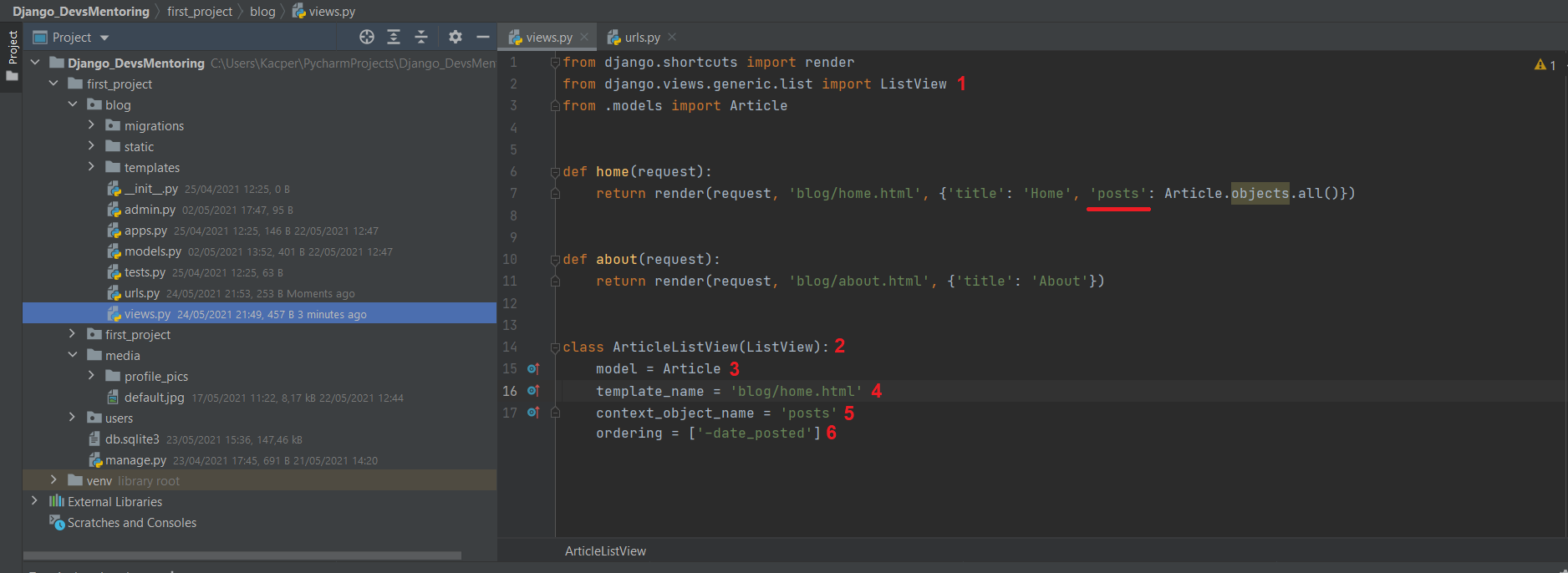
**Django - Szkolenie 8**

**CRUD artykułów**

**Nowe funkcjonalności...**

Nadszedł czas na zapewnienie pojedynczemu użytkownikowi możliwości publikowania własnych artykułów. Tak, aby wszystko działo się z poziomu serwisu, a nie panelu administratora. W tym celu wykorzystamy, tzw. class-based view. Najbardziej popularnymi klasami tego typu są DetailView oraz ListView. Różnica między nimi jest dość prosta do zrozumienia - ListView przechowuje i wyświetla kolejne elementy w formie listy (np. lista wyświetlanych artykułów na blogu), natomiast DetailView to widok powiązany z konkretnym już postem - umożliwiający przedstawienie pojedynczego artykułu. Jednym słowem - przedstawienie widoków w formie klasy, zapewnia o wiele wygodniejszy proces tworzenia funkcjonalności. Dodatkowo, wykorzystując ten rodzaj widoków, będziemy mogli zaimplementować funkcjonalność usuwania (DeleteView), aktualizacji (UpdateView) oraz tworzenia artykułów (CreateView).

Utwórzmy zatem pierwszy widok klasowy - ArticleListView, który reprezentować będzie cały panel bloga z dodanymi artykułami. W tym celu udajmy się do pliku blog/views.py i umieśćmy w nim klasę ArticleListView:



from django.shortcuts import render

from django.http import HttpResponse

from .models import Article

from django.views.generic.list import ListView

def home(request):

return render(request, 'blog/home.html', {'title': 'Home', 'posts': Article.objects.all()})

# return render(request, 'blog/home.html', {'title': 'Home', 'posts': POSTS})

def about(request):

return render(request, 'blog/about.html', {'title': 'About'})

# Create your views here.

class ArticleListView(ListView):

model = Article

template\_name = 'blog/home.html'

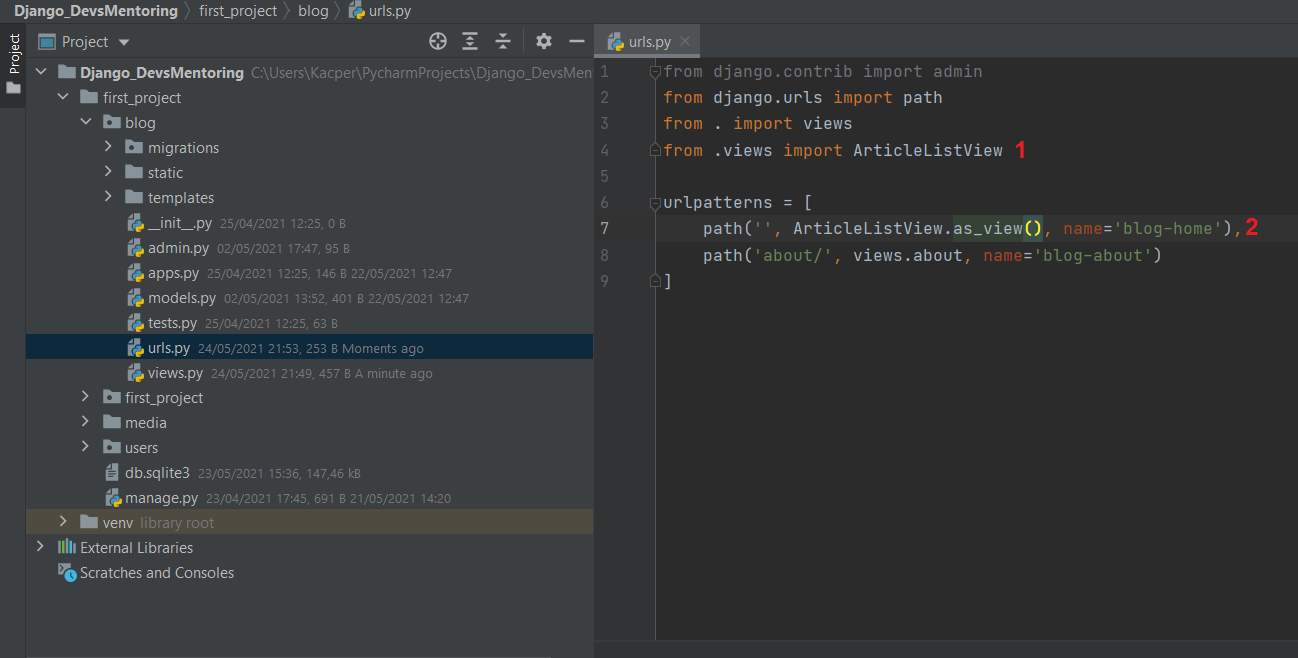
context\_object\_name = 'posts'

ordering = ['-date\_posted']

Cyframi oznaczyłem kluczowe zmiany w kodzie:

* (1) - to zaimportowanie z modułu django.views.generic.list klasy ListView, w której przechowywać będziemy wszystkie artykuły bloga
* (2) - to klasa dziedzicząca po ListView pełniąca rolę ClassView; umieściliśmy w niej w sumie 4 pola
* (3) - to zmienna przechowująca nazwę tabeli, z którą powiązany jest ListView - z racji, iż posty umieszczane na blogu to kolejne rekordy tabeli Article - to właśnie jej nazwę przypisaliśmy do model
* (4) - template\_name - to pole określające, w którym szablonie generowany będzie widok listowy; w Naszym przypadku widok bloga generowany jest na blog/home.html, stąd też podanie takiej, a nie innej wartości
* (5) - odwołuje się do zmiennej, pod którą trzymane są i rozumiane przez silnik Jinja posty bloga; nazwę tej zmiennej możemy odczytać, analizując funkcję home (została ona podreślona czerwoną linią)
* (6) - umieszczenie tego pola było konieczne z racji, iż domyślnie posty wewnątrz ArticleListView porządkowane są od najstarszego do najnowszego; z racji, iż chcielibyśmy, aby aplikacja umieszczała najnowsze publikacje na samej górze strony, zmieniliśmy kolejność porządkowania, przypisując listę z wartością [‘-date\_posted’] do zmiennej ordering; zwróć uwagę, iż podany w liście string to nazwa kolumny tabeli Article, przechowującej aktualną datę i czas publikacji posta

Na koniec, przeszedłem do pliku blog/urls.py i zmodyfikowałem nieco listę ścieżek URL - tak, aby endpoint ‘’ prowadził do utworzonej przed chwilą klasy widoku (ArticlelistView.as\_view()), a nie widoku home tak jak do tej pory. Standardowo zaimportowałem również klasę ArticleListView, do której konieczne było odwołanie w liście ulrpatterns.



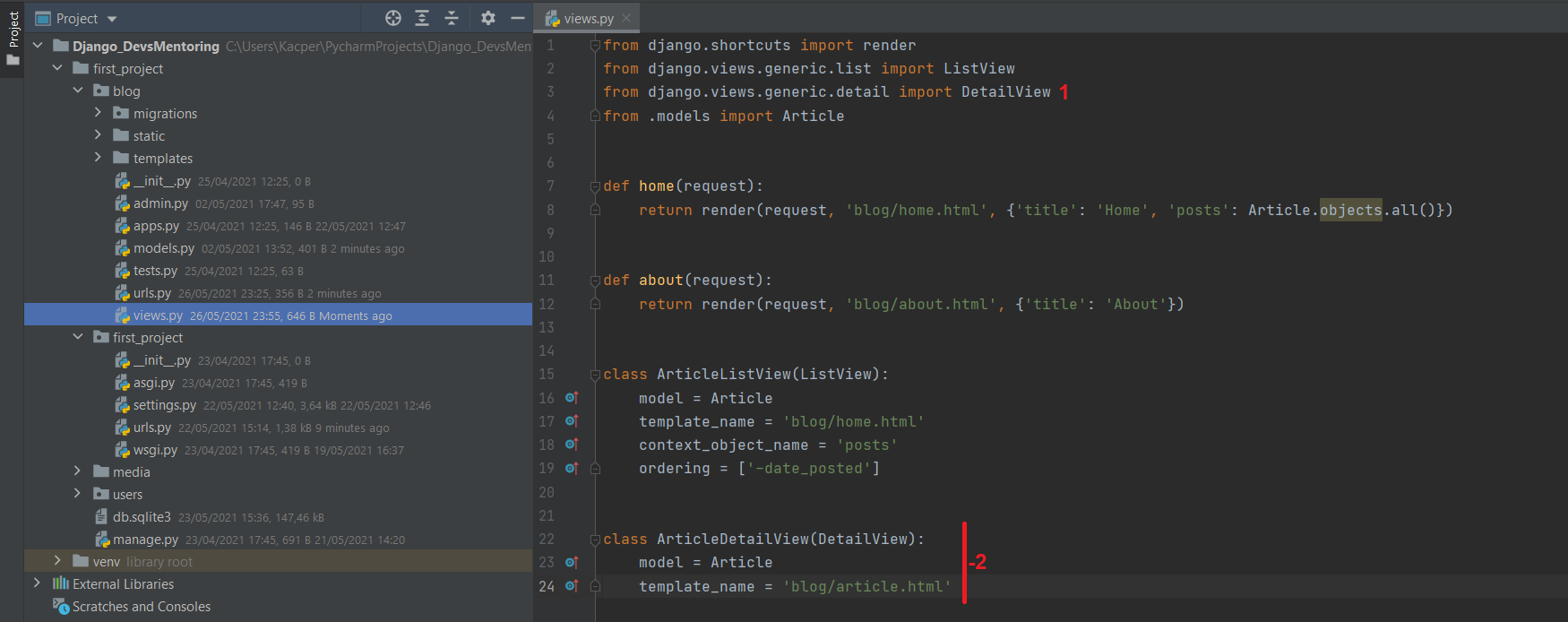
**Czy coś się zmieniło?**

Po wyżej wprowadzonych zmianach, widok aplikacji nie powinien się w ogóle zmienić. Celem modyfikacji było bowiem przygotowanie projektu pod możliwość łatwego dodawania kolejnych artykułów bloga (ListView można wygodnie rozszerzać o kolejne elementy widoku). Dlatego na ten moment oczekujemy tylko prawidłowo uruchamiającej się aplikacji i wyświetlającej wszystkie artykułu w taki sam sposób jak do tej pory.

