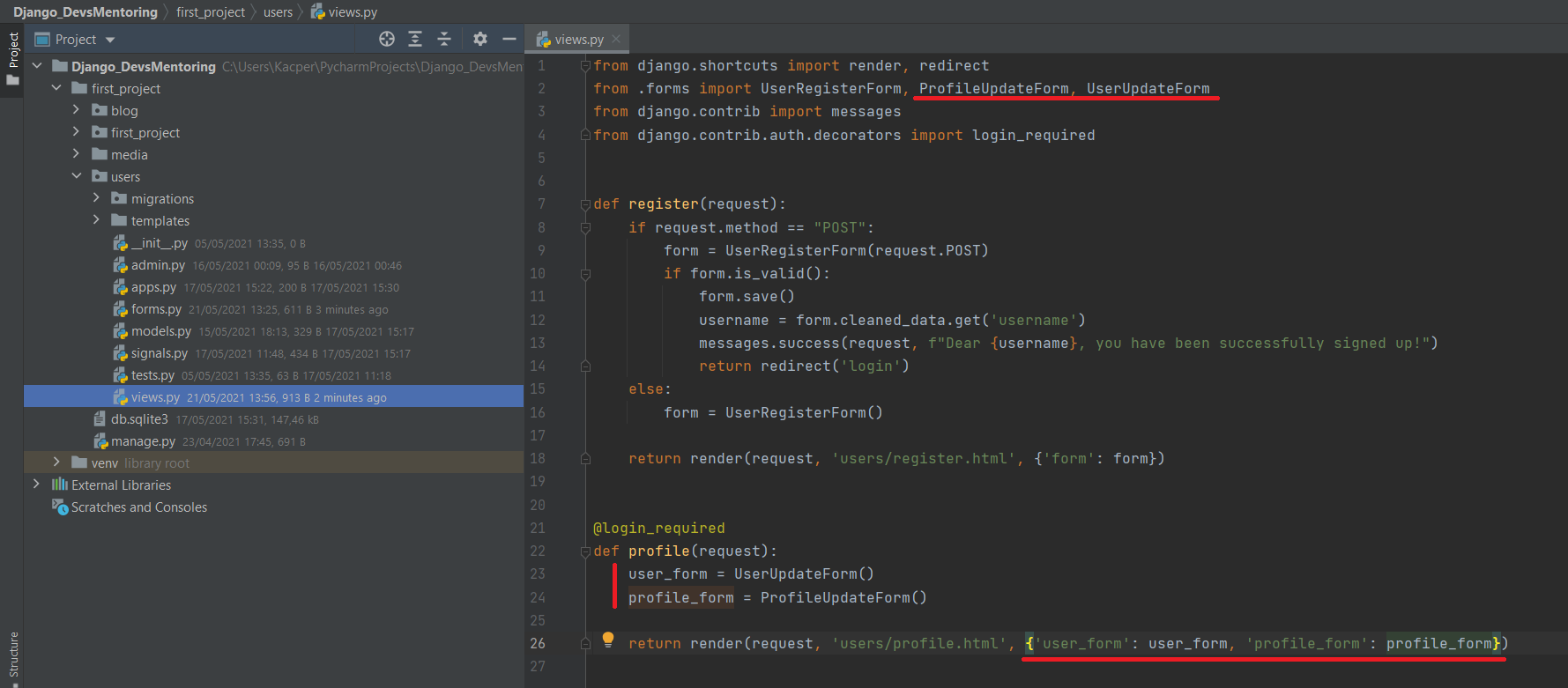
class Meta:

model = Profile

fields = ['image']

Powyższy screenshot prezentuje dokonane zmiany w pliku forms.py. Jak widzisz, konieczne było przede wszystkim utworzenie nowych formularzy z odpowiednimi polami do modyfikacji - username oraz email w przypadku UserUpdateForm oraz image w przypadku ProfileUpdateForm. Gdybyś zastanawiał się, dlaczego musieliśmy utworzyć dwie klasy, a nie możliwe było zawarcie wszystkiego w jednej, to wynika to z tego, iż Profile reprezentowany jest przez oddzielną tabelę Profile (co zmusza Nas również do zaimportowania odpowiedniej klasy, tak jak jest to widoczne przy punkcie 1).

Po utworzeniu klas, przejdźmy już do pliku users/views.py, aby móc utworzyć i przekazać z jego poziomu obiekty UserUpdateForm oraz ProfileUpdateForm do szablonu profilu.



from django.shortcuts import render, redirect

from .forms import UserRegisterForm, ProfileUpdateForm, UserUpdateForm

from django.contrib import messages

from django.contrib.auth import logout

from django.urls import reverse

from django.contrib.auth.decorators import login\_required

def register(request):

if request.method == 'POST':

form = UserRegisterForm(request.POST)

if form.is\_valid():

form.save()

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

messages.success(request, f"Dear {username}, you have been succesfully signed up!!")

return redirect('login')

else:

form = UserRegisterForm()

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

# Create your views here.

def logout\_view(request):

logout(request)

messages.info(request, "Logged out successfully!")

return render(request, 'users/logout.html')

@login\_required

def profile(request):

user\_form = UserUpdateForm()