**Dodanie kolejne klasy widoku**

Klasę reprezentującą listę artykułów już dodaliśmy, teraz nadszedł czas na przygotowanie widoku, który będzie pełnił rolę “podglądu” pojedynczego artykułu. Otóż chcielibyśmy, aby użytkownik po wybraniu z listy określonego posta, był przekierowywany do oddzielnej strony, na której prezentować się będzie wybrany artykuł. Do tego celu posłuży nam widok ArticleDetailView.

Aby utworzyć wyżej wspomniany widok, przejdziemy do pliku blog/views.py i dodamy w nim następującą klasę (uprzednio importując klasę bazową DetailView):



from django.shortcuts import render

from django.http import HttpResponse

from .models import Article

from django.views.generic.list import ListView

from django.views.generic.detail import DetailView

def home(request):

return render(request, 'blog/home.html', {'title': 'Home', 'posts': Article.objects.all()})

# return render(request, 'blog/home.html', {'title': 'Home', 'posts': POSTS})

def about(request):

return render(request, 'blog/about.html', {'title': 'About'})

# Create your views here.

class ArticleListView(ListView):

model = Article

template\_name = 'blog/home.html'

context\_object\_name = 'posts'

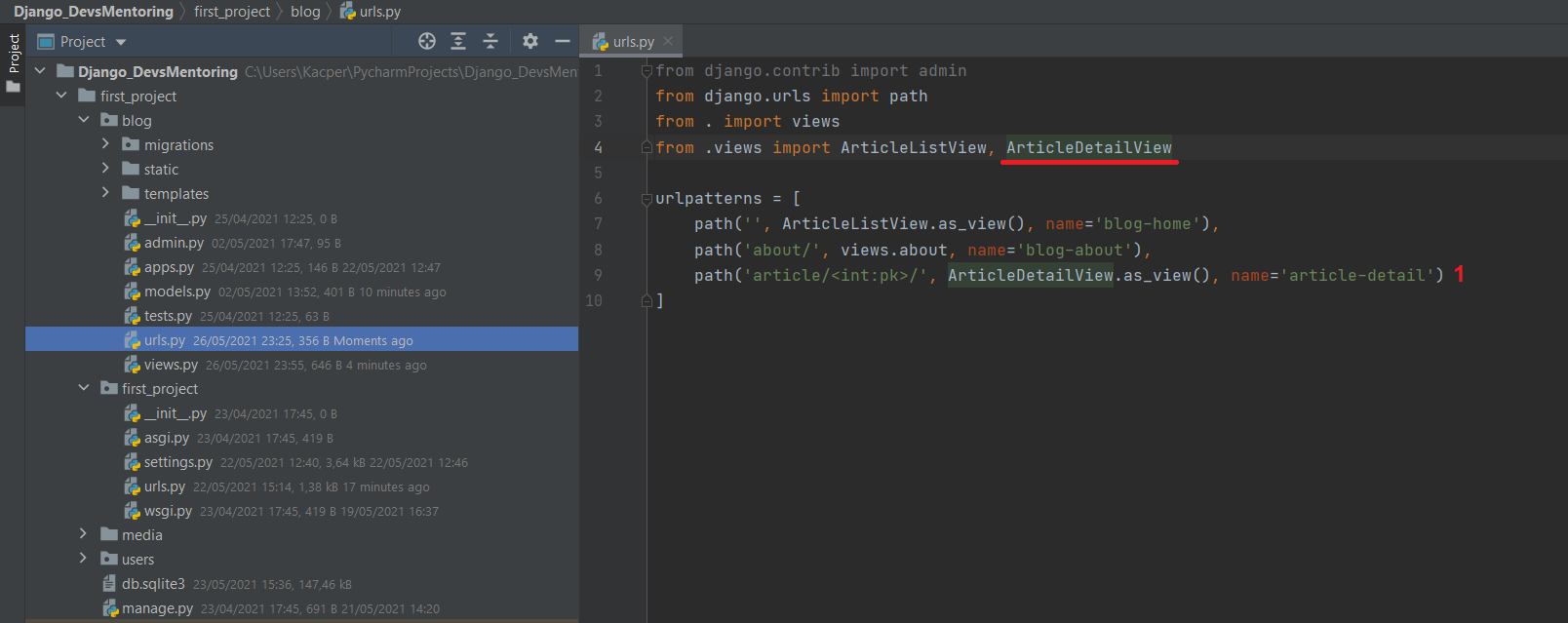
ordering = ['-date\_posted']

class = ArticleDetailView(DetailView):

model = Article

template\_name = 'blog/article.html'

Z racji, iż już robiłem niemal to samo przy ArticleListView, nie muszę opisywać dodanych pól do tak stworzonej klasy. Przejdźmy zatem płynnie do pliku blog/urls.py i dodajmy w nim odpowiednie referencje do pojedynczej strony artykułu.



from django.contrib import admin

from django.urls import path

from . import views

from .views import ArticleListView, ArticleDetailView

app\_name = 'blog'

urlpatterns = [

path("", ArticleListView.as\_view(), name='blog\_home'),

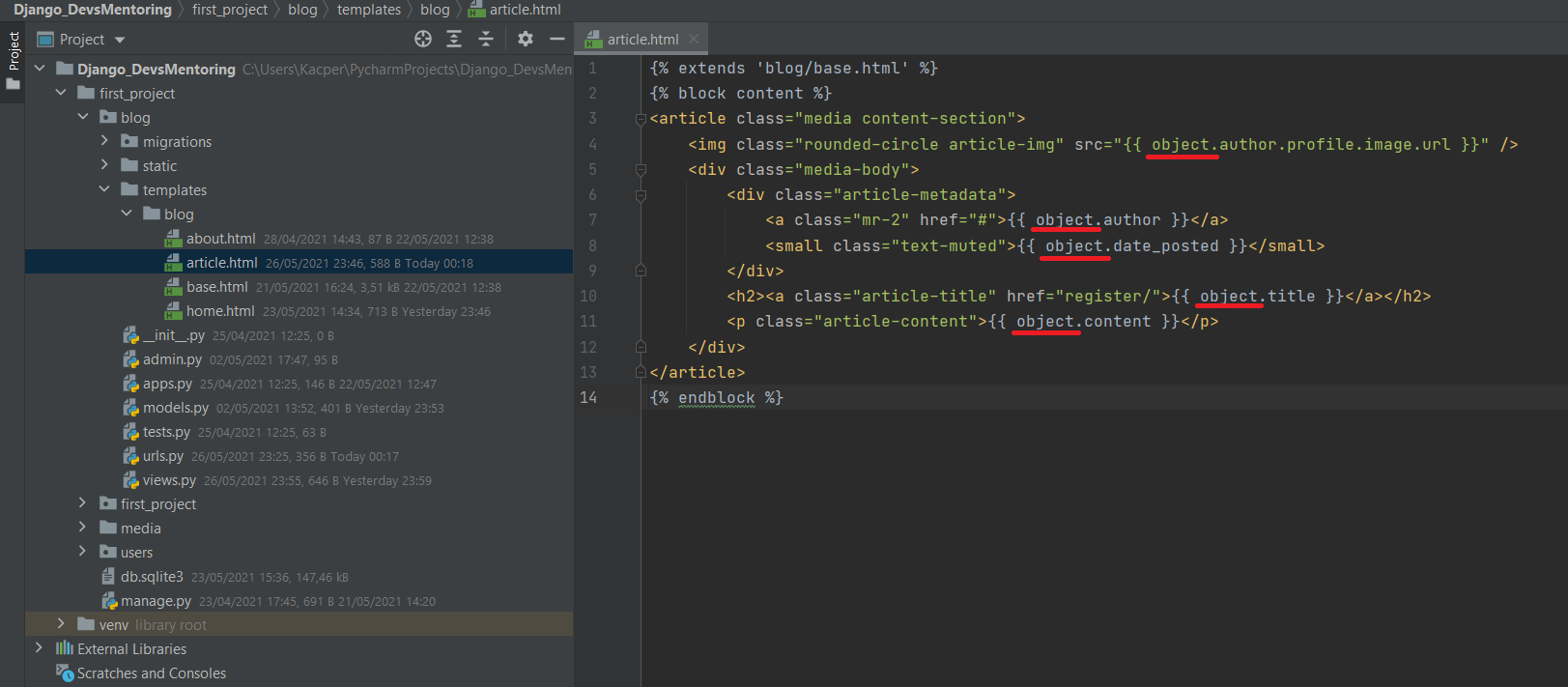
path("about/", views.about, name='blog\_about'),

path("article/<int:pk>", ArticleDetailView.as\_view(), name='article-detail'),

]

Podkreślony fragment kodu to zaimportowanie dopiero co utworzonej klasy.   
**(1)** to natomiast ścieżka do endpointa postaci /article/<int:pk>/, gdzie pk to klucz główny danego artykułu. Zwróć uwagę, że w formie pk nie musimy podawać konkretnego pola z tabeli Article - klucz główny automatycznie zostanie dopasowany do danego rekordu i co więcej obiekt takiego rekordu zostanie wygodnie przesłany do szablonu (tak naprawdę bez naszego udziału). Dzięki temu będziemy mieli łatwą możliwość odwoływania się do artykułu, dla którego wyświetlać będzie szczegółowy widok.

Zanim jednak skupimy się na wyświetlaniu określonych artykułów, musimy utworzyć nowy plik .html reprezentujący pojedynczy post na blogu. W tym celu dodamy nowy plik article.html do blog/templates/blog/:



*Plik article.html*

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

{% block content %}

<article class="media content-section">

<img class="rounded-circle article-img" src="{{ object.author.profilce.image.url }}" />

<div class="media-body">

<div class="article-metadata">

<a class="mr-2" href="#"> {{ object.author }} </a>

<small class="text-muted">{{ object.date\_posted }}</small>

</div>

<h2><a class="article-title" href="register/">{{ object.title }}</a></h2>

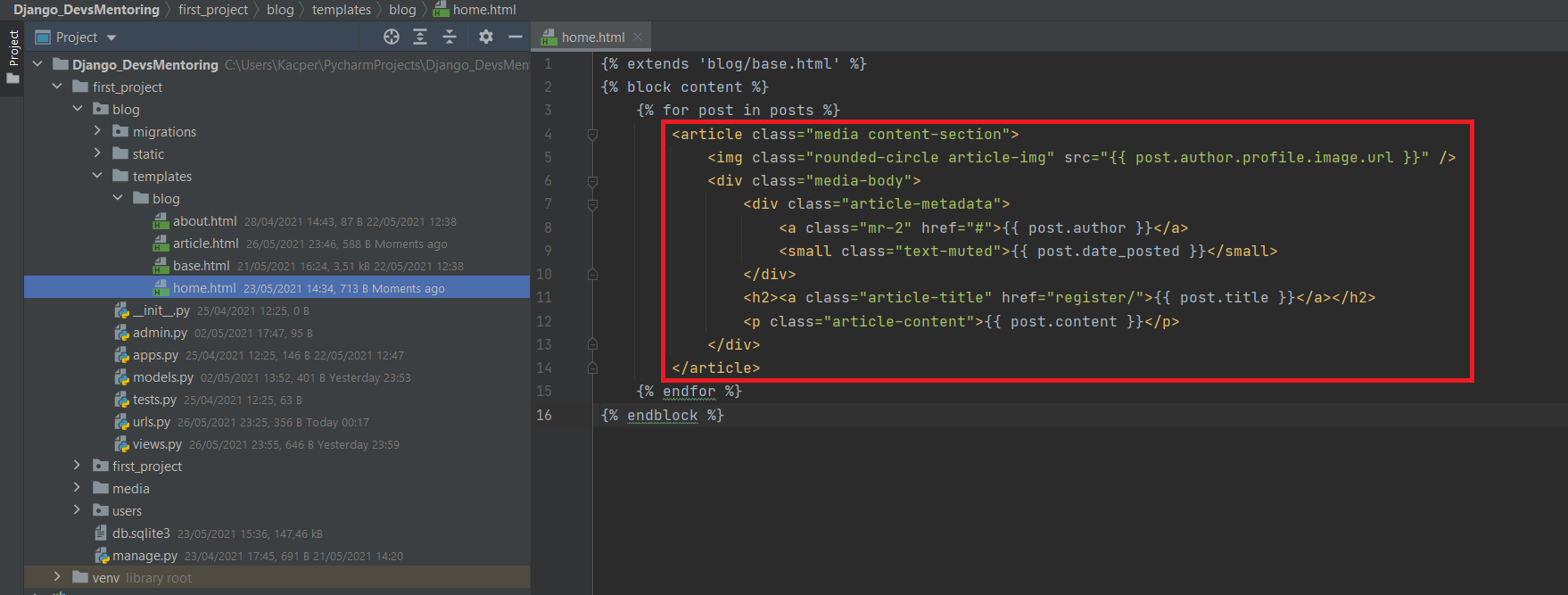
<p class="article-content">{{ object.content }}</p>

</div>

</article>

{% endblock %}

Zawartość powyższego szablonu przekopiowałem z pliku blog/home.html, usuwając jedynie pętle przechodzącą po wszystkich artykułach.