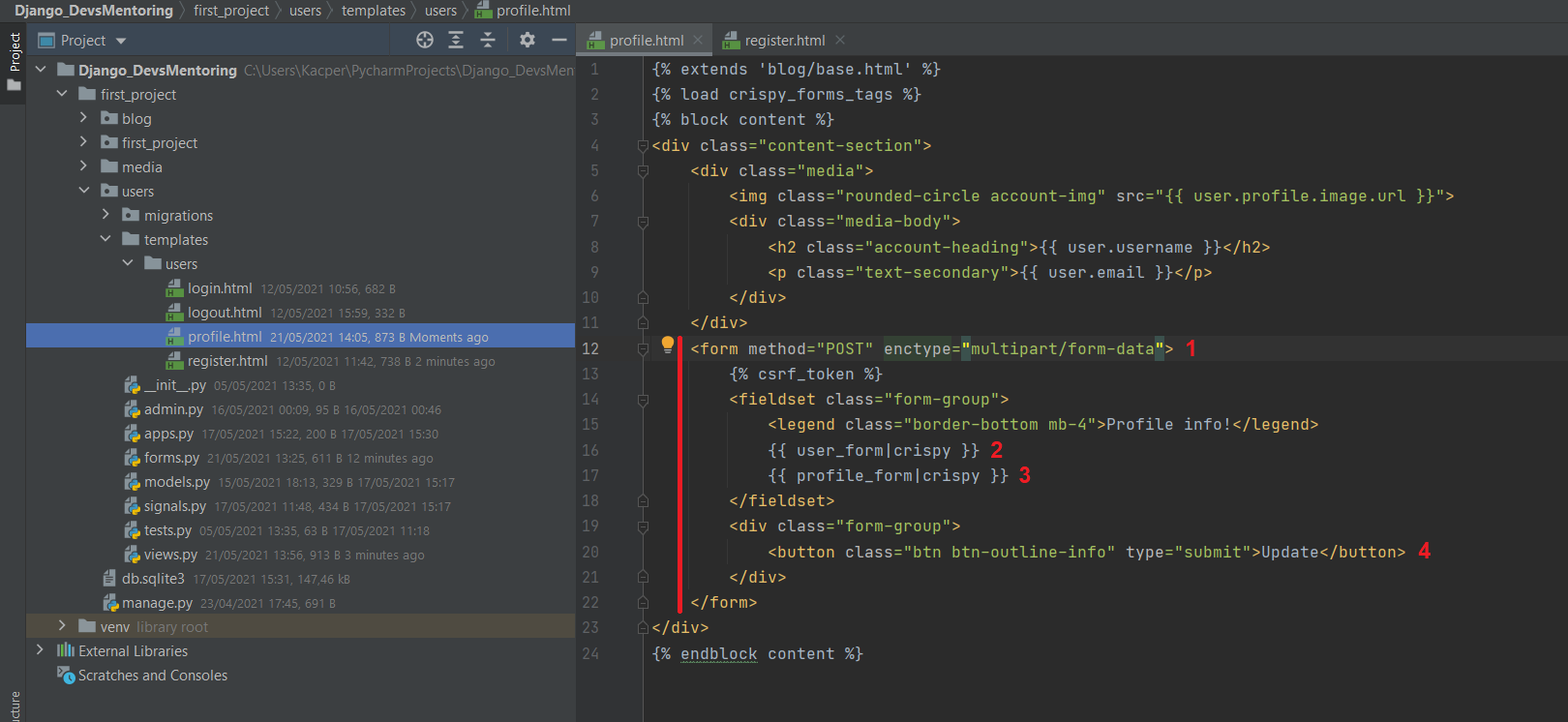
profile\_form = ProfileUpdateForm()

return render(request, 'users/profile.html', {'user\_form': user\_form, 'profile\_form': profile\_form})

Czerwonym kolorem zaznaczyłem zmiany wprowadzone do pliku views.py. Myślę, że nie wymagają one mocno rozbudowanego komentarza - w skrócie, konieczne było utworzenie obiektów formularzy user\_form oraz profile\_form oraz przekazanie ich do renderowanego szablonu users/profile.html, aby mogły zostać prawidłowo wyświetlone.

Teraz nadszedł czas na zakodowanie, jak wizualnie ma wyglądać edycja profilu.

Przejdźmy do pliku users/templates/users/profile.html i wprowadźmy następujące zmiany.



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

{% load crispy\_forms\_tags %}

{% block content %}

<div class="content-section">

<div class="media">

<img class="rounded-circle account-img" src="{{ user.profile.image.url }}">

<div class="media-body">

<h2 class="accound-heading">{{ user.username }}</h2>

<p class="text-secondary">{{ user.email }}</p>

</div>

</div>

<form method="POST" enctype="multipart/form-data">

{% csrf\_token %}

<fieldset class="form-group">

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

{{ user\_form|crispy }}

{{ profile\_form|crispy }}

</fieldset>

<div class="form-group">

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

</div>

</form>

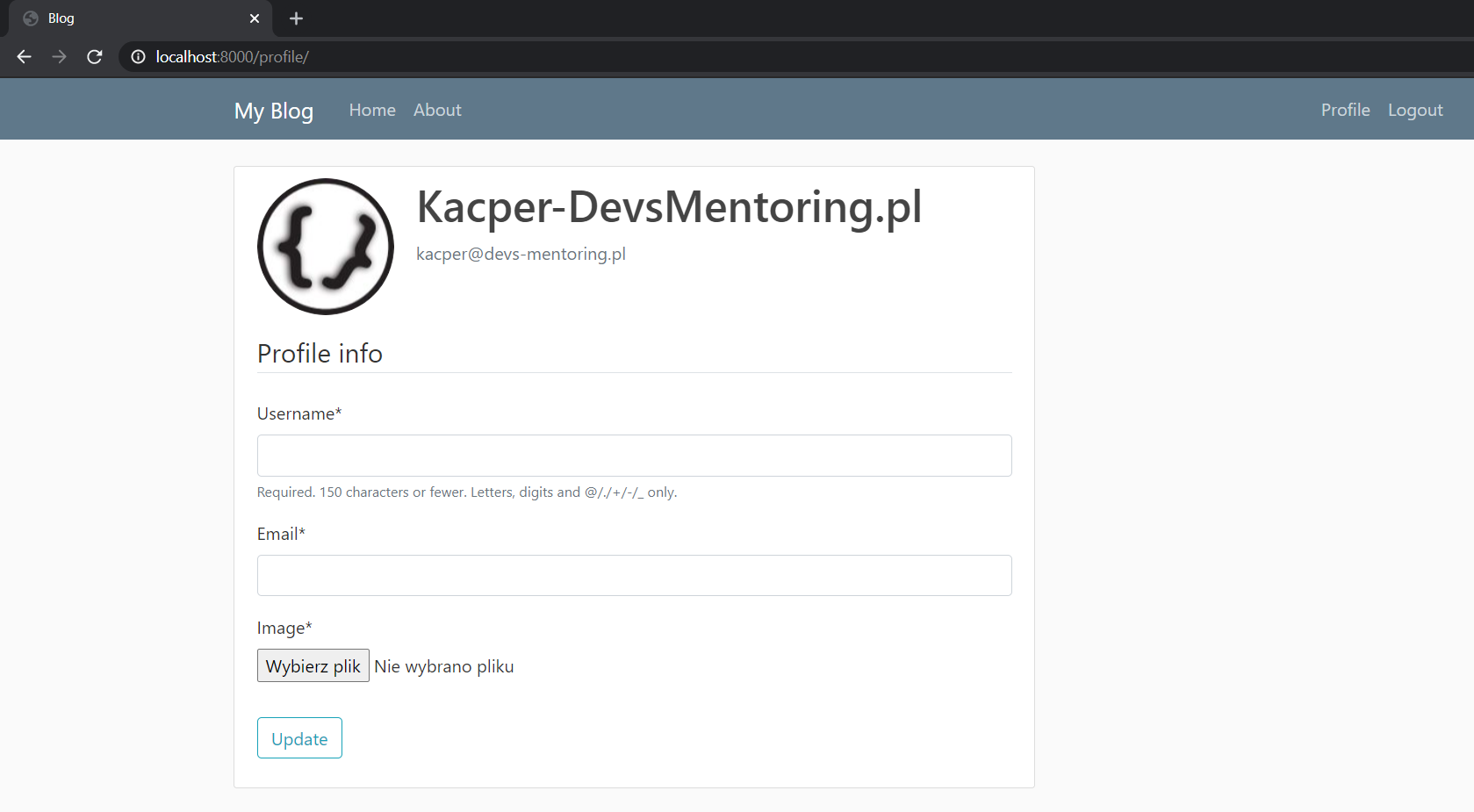
</div>

{% endblock %}

Czerwoną linią zaznaczyłem cały obszar kodu, który tak naprawdę skopiowałem z sąsiedniego pliku register.html (w którym znajduje się już gotowy formularz rejestracji). Jedynie do takiego snippetu dodałem pewne modyfikacje, które oznaczyłem cyframi 1-4.

* (1) - czyli dodanie atrybutu enctype z wartością “multipart/form-data”; to w zasadzie najważniejsza i najbardziej kluczowa zmiana z przeprowadzonych; odnosi się ona do możliwości update-owania zdjęcia profilowego; gdybyśmy zapomnieli w ten sposób dodać parametr enctype do budowanego szablonu form, to wystąpiłoby duże ryzyko nieprawidłowego zapisu na serwerze nowo uploadowanych awatarów użytkownika
* (2) i (3) - to bezpośrednie już umieszczenie formularzy do zmiany informacji o użytkowniku
* (4) - to bardzo subtelna zmiana przekopiowanego snippetu kodu, bo jedynie dostosowanie, aby wyświetlany był napis Update zamiast Sign Up

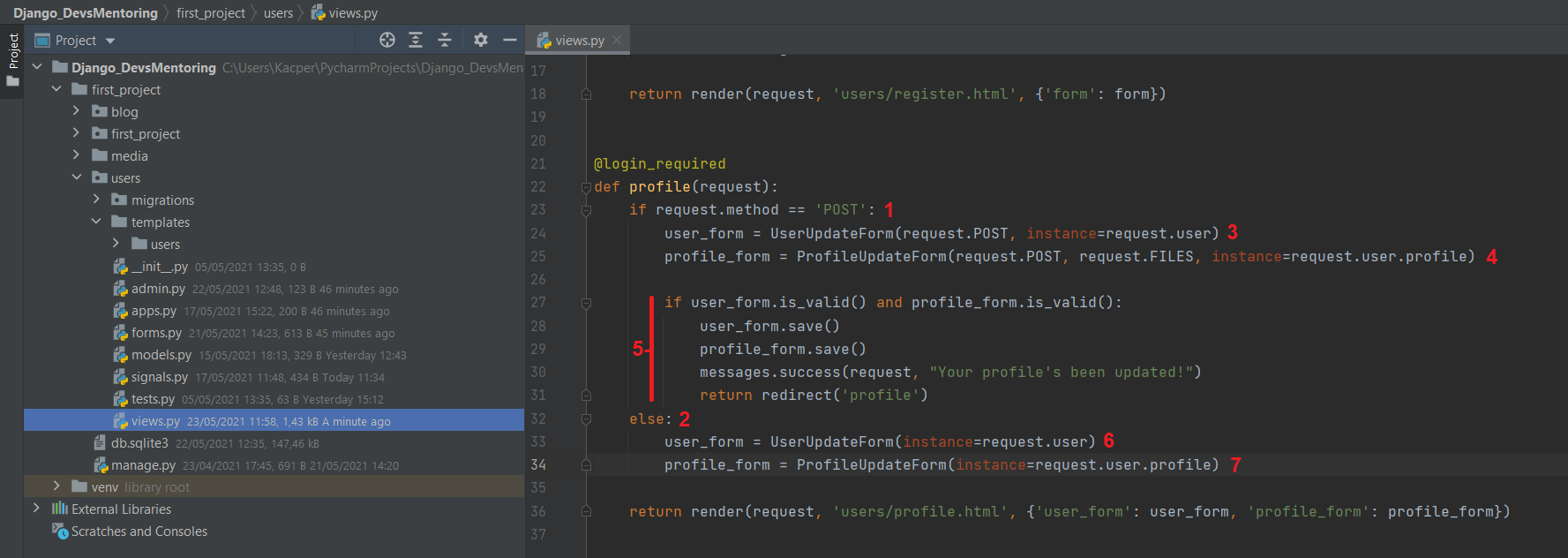
Sprawdźmy, jaki efekt przyniosły wprowadzone wyżej zmiany. Zalogujmy się do Naszej aplikacji i przejdźmy do zakładki /profile. Naszym oczom powinien ukazać się poniższy widok:



Brakuje jednak tutaj dwóch funkcjonalności. Przede wszystkim - możliwości walidacji (pod kątem poprawności) wprowadzonych zmian i update-owania proflu użytkownika po wciśnięciu przycisku Update. Ponadto chcielibyśmy, aby pola Username i Email były domyślnie wypełnione odpowiednimi danymi użytkownika (aby przy przejściu do zakładki profilu, rubryki były automatycznie uzupełnione odpowiednią nazwą użytkownika i adresem email).