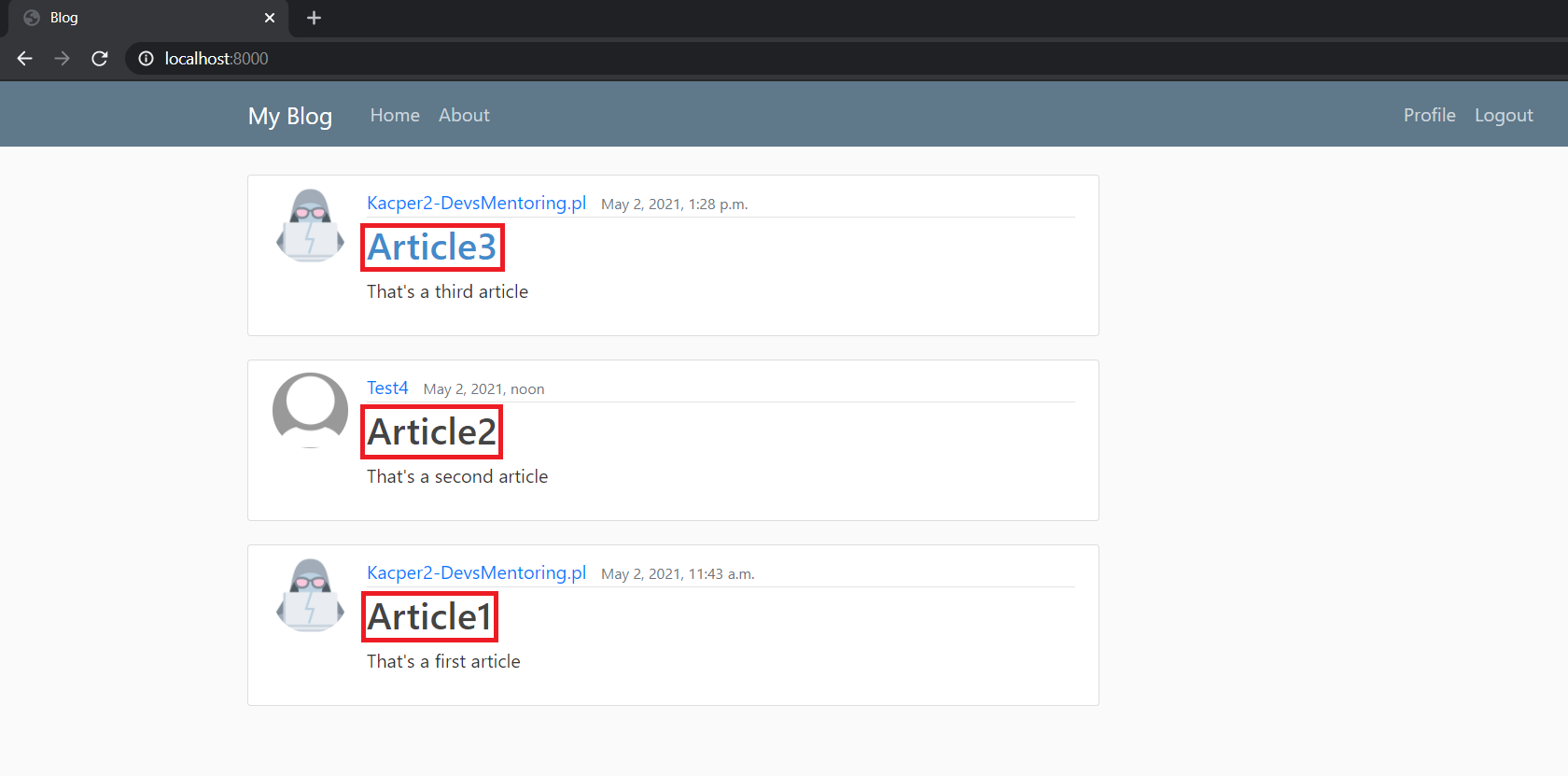


*Plik home.html*

Zwróć uwagę na podkreślone fragmenty kodu w pliku article.html. Z racji, iż artykuł o określonym pk (pobranym z adresu URL - /article/**<int:pk>**) jest automatycznie zaciągany do Naszego szablonu, to występuje on pod nazwą object. I z tego właśnie powodu, chcąc odwołać się do poszczególnych pól artykułu - zapiszemy, np. object.author, object.date\_posted.

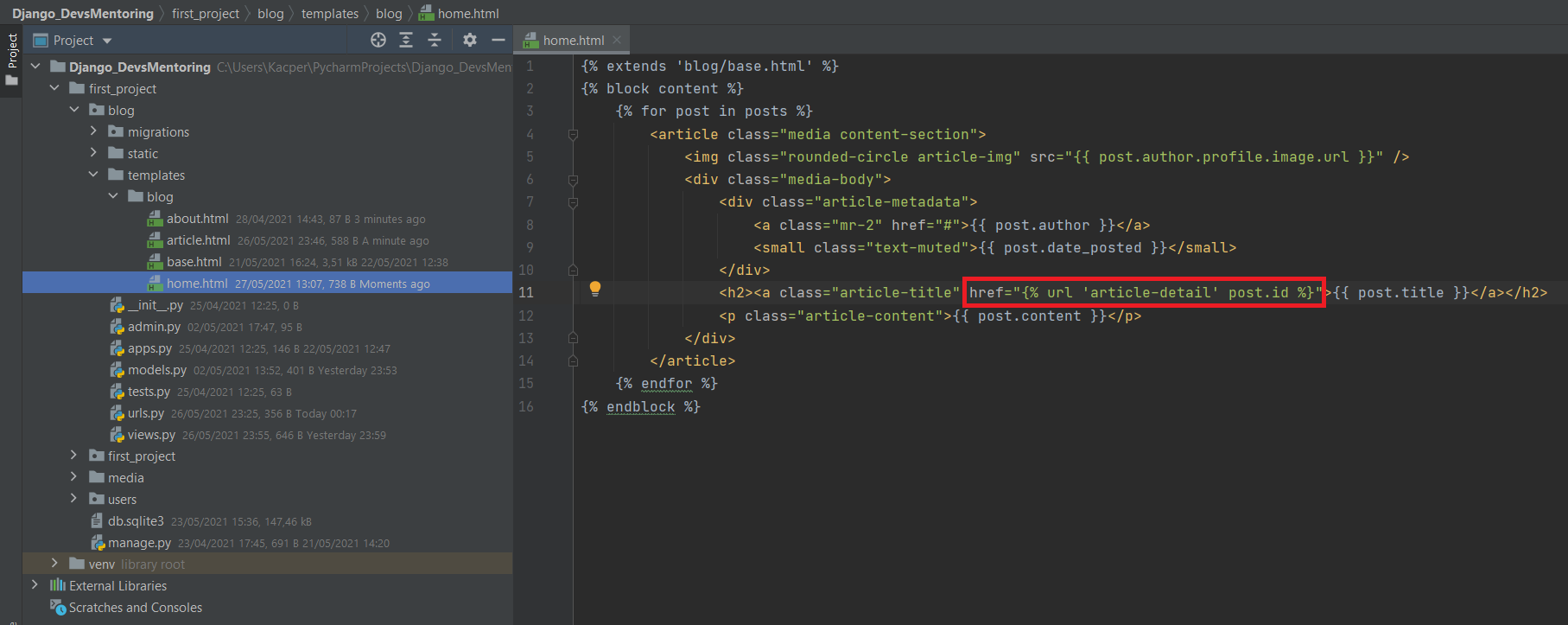
**Dodanie linku przekierowującego**

Zanim sprawdzimy, czy “szczegółowy widok” do danego artykułu faktycznie działa, musimy dodać możliwość przechodzenia przez kliknięcie w dany link do określonego artykułu. Do danego posta będziemy się przenosić przez niżej zaznaczone odnośniki:



*Odnośniki do poszczególnych artykułów*

W tym celu, zmodyfikujemy szablon home.html i dodamy odnośniki do odpowiednich podstron.



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

{% block content %}

{% for post in posts %}

<article class="media content-section">

<img class="rounded-circle article-img" src="{{ post.author.profile.image.url }}">

<div class="media-body">

<div class="article-metadata">

<a class="mr-2" href="#">{{ post.author }}</a>

<small class="text-muted">{{ post.date }}</small>

</div>

<h2><a class="article-title" href="{% url 'blog:article-detail' post.id %}">{{ post.title }}</a></h2>