**Dalsza modyfikacja widoku profile**

W celu zakodowania wyżej przedstawionych funkcjonalności, przejdziemy do pliku users/views.py i dokonamy modyfikacji.



from django.shortcuts import render, redirect

from .forms import UserRegisterForm, ProfileUpdateForm, UserUpdateForm

from django.contrib import messages

from django.contrib.auth import logout

from django.urls import reverse

from django.contrib.auth.decorators import login\_required

def register(request):

if request.method == 'POST':

form = UserRegisterForm(request.POST)

if form.is\_valid():

form.save()

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

messages.success(request, f"Dear {username}, you have been succesfully signed up!!")

return redirect('login')

else:

form = UserRegisterForm()

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

# Create your views here.

def logout\_view(request):

logout(request)

messages.info(request, "Logged out successfully!")

return render(request, 'users/logout.html')

@login\_required

def profile(request):

if request.method == 'POST':

user\_form = UserUpdateForm(request.POST, instance=request.user)

profile\_form = ProfileUpdateForm(request.POST, request.FILES, instance=request.user.profile)

if user\_form.is\_valid() and profile\_form.is\_valid():

user\_form.save()

profile\_form.save()

messages.success(request, "Your profile has been updated!")

return redirect('profile')

else:

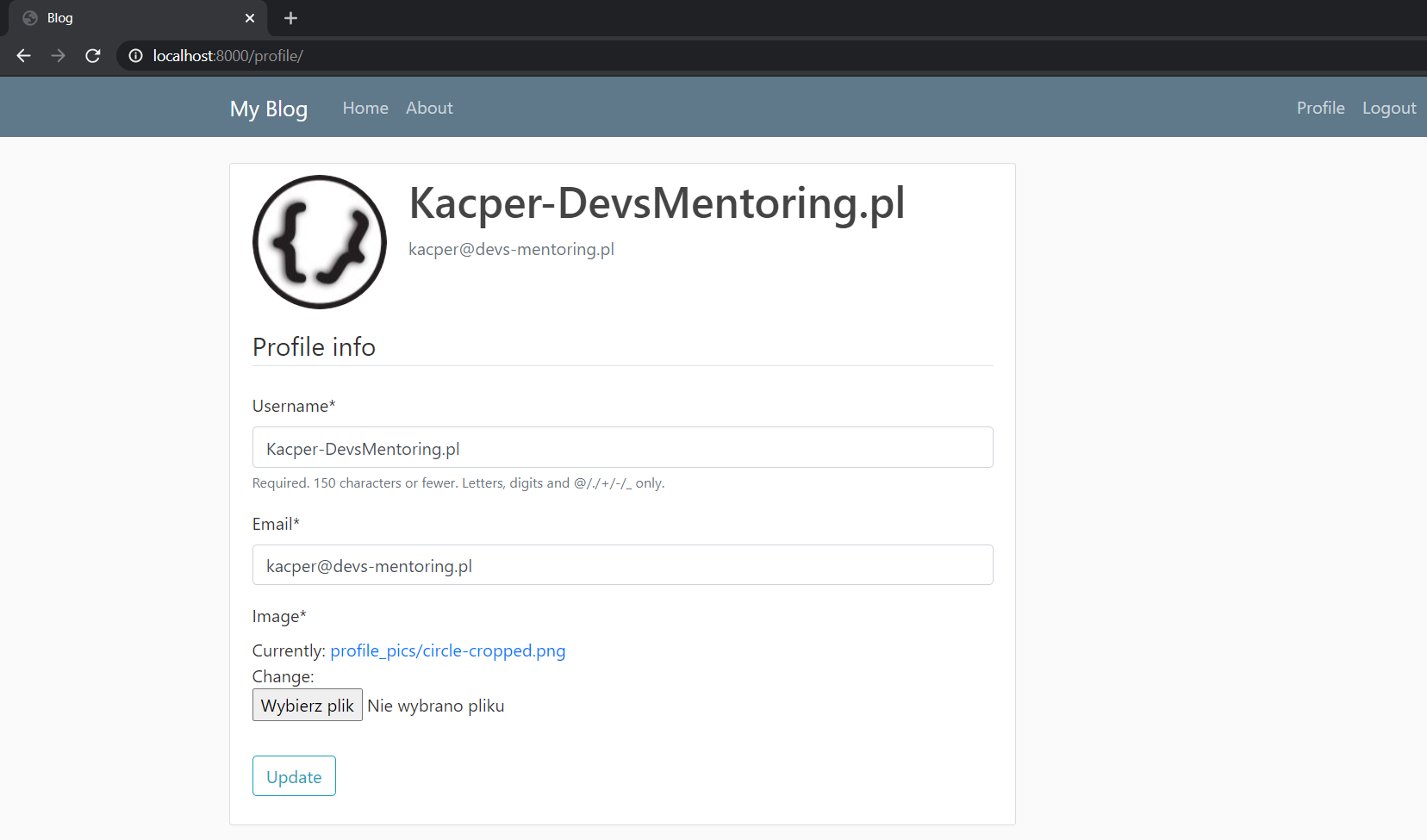
user\_form = UserUpdateForm(instance=request.user)

profile\_form = ProfileUpdateForm(instance=request.user.profile)