<p class="article-content">{{ post.content }}</p>

</div>

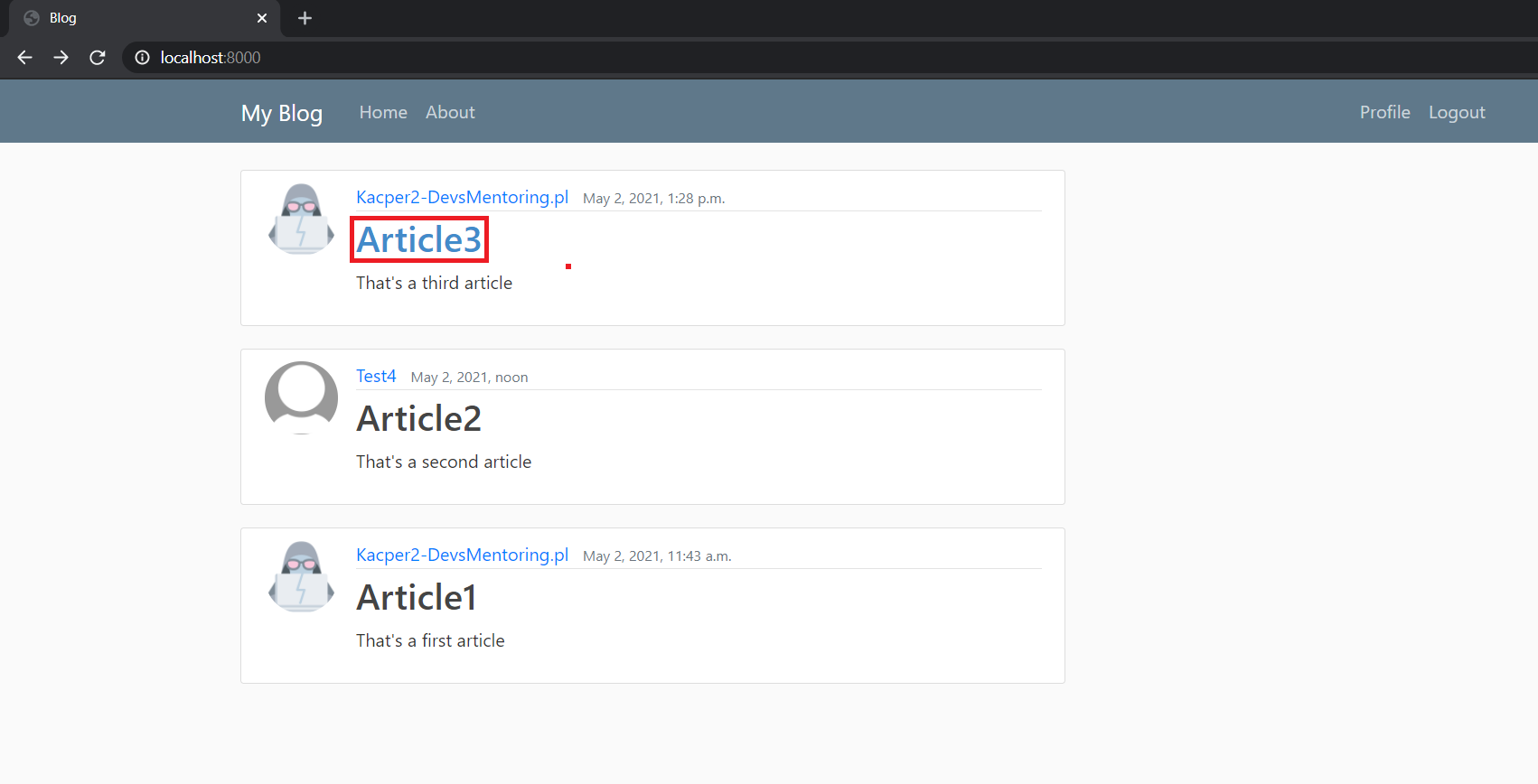
</article>

{% endfor %}

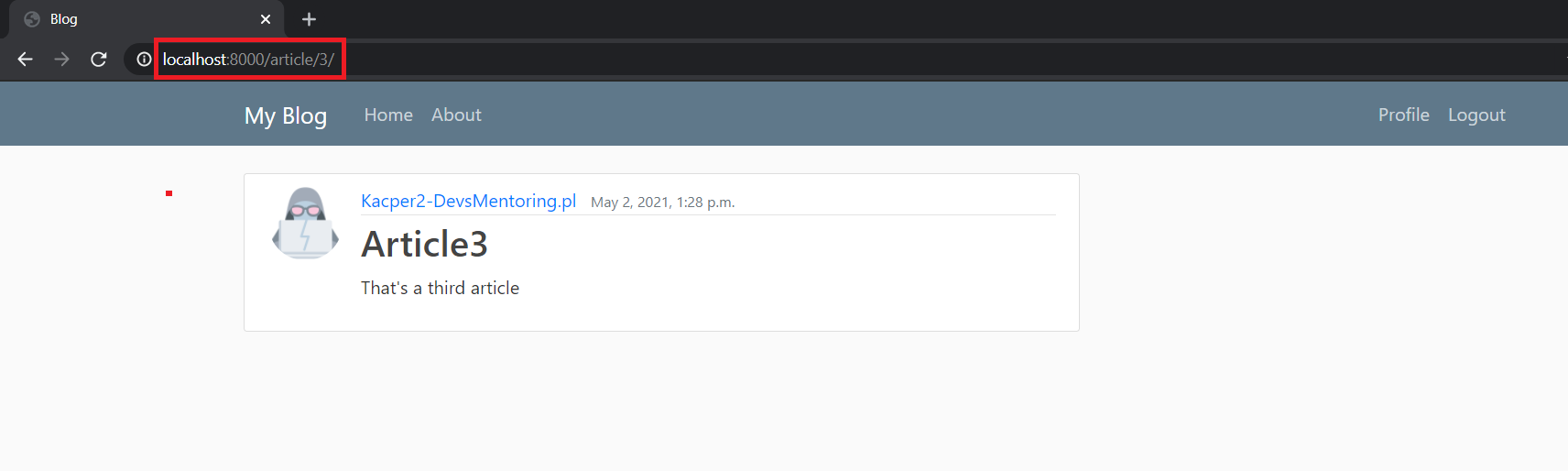
{% endblock %}

Kolorem zaznaczyłem wprowadzone w pliku zmiany. Jak widzisz, wystarczy tak naprawdę zapewnić odpowiednią wartość znacznikowi href - odwołanie do endpointa nazwanego jako ‘article-detail’ z odpowiednim parametrem - kluczem głównym post.id. Dzięki takiej zmianie, kliknięcie w dany tytuł artykułu, przekieruje Nas do szczegółowego rozwinięcia wybranego posta.

W takim przypadku, po kliknięciu w napis Article3…



Zostajemy przekierowani do…



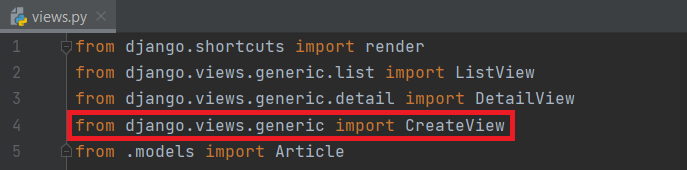
… artykułu o zaznaczonym wyżej adresie URL.

Wygląda na to, że wszystko śmiga i jesteśmy w stanie podglądać utworzone artykuły. Zajmijmy się teraz dodaniem możliwości tworzenia własnych artykułów z poziomu aplikacji (nie panelu administratora).

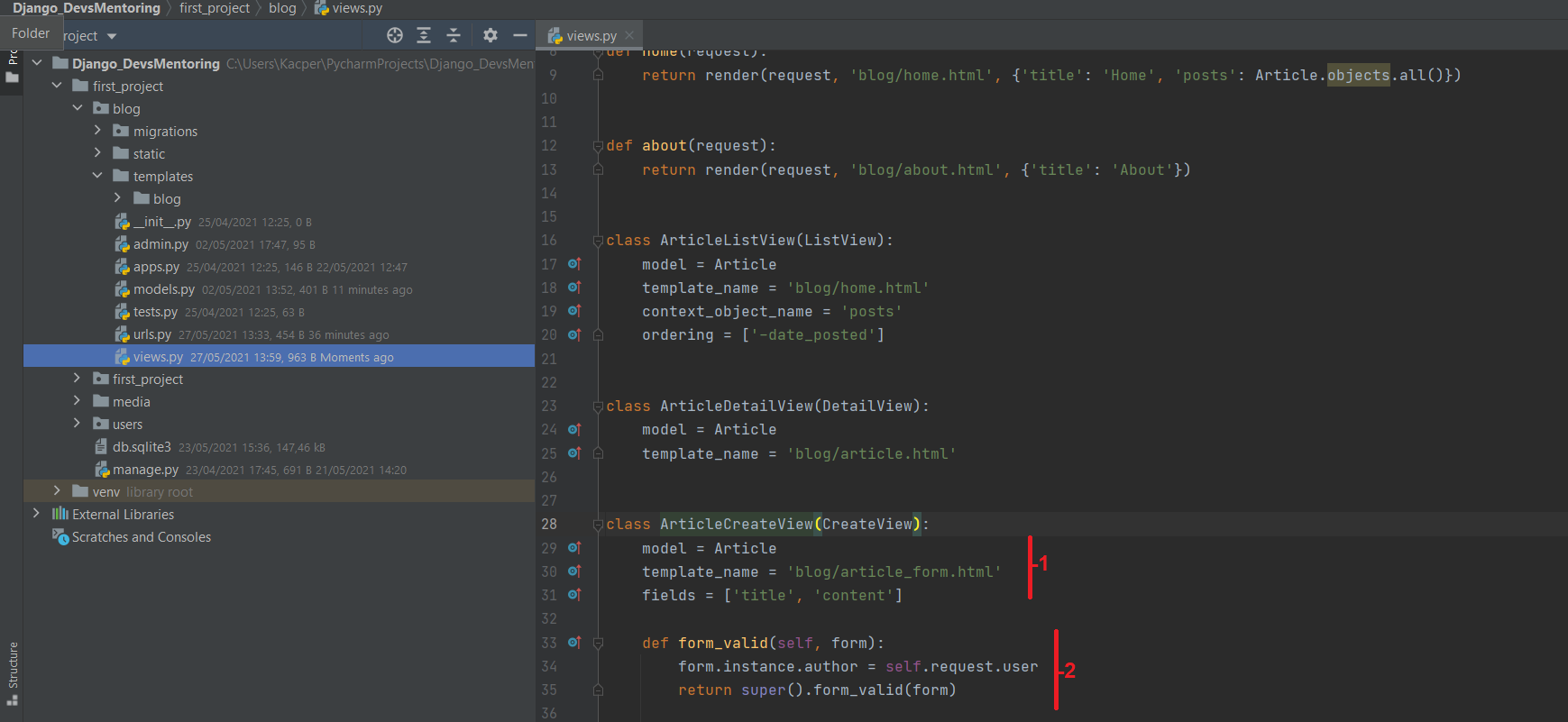
Panel tworzenia nowych artykułów

W pierwszej kolejności musimy dodać nową klasę widoku w pliku blog/views.py. Będzie to klasa dziedzicząca po CreateView.

Zaimportujmy potrzebne zasoby:



Następnie, mając już do dyspozycji CreateView, stwórzmy nową klasę ArticleCreateView i umieścimy w niej odpowiednie pola:



from django.shortcuts import render

from django.http import HttpResponse

from .models import Article

from django.views.generic.list import ListView

from django.views.generic.detail import DetailView

from django.views.generic import CreateView

def home(request):

return render(request, 'blog/home.html', {'title': 'Home', 'posts': Article.objects.all()})

# return render(request, 'blog/home.html', {'title': 'Home', 'posts': POSTS})

def about(request):

return render(request, 'blog/about.html', {'title': 'About'})

# Create your views here.

class ArticleListView(ListView):

model = Article

template\_name = 'blog/home.html'

context\_object\_name = 'posts'

ordering = ['-date\_posted']

class ArticleDetailView(DetailView):

model = Article

template\_name = 'blog/article.html'

class ArticleCreateView(CreateView):

model = Article

template\_name = 'blog/article\_form.html'

fields = ['title', 'content']

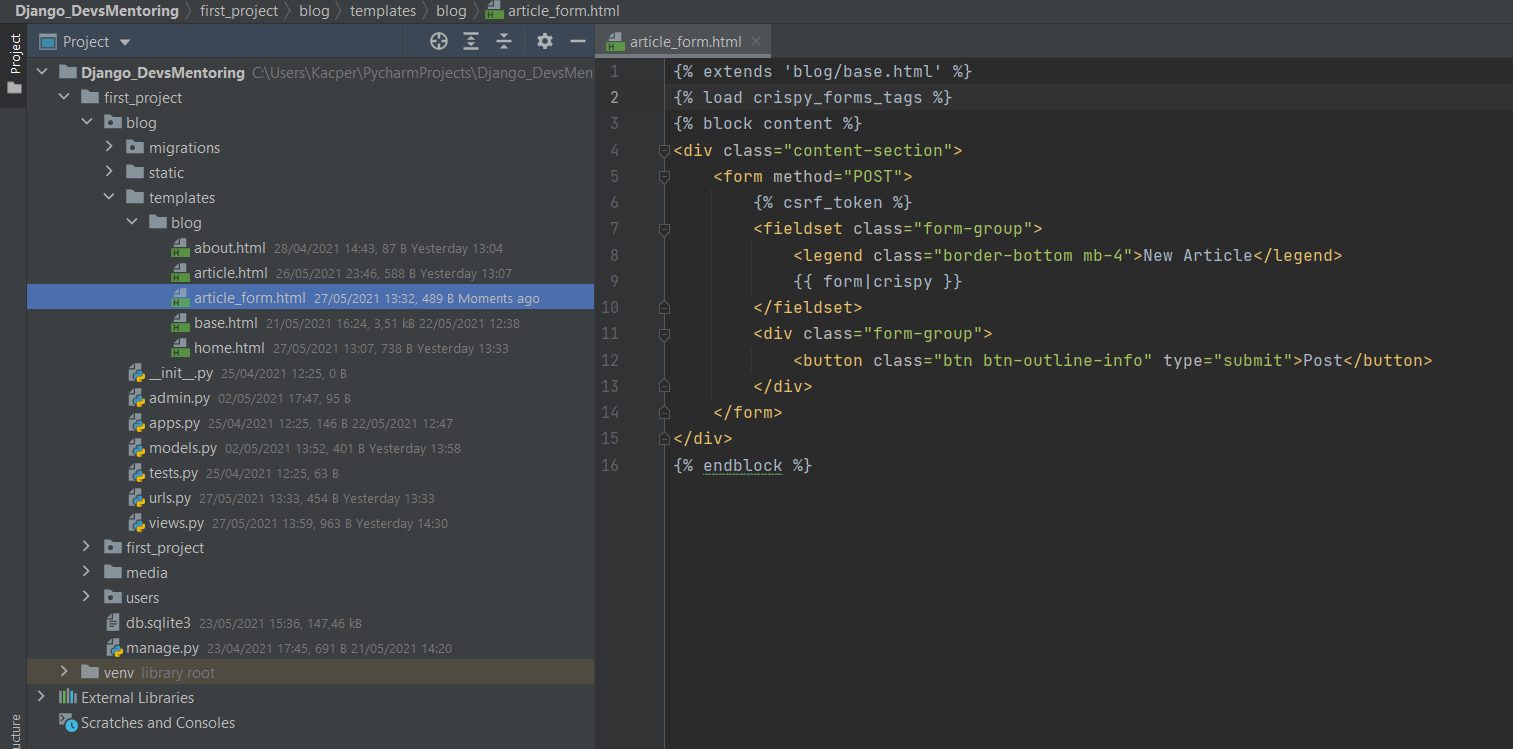
def form\_valid(self, form):

form.instance.author = self.request.user

return super().form\_valid(form)

* (1) to zbiór znanych Ci już pól; na uwagę zasługuje jedynie atrybut fields, który umożliwi Nam dostosowanie pól formularza (który za chwilę umieścimy w szablonie blog/article\_form.html) i dzięki temu będziemy mieli dostęp do dwóch modyfikowalnych rekordów: title oraz content.
* (2) to nowa metoda, która umożliwi Nam automatyczne przypisanie nowo utworzonego artykułu do autora - użytkownika serwisu; form\_valid() jest bowiem przeciążeniem istniejącej metody w CreateView i jest on uruchamiany automatycznie przy sprawdzaniu poprawności formularza (czyli po naciśnięciu przycisku submitującego formularz); jak już wspomniałem, takie rozwiązanie umożliwi Nam w efektywny sposób przypisywać utworzony artykuł do aktualnie zalogowanego użytkownika: form.instance.author = self.request.user

Mając już tak zaprogramowany backend, nadszedł czas na dostarczenie frontendu aplikacji. Utwórzmy zatem nowy plik blog/templates/blog/article\_form.html i umieśćmy w nim następujący kod (będzie to kod pobrany i nieco zmodyfikowany z istniejącego już pliku users/templates/users/register.html):



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

{% load crispy\_forms\_tags %}

{% block content %}

<div class="content-section">

<form method="POST">

{% csrf\_token %}

<fieldset class="form-group">

<legend class="border-bottom mb-4">New Article</legend>

{{ form|crispy }}

</fieldset>

<div class="form-group">

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

</div>

</form>

</div>

{% endblock %}

Na koniec - zarejestrujmy przed chwilą utworzony widok ArticleCreateView pod endpointem /article/new.



"""

URL configuration for first\_project project.

The `urlpatterns` list routes URLs to views. For more information please see:

https://docs.djangoproject.com/en/5.0/topics/http/urls/

Examples:

Function views

1. Add an import: from my\_app import views

2. Add a URL to urlpatterns: path('', views.home, name='home')

Class-based views

1. Add an import: from other\_app.views import Home

2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home')

Including another URLconf

1. Import the include() function: from django.urls import include, path

2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))

"""

from django.contrib import admin

from django.urls import path

from . import views

from .views import ArticleListView, ArticleDetailView, ArticleCreateView

app\_name = 'blog'

urlpatterns = [

path("", ArticleListView.as\_view(), name='blog\_home'),

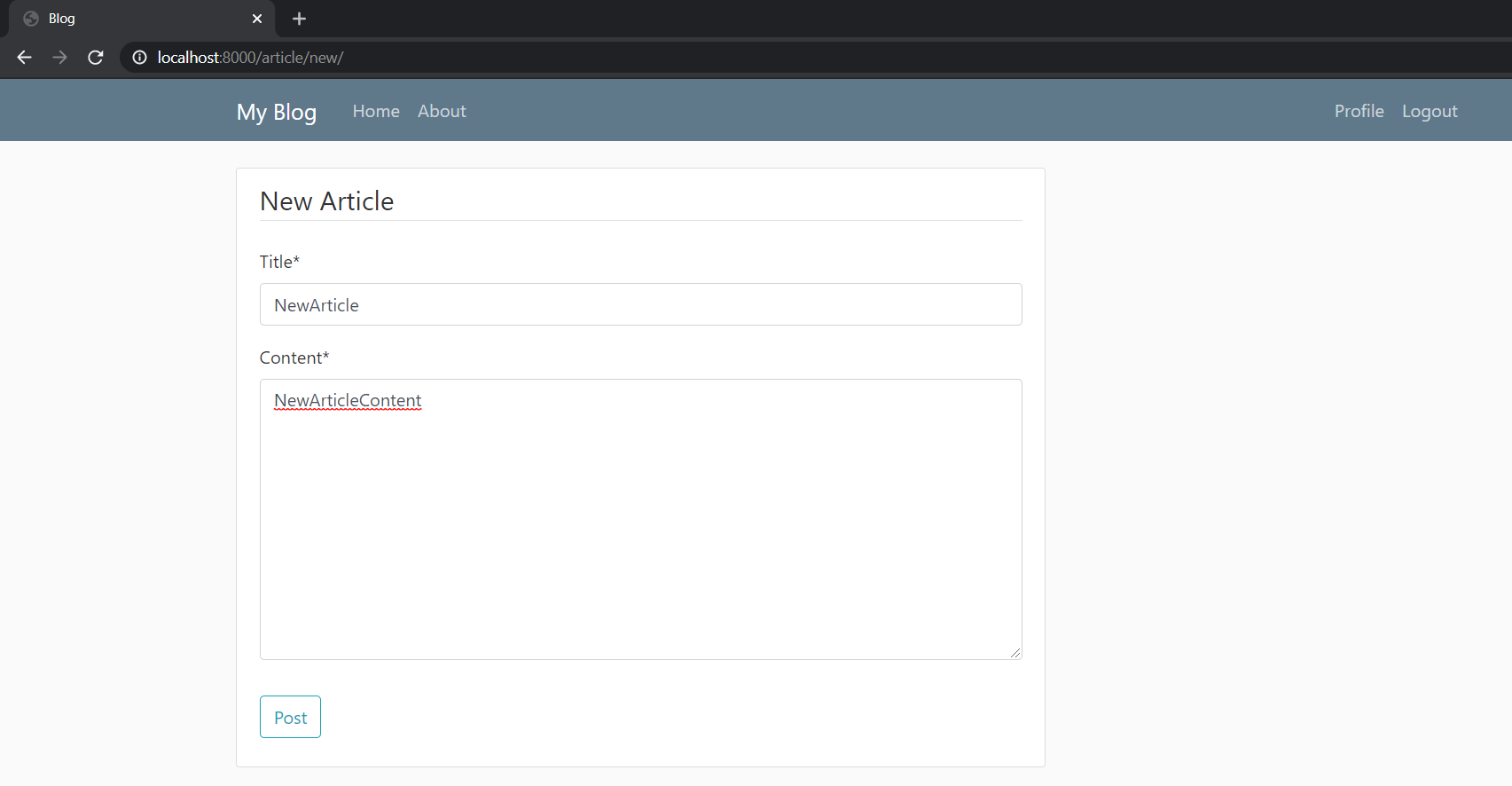
path("about/", views.about, name='blog\_about'),

path("article/<int:pk>", ArticleDetailView.as\_view(), name='article-detail'),

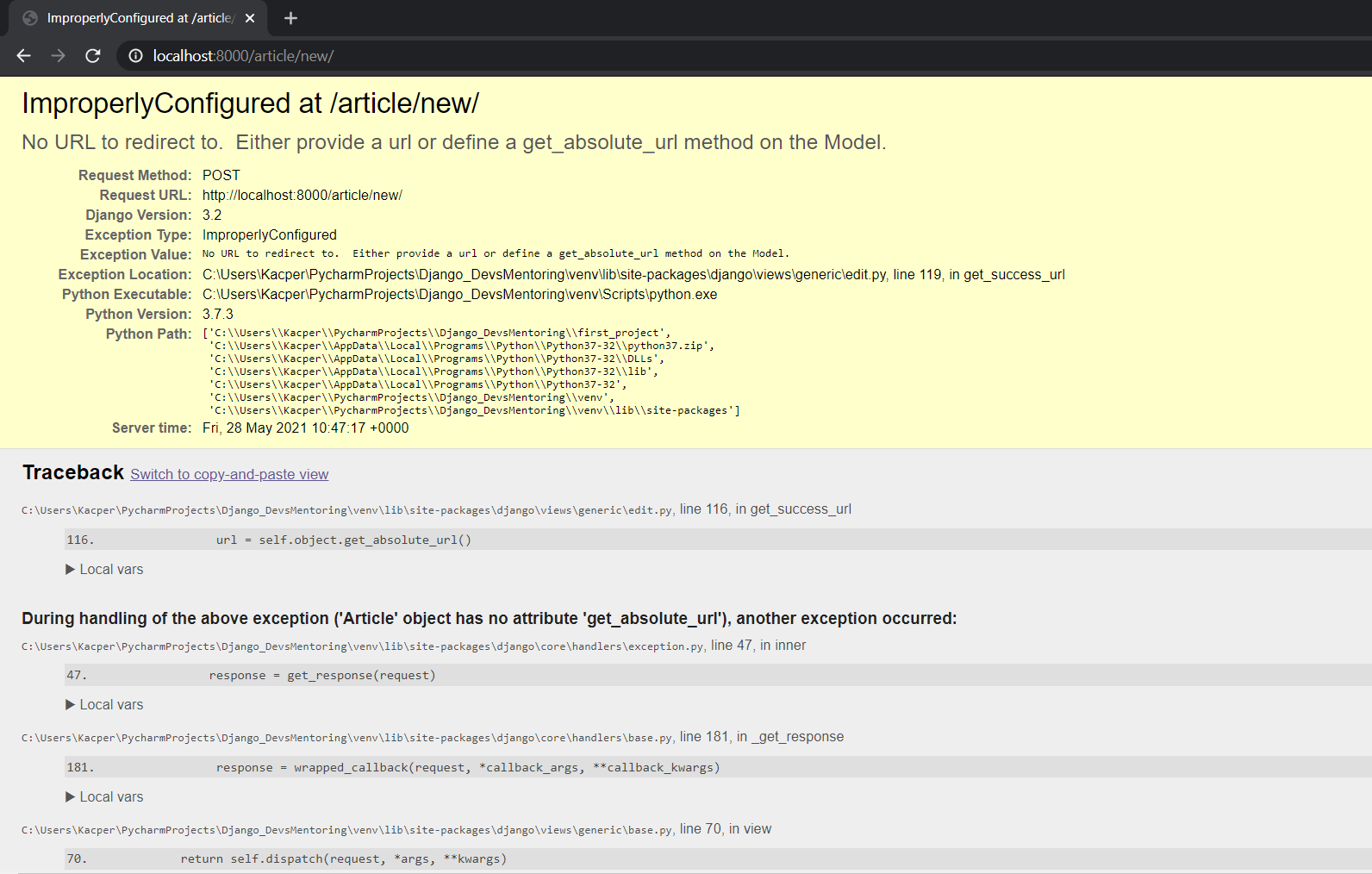
path("article/new/", ArticleCreateView.as\_view(), name='article-create'),

]

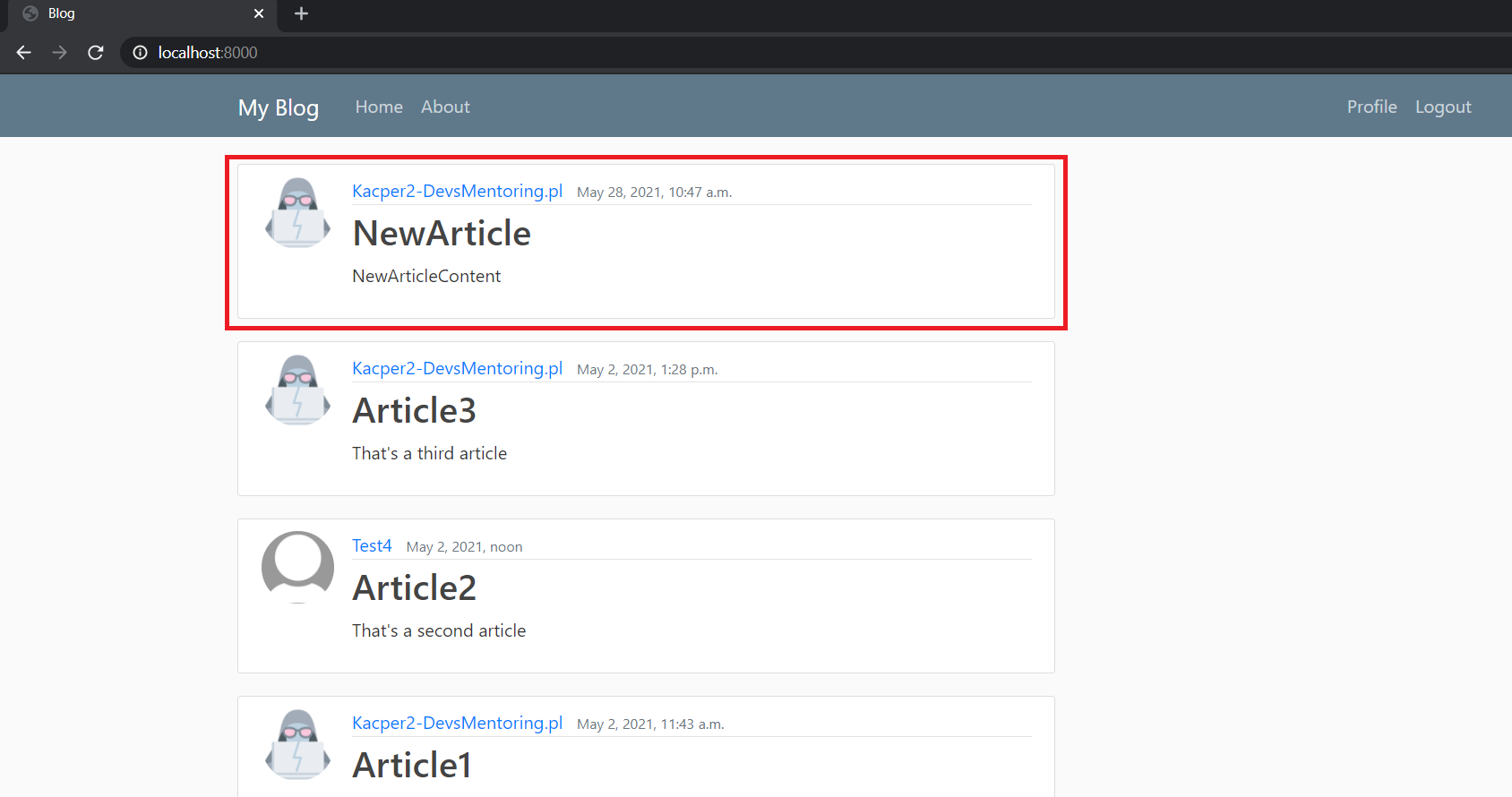
Następnie przejdźmy do Naszej aplikacji i endpointa /article/new/ oraz spróbujmy dodać nowy artykuł.