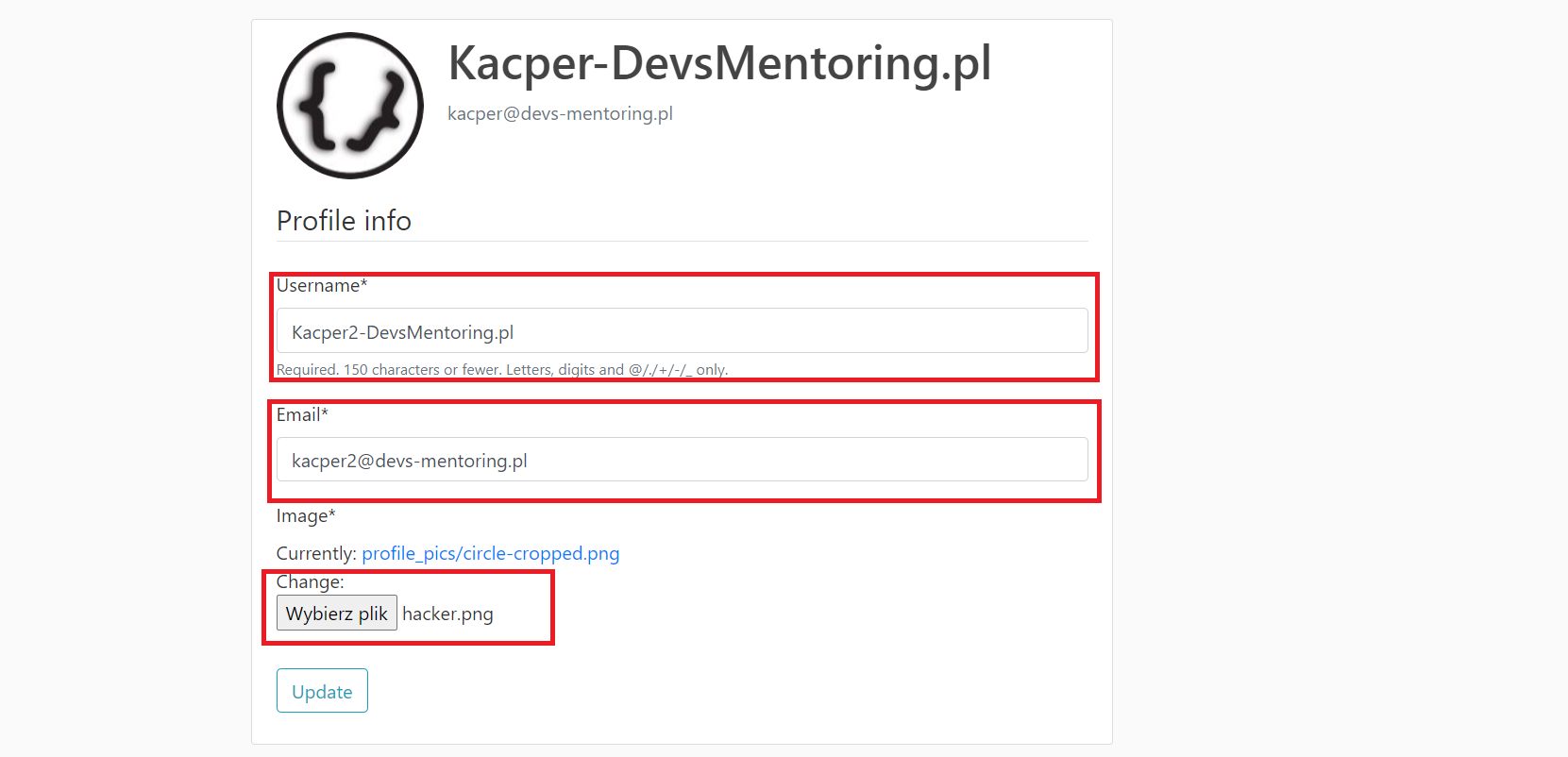
return render(request, 'users/profile.html', {'user\_form': user\_form, 'profile\_form': profile\_form})

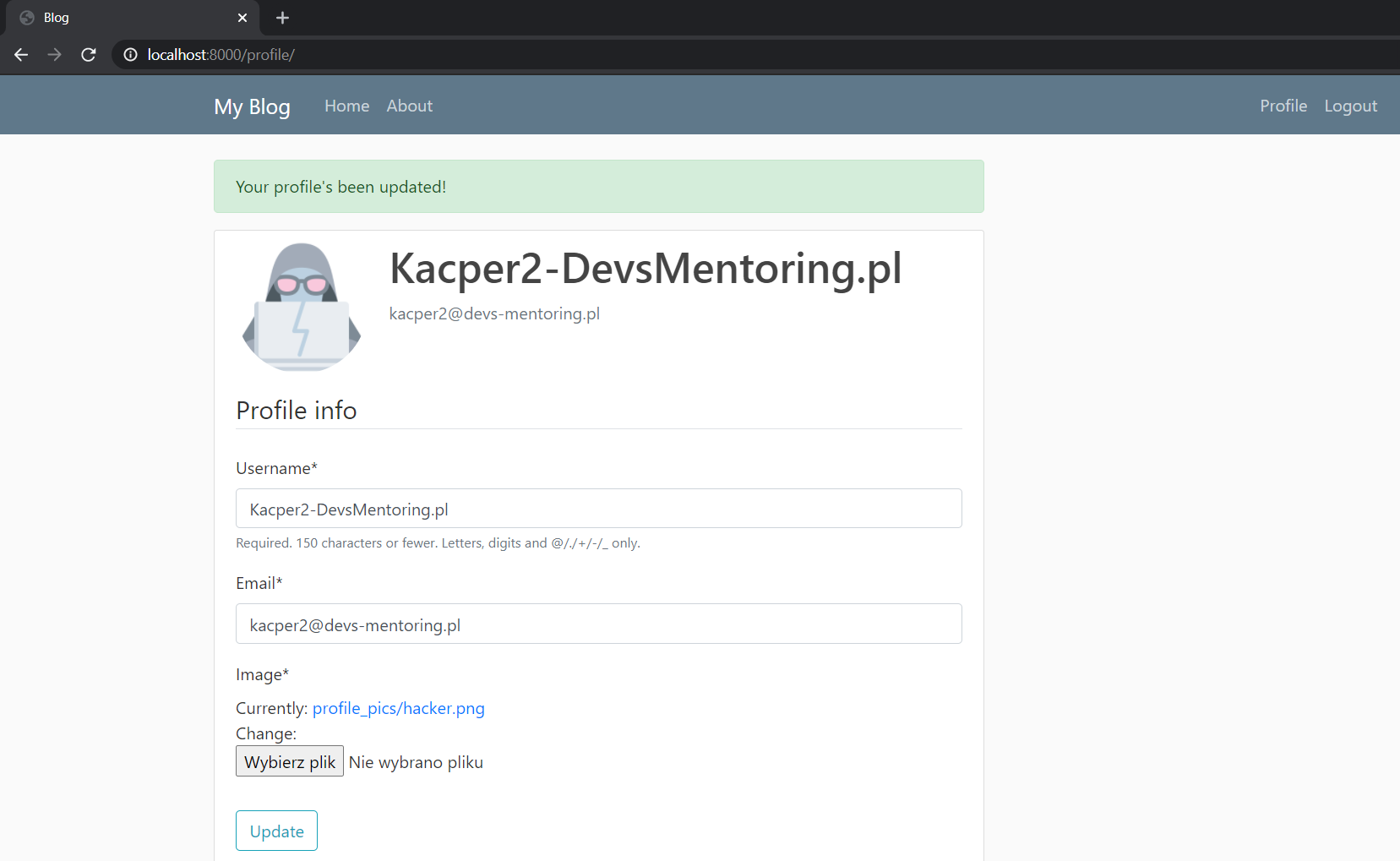
* (1) i (2) to linie kodu umożliwiające rozróżnienie, czy w danym momencie użytkownik wcisnął przycisk Update i musimy obsłużyć wysłany formularz danych (if request.method == ‘POST’), czy go po prostu go wygenerować (else)
* po tym, jak użytkownik wciśnie przycisk Update i zostanie obsłużone zapytanie POST, wykonają się punkty (3) i (4); polegają one na utworzeniu obiektów UserUpdateForm i ProfileUpdateForm z odpowiednimi argumentami: request.POST - poinformowanie, iż właśnie wysłaliśmy na serwer zapytanie typu POST, request.FILES - określa, skąd mamy czerpać informacje o awatarze użytkownika, instance=request.user - czyli pobranie podanej do formularza nazwy użytkownika, instance=request.user.profile - pobranie profilu użytkownika, który zatwierdził formularz przyciskiem Update
* (5) - część kodu sprawdzająca, czy użytkownik poprawnie wypełnił formularz, a jeżeli tak to wykonująca również takie operacje jak; zapis nowo podanych danych do tabel User oraz Profile: zapis user\_form.save() oraz profile\_form.save(), następnie flashowanie wiadomość o poprawnej zmianie informacji: messages.success(request, “Your profile’s been updated”) i przekierowanie użytkownika do tej samej strony profile   
  - redirect(‘profile’)
* (6) i (7) to linie tworzące formularze do wyświetlenia w ramach zapytania GET; przekazanie do nich argumentów instance=request.user oraz instance=request.user.profile umożliwia wygenerowanie formularza edycji z automatycznie uzupełnionymi polami zgodnie z danymi, jakie są już zapisane w ramach profilu danego użytkownika