Podstrona faktycznie prawidłowo się wyświetla, jednak po wciśnięciu przycisku Post, nie zostaliśmy niestety przekierowani do żadnej z podstron aplikacji, tylko napotkaliśmy błąd “No URL to redirect to.”.

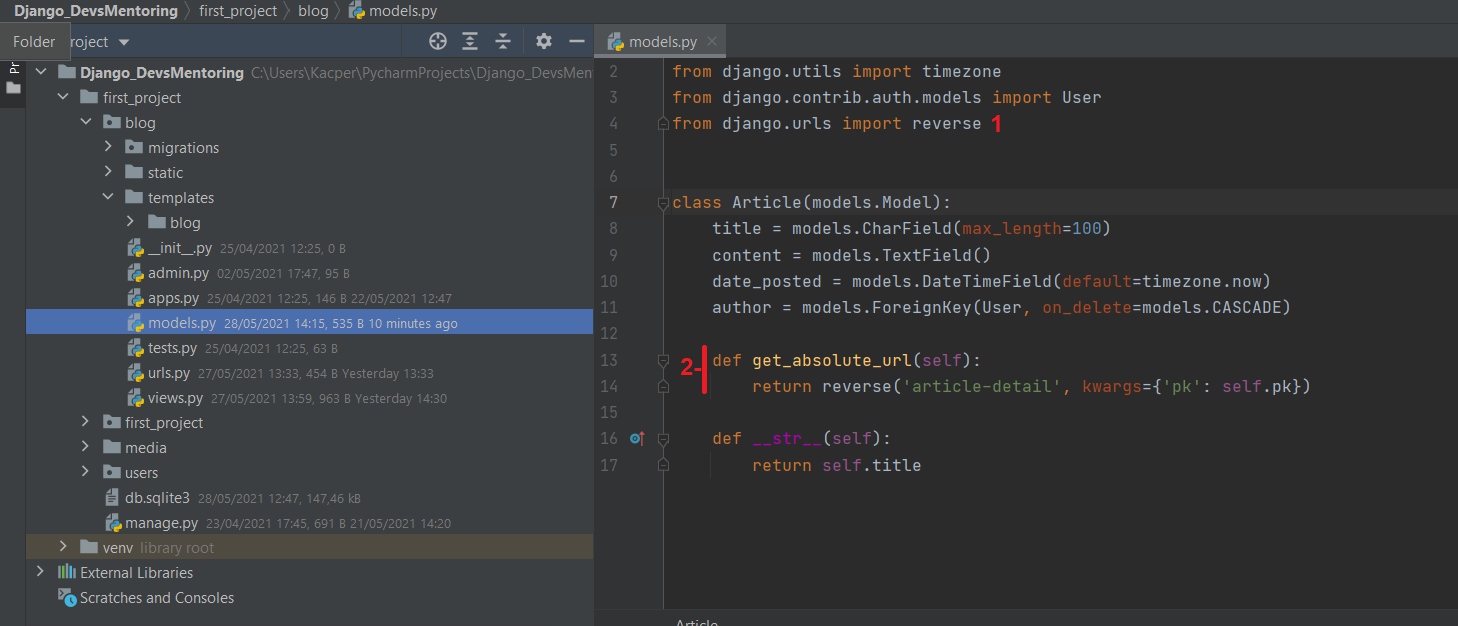


Okazuje się jednak, że gdy przejdziemy do podstrony /home/, to będzie na niej widniał przed chwilą utworzony artykuł:



Wniosek jest prosty - zapis nowego artykułu do bazy jest przeprowadzony prawidłowo, jedynie nie zaprogramowaliśmy jeszcze, gdzie ma być przekierowywany użytkownik po dodaniu posta. Zgodnie z komunikatem błędu, aby to zrealizować, dodamy metodą get\_absolute\_error wewnątrz klasy Article. Zadaniem takiej metody będzie zwracanie ścieżki URL, do której powinien zostać przeniesiony użytkownik po dodaniu nowego artykułu. Ścieżka ta będzie odpowiadała endpoint-owi nowo utworzonego artykułu.

W celu pobrania odpowiedniego adresu, użyjemy funkcji reverse zaimportowanej z django.urls. Jeżeli zastanawiasz się skąd ta nazwa, to jest ona powiązana z tym, co tak naprawdę robi ta funkcja: buduje ona bowiem adres URL do danego widoku na podstawie podanej przez użytkownika nazwy. Tak więc z racji, iż po dodaniu nowego artykułu, będziemy chcieli przejść do jego ArticleDetailView dostępnego pod adresem /article/<pk>, funkcję reverse zbudujemy następująco: reverse(‘article-detail’, kwargs={‘pk’: self.pk}).



from django.utils import timezone

from django.db import models

from django.contrib.auth.models import User

from django.urls import reverse

class Article(models.Model):

title = models.CharField(max\_length=100)

content = models.TextField()

date\_posted = models.DateTimeField(default=timezone.now)

author = models.ForeignKey(User, on\_delete=models.CASCADE)

def get\_absolute\_url(self):

return reverse('blog:article-detail', kwargs={'pk': self.pk})

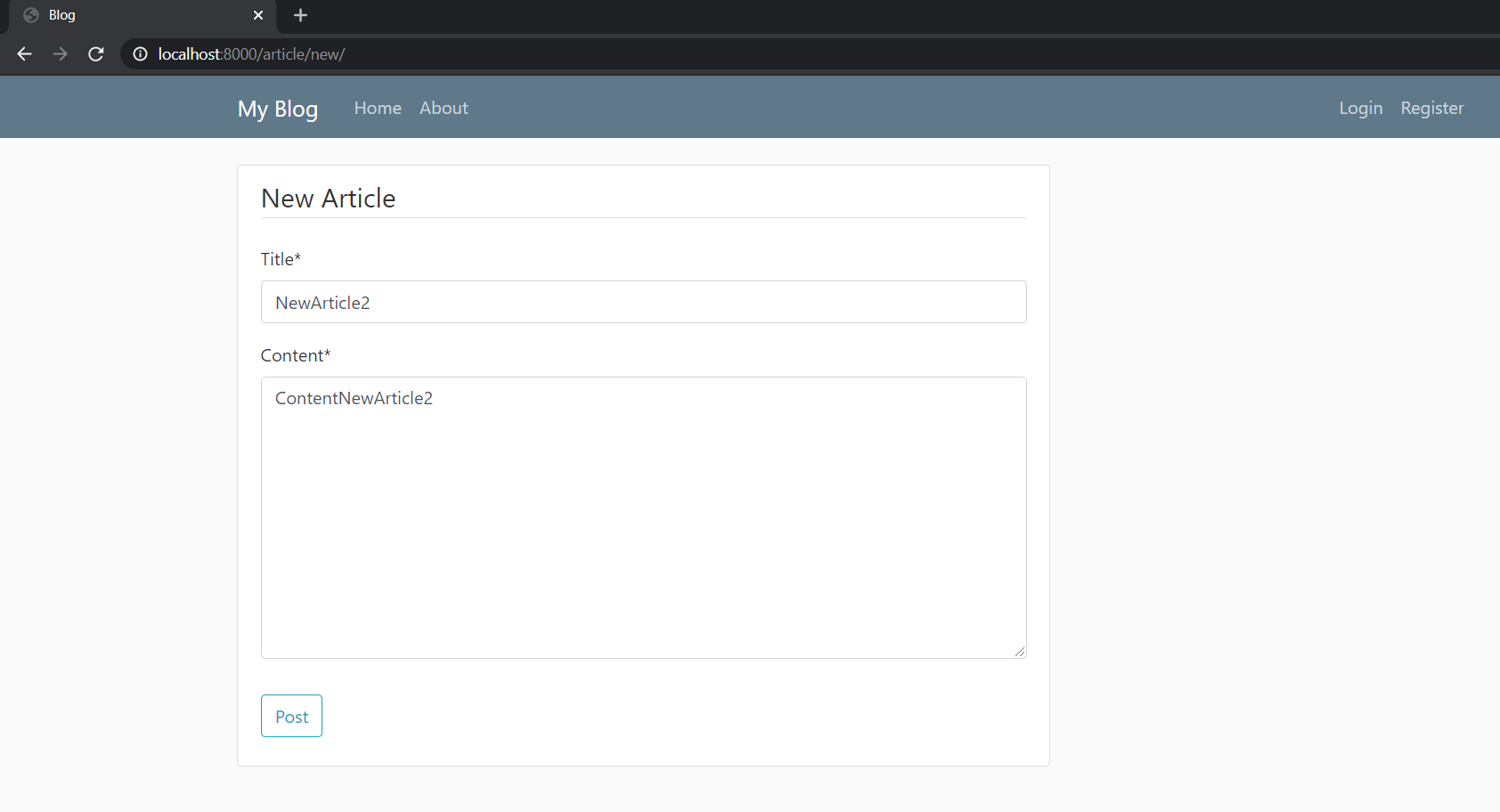
def \_\_str\_\_(self):

return self.title

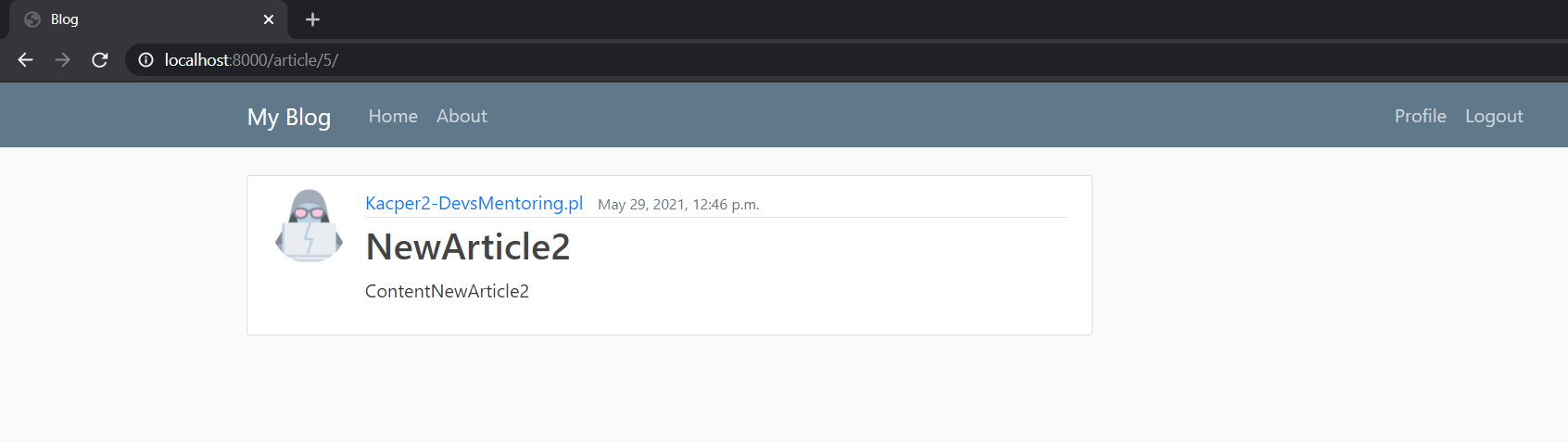
# Create your models here.

# Create your models here.

Po takich zmianach, powinniśmy już mieć możliwość sprawnego publikowania nowych artykułów. Sprawdźmy, jak teraz będzie wyglądał proces umieszczania nowego posta na stronie.



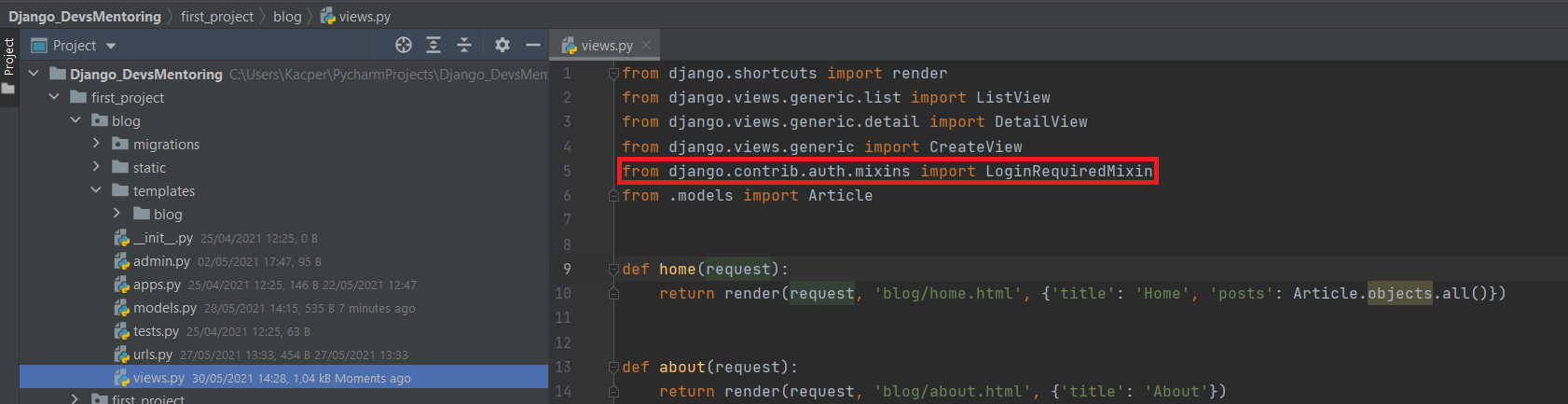
Po wciśnięciu przycisku Post, zostaniemy automatycznie przekierowani do utworzonego artykułu NewArticle2:



Wszystko działa zgodnie z oczekiwaniami! Musimy jednak pochylić się na jeszcze jednym dość istotnym szczególe. Co w momencie, gdy użytkownik będzie chciał przejść na endpoint /article/new, nie będąc zalogowanym? Otóż zostanie on prawidłowo przekierowany do formularza, gdzie będzie mógł wypełnić informacje o publikowanym artykule. Problem jednak pojawi się w momencie, gdy naciśnie przycisk Post. Aplikacja będzie próbowała przypisać zalogowanego użytkownika do utworzonego posta, ale przecież nie będzie to możliwe z racji, iż żaden z użytkowników nie jest aktualnie zalogowany. Oczywiste więc jest, że próba takiego dodania nowego artykułu zakończy się błędem. Dlatego, aby się przed tym uchronić, zmodyfikujemy Naszą aplikację w taki sposób, że próba przejścia niezalogowanego użytkownika do strony umożliwiającej utworzenie nowego artykułu, zakończy się automatycznym przekierowaniem go do panelu logowania.

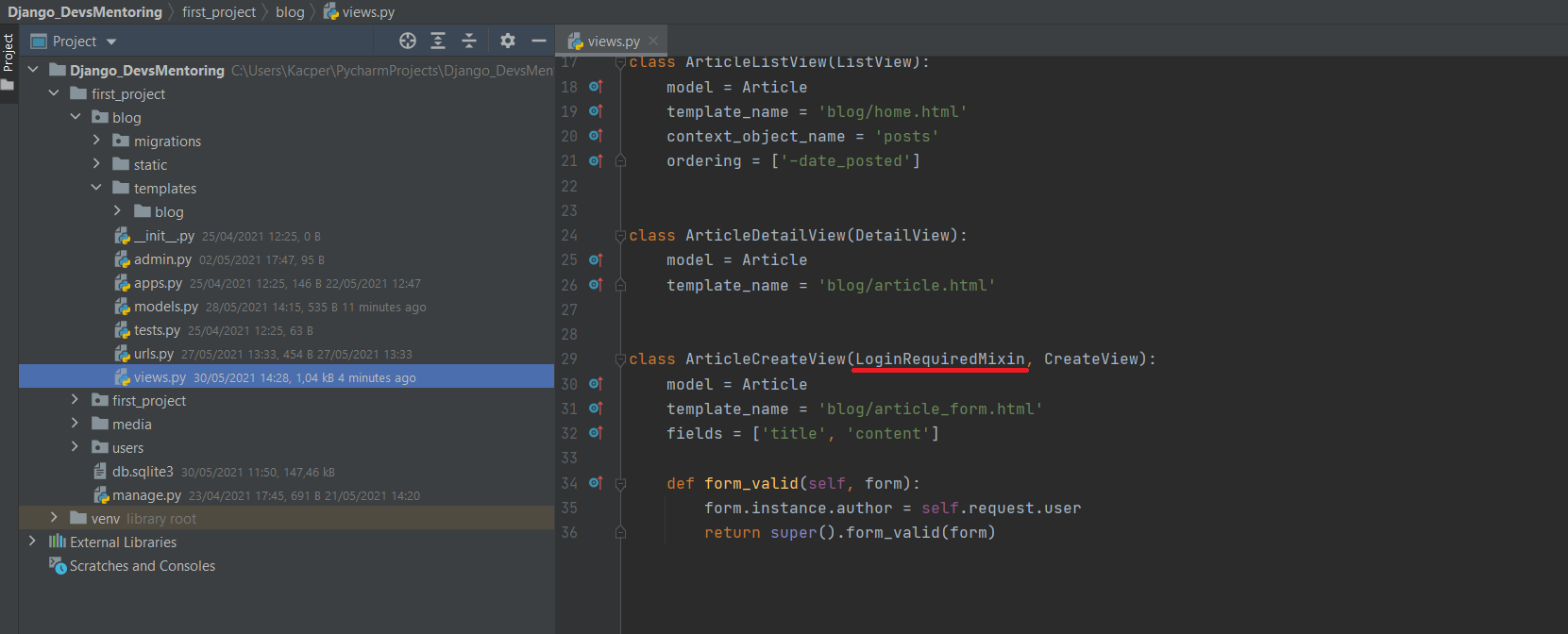
Aby to zrealizować, wykorzystamy wbudowany do bliblioteki Django moduł mixins - który dostarcza, tzw. mixin-y nazywane również “ukrytymi funkcjonalnościami’ - czyli takimi, które są wykonywane wraz z pewnymi głównymi realizacjami kodu. Możemy zatem zaimportować LoginRequiredMixin umożliwiający, przed wykonaniem pewnej metody, sprawdzenie, czy użytkownik jest zalogowany.

Tak więc w pierwszej kolejności zaimportujemy z biblioteki django.contrib.auth.mixins klasę LoginRequiredMixins, a następnie przekażemy ją do listy rodziców widoku odpowiedzialnego za generowanie formularza artykułu - ArticleCreateView.



*Zaimportowanie klasy LoginRequiredMixin*

*from django.contrib.auth.mixins import LoginRequiredMixins*



*Dodanie LoginRequiredMixin do listy parametrów*

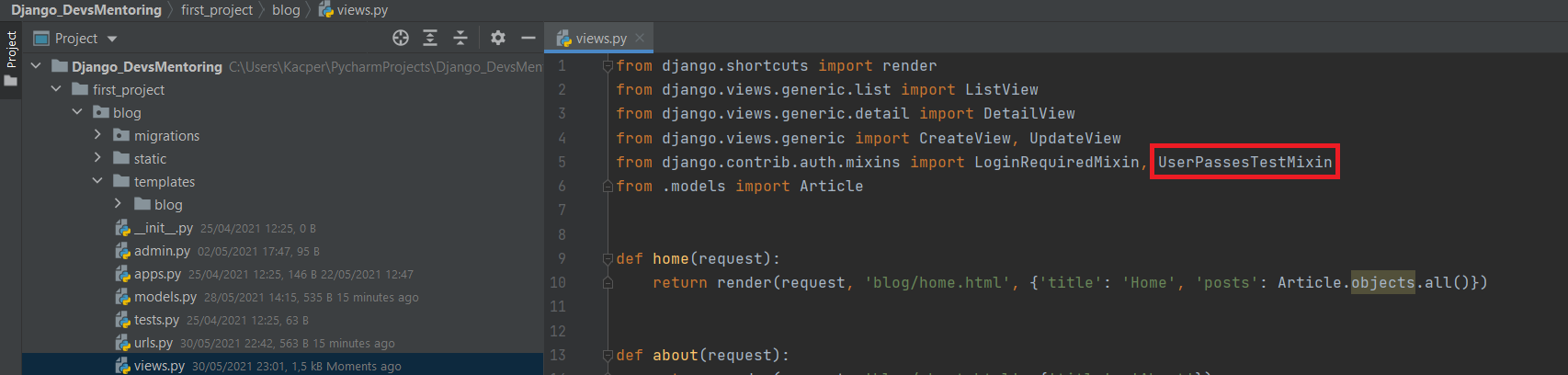
Ciężko mi będzie zaprezentować w szkoleniu uzyskany po powyższych zmianach efekt, ale możesz go sprawdzić na własnej skórze. Jeżeli więc, nie będąc zalogowanym, spróbujesz przejść pod endpoint /article/new - powinieneś zostać automatycznie przekierowany do strony logowania. Dopiero zalogowany użytkownik będzie mógł stworzyć i opublikować określony artykuł.

**Dodanie aktualizowania artykułów**

W następnym kroku zajmiemy się wprowadzeniem możliwości aktualizowania danych artykułów przez użytkowników. Możliwość edytowania dowolnej publikacji będzie możliwa do przeprowadzenia spod adresu /article/update/<int:pk>. Jednak zanim przystąpimy do implementacji, musimy zwrócić uwagę na pewien szczegół. Otóż edycję danego artykułu będziemy chcieli umożliwić tylko temu użytkownikowi, który jest autorem edytowanego posta. Aby zrealizować to założenie, wykorzystamy kolejny mixin - UserPassesTestMixin. Mixin ten umożliwia określenie, jaki warunek musi zostać spełniony, aby użytkownik mógł zostać przekierowany do danego endpointa.

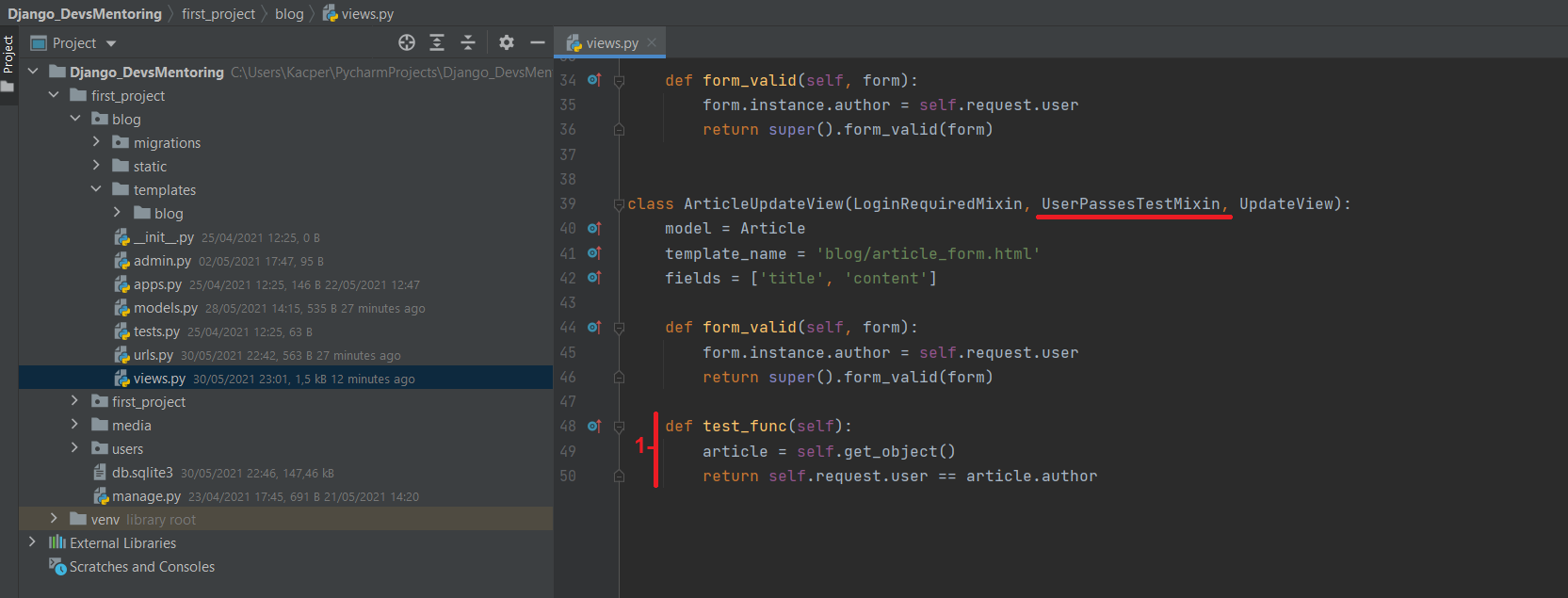
Zacznijmy od zaimportowania odpowiednich modułów. W pliku blog/views.py dodamy import klasy UserPassesTestMixin.

Następnie utworzymy nową klasę ArticleUpdateView (która będzie tak naprawdę copy-paste klasy CreateView) z pewną subtelną zmianą. Zmiana ta polegać będzie na dodaniu zaimportowanego mixin-a do listy rodziców wspomnianej klasy ArticleUpdateView. Z racji, iż UserPassesTestMixin służy do tworzenia dowolnych “warunków przejścia” do danego endpointa, konieczne będzie również zaimplementowanie wewnątrz ArticleUpdateView funkcji test\_func, w której zawrzemy sprawdzenie, czy aktualnie zalogowany użytkownik jest również tym, który utworzył wybrany artykuł. Jeśli tak, to zezwolimy mu na edytowanie, w innym natomiast przypadku wyświetlimy błąd 403.



*Zaimportowanie odpowiedniej klasy*

from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin



*Logika weryfikująca, czy możliwa jest edycja artykułu*

from django.shortcuts import render

from django.http import HttpResponse

from .models import Article

from django.views.generic.list import ListView

from django.views.generic.detail import DetailView

from django.views.generic import CreateView

from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin

def home(request):

return render(request, 'blog/home.html', {'title': 'Home', 'posts': Article.objects.all()})

# return render(request, 'blog/home.html', {'title': 'Home', 'posts': POSTS})

def about(request):

return render(request, 'blog/about.html', {'title': 'About'})

# Create your views here.

class ArticleListView(ListView):

model = Article

template\_name = 'blog/home.html'

context\_object\_name = 'posts'

ordering = ['-date\_posted']

class ArticleDetailView(DetailView):

model = Article

template\_name = 'blog/article.html'

class ArticleCreateView(LoginRequiredMixin, CreateView):

model = Article

template\_name = 'blog/article\_form.html'

fields = ['title', 'content']

def form\_valid(self, form):

form.instance.author = self.request.user

return super().form\_valid(form)

class ArticleUpdateView(LoginRequiredMixin, UserPassesTestMixin, UpdateView):

model = Articletemplate\_name = 'blog/article\_form.html'

fields = ['title', 'content']

def form\_valid(self, form):

form.instance.author = self.request.user

return super().form\_valid(form)

def test\_func(self):

article = self.get\_object()

return self.request.user == article.author

Pochylmy się na chwilę nad metodą test\_func. Tak jak wspomniałem - to w niej przechowywana jest logika sprawdzająca, czy użytkownik może zostać dopuszczony do edycji danej strony. Linia 49 to pobranie obiektu artykułu, do którego chcemy się w danym momencie przenieść, a linia 50 to konkretne już sprawdzenie, czy autor tego artykułu jest tą samą osobą, co aktualnie zalogowany użytkownik.

Zauważ, że linię 50 moglibyśmy zapisać mniej optymalnie (wystrzegaj się takiego rozwiązania i w miarę możliwość usprawniaj kod):

if self.request.user == article.author:

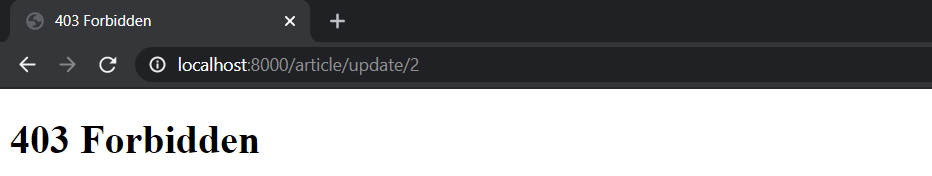
return True

return False

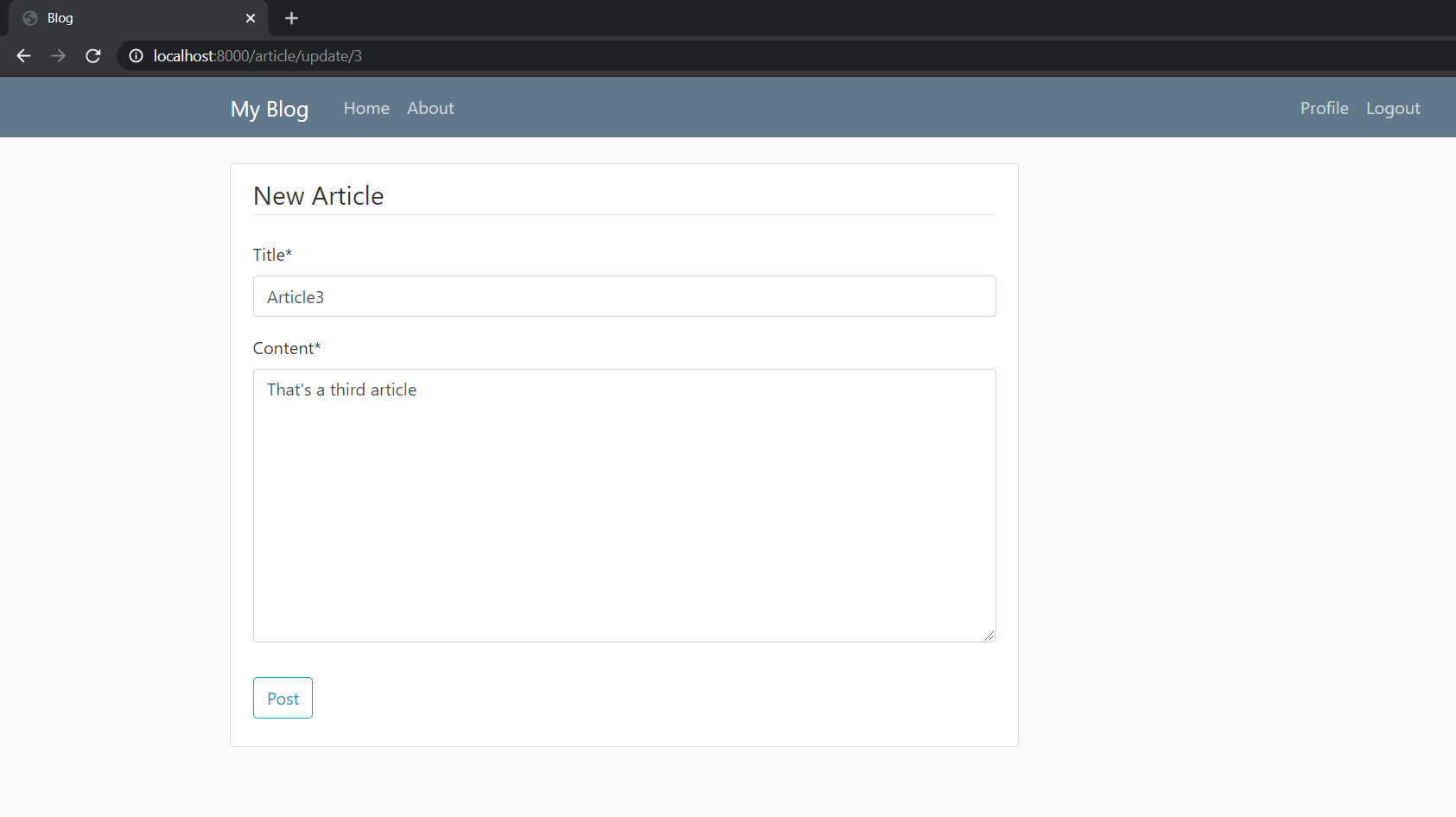
Będąc zalogowanym na konto Kacper-DevsMentoring.pl, spróbujemy edytować artykuł utworzony przez konto Test4 (jest to Article2 o pk = 2).



Tak więc po wpisaniu w pasek URL adresu localhost:8000/article/update/2, powinniśmy otrzymać błąd o braku zezwolenia na edycję artykułu.

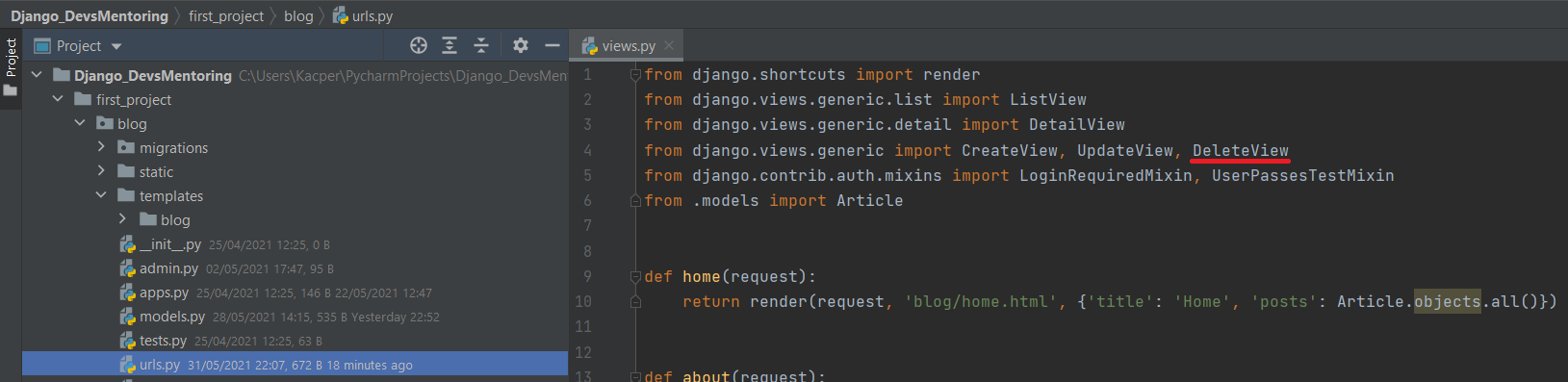


Natomiast, gdy będziemy chcieli dokonać edycji na artykule przypisanym już do właściwego konta - autora będącego zalogowanym użytkownikiem, aplikacja umożliwi tę operację.

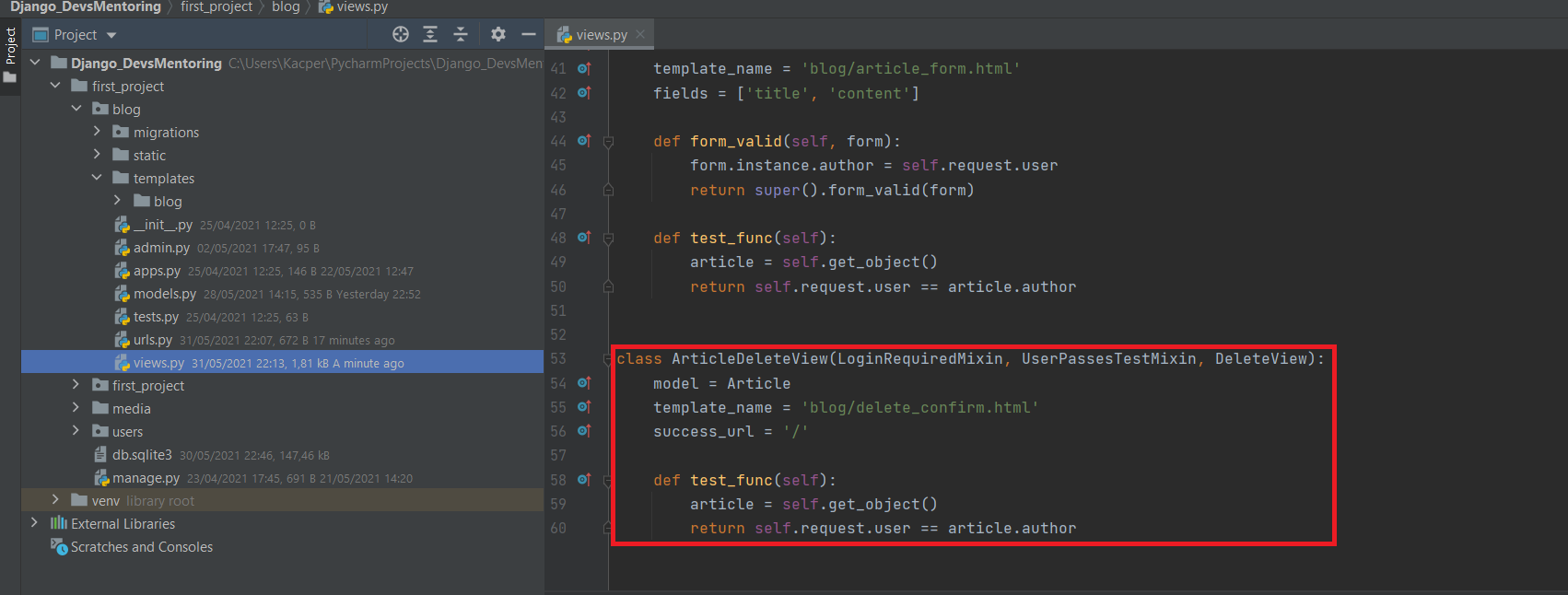


**Usuwanie artykułu**

Jak dotąd zajęliśmy się dodaniem możliwości tworzenia i edytowania artykułów. Teraz nadszedł czas na umożliwienie usuwania poszczególnych postów. Z procesem usuwania powiązany będzie endpoint /article/delete/<pk> oraz widok ArticleDeleteView