Spójrz na ostateczny efekt, jaki powinniśmy otrzymać po przejściu na podstronę /profile:

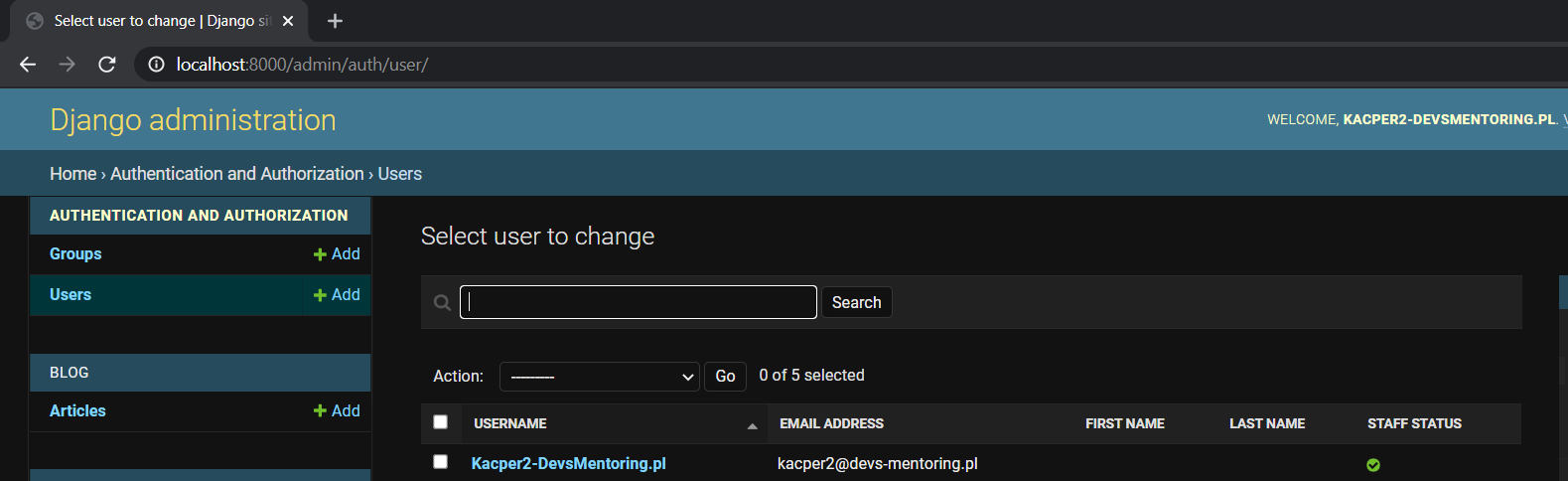


Spróbujmy zaktualizować informacje z poziomu profilu wyżej przedstawionego użytkownika:





Widzimy, że zmiany faktycznie zostały zaaplikowane i Nasz profil się zaktualizował. Upewnijmy się jeszcze, że adres użytkownika wraz z emailem i zdjęciem również zmienił się w panelu administracyjnym i bazie danych. Przejdźmy zatem do Naszego panelu i zweryfikujemy zawartość tabeli Users:

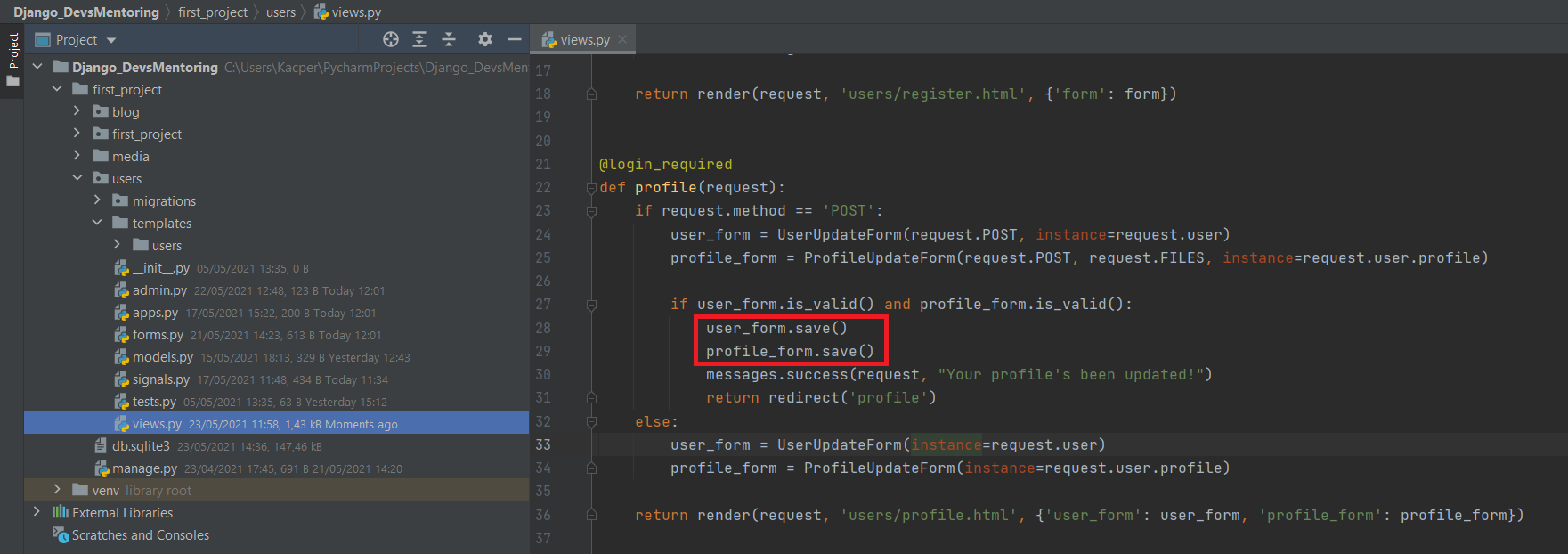


Faktycznie - dane poprawnie się zaktualizowały i wygląda na to, że prawidłowo zaimplementowaliśmy nową funkcjonalność.

**Skalowanie awatarów…**

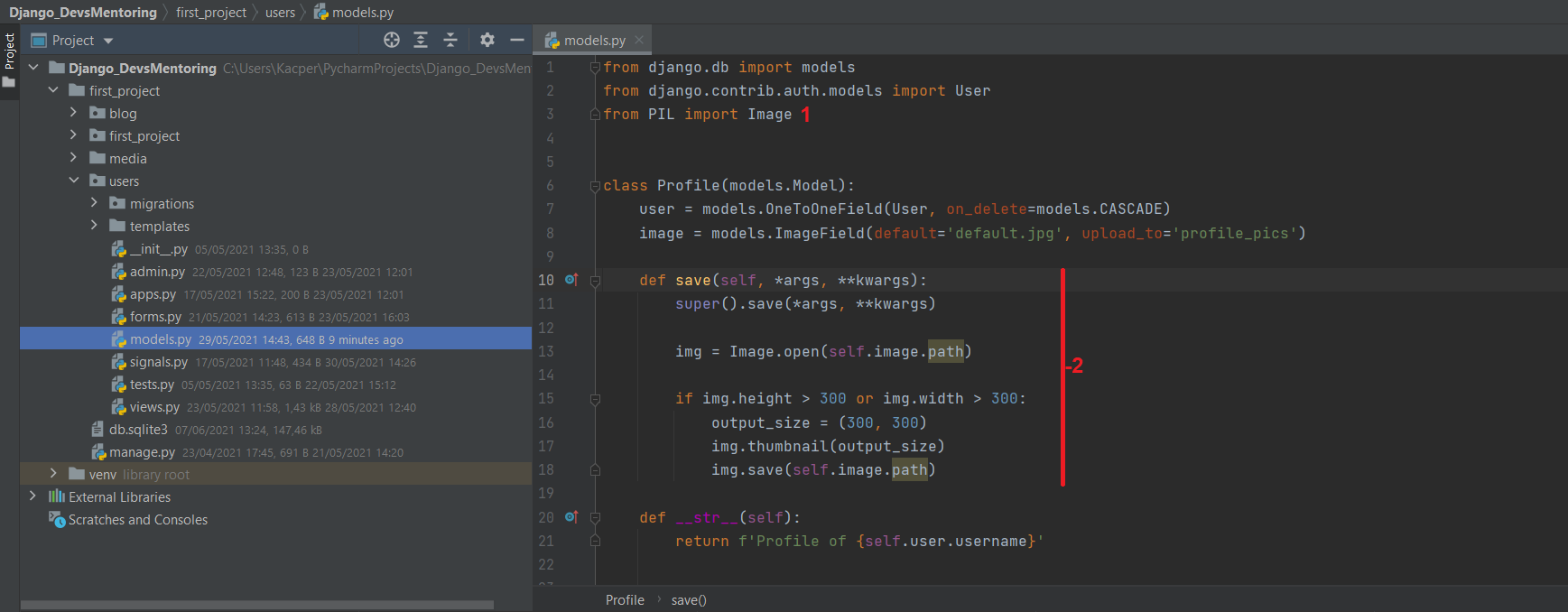
Dając użytkownikowi możliwość uploadowania różnych zdjęć, musimy mieć na uwadze fakt, iż może on wrzucać zdjęcia o całkowicie dowolnym rozmiarze. Dlatego chcielibyśmy przestrzec się przed sytuacją, kiedy user udostępni zdjęcie o bardzo dużym rozmiarze i mogłoby ono nieprawidłowo generować się w Naszym serwisie. Dlatego musimy zmienić nieco proces zapisu informacji o profilu użytkownika.

Przypomnę Ci, że do tej pory informacje do odpowiednich tabel zapisywaliśmy dzięki poniższym liniom kodu:



Naszym zadaniem będzie teraz zmodyfikowanie funkcjonalności wywoływanej metody save na rzecz obiektu profile\_form tak, aby oprócz zapisu informacji do tabeli Profile, odbywało się również skalowanie obrazu w razie przesłanie zbyt dużego awatara.

W tym celu nadpiszemy metodę save() wewnątrz klasy Profile w blog/models.py i dodamy do niej nową funkcjonalność - skalowanie obrazów.



from django.db import models

from django.contrib.auth.models import User

from PIL import Image

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

image = models.ImageField(default='default.png', upload\_to='profile\_pics')

def save(self, \*args, \*\*kwargs):

super().save(\*args, \*\*kwargs)

img = Image.open(self.image.path)

if img.height>300 or img.width > 300:

output\_size = (300, 300)

img.thumbnail(output\_size)

img.save(self.image.path)

def \_\_str\_\_(self):

return f'Profile of {self.user.username}'

* (1) aby umożliwić skalowanie obrazów, musimy wykorzystać klasę Image z biblioteki PIL (Pillow), z której w zasadzie korzystaliśmy już wcześniej przy operowaniu na obrazach
* (2) to cała metoda zapisująca informacje z formularza ProfileUpdateForm do tabeli Profile; zwróć uwagę na logikę kodu skalującego obraz - od 13stej do 18stej linii; jeżeli szerokość lub wysokość obrazu przekracza dopuszczalny rozmiar 300px, to jest ona zmniejszana do maksymalnej dopuszczalnej szerokości i wysokości
* na uwagę zasługuje również dodanie parametrów \*args oraz \*\*kwargs do metody save i przekazywanie ich w wywołaniu super().save(\*args, \*\*kwargs); konieczne jest to, aby zapewnić prawidłowy zapis profilu

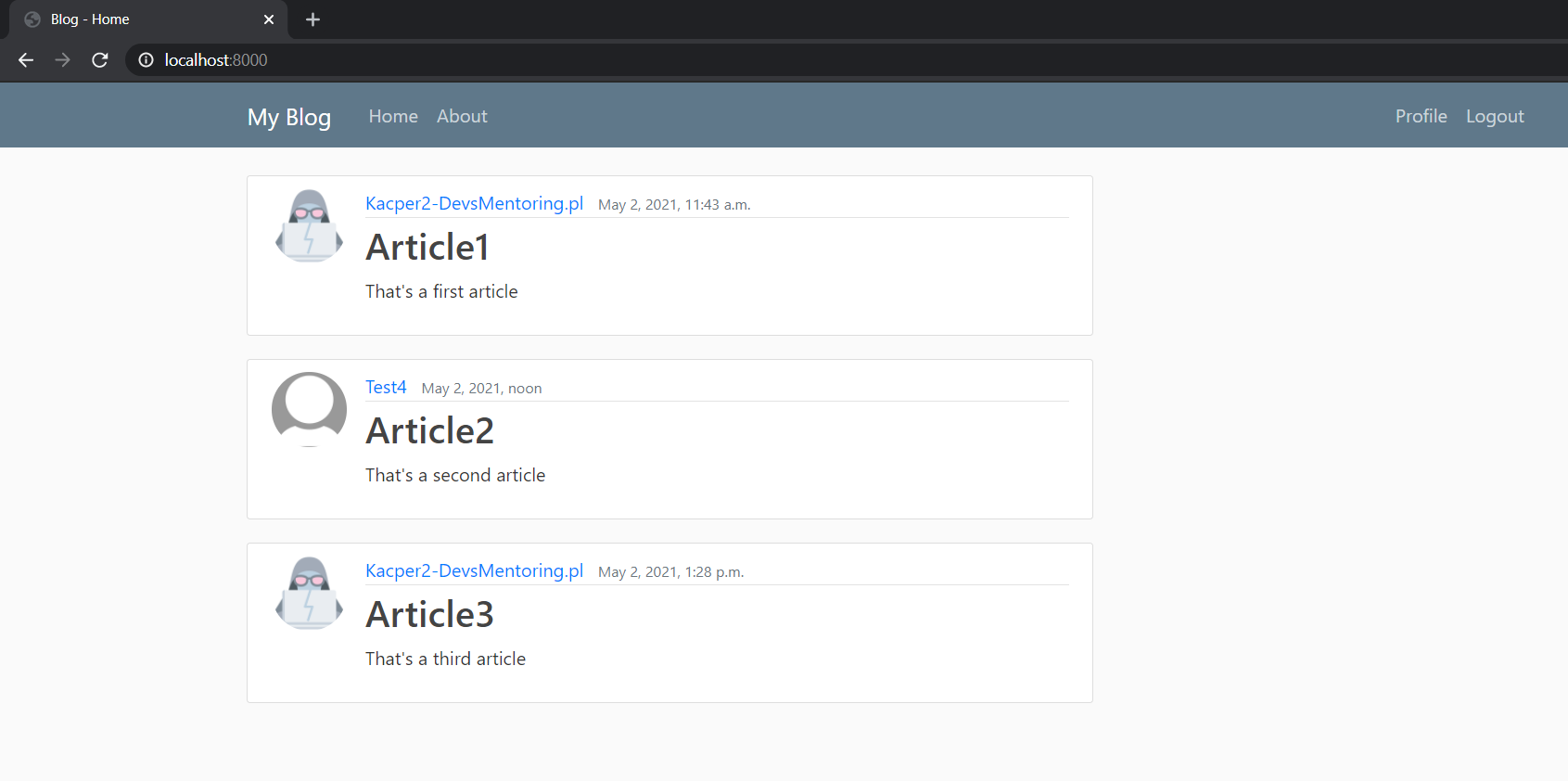
**Wyświetlanie awatara użytkownika przy opublikowanym poście**

Na koniec zajmiemy się wyświetlaniem obrazu przy każdym z postów - tak aby, oprócz tytułu artykułu widoczny był również awatar autora postów. Przypomnę Ci, że w tym momencie na stronie głównej aplikacji widnieją trzy posty: Article1, Article2, Article3. Z poziomu panelu administratora i tabeli Article, zmodyfikowałem nieco jej autorów i są to kolejno: Kacper2-DevsMentoring.pl, Test4 oraz Kacper2-DevsMentoring.pl. Aby umożliwić wyświetlanie awatara przy każdym z postów, wystarczy dodać pojedynczą linię HTML w pliku users/home.html. Spójrz poniżej:



Wspomniana linia kodu jest oznaczona punktem (1).

Po odświeżeniu strony głównej, powinniśmy otrzymać następujący widok:



**Zakończenie**

W najbliższym szkoleniu skupimy się na tym, aby każdy z użytkowników miał możliwość tworzenia własnych artykułów. Tak, aby wszystko działo się z poziomu aplikacji, a nie, jak do tej pory, panelu administratora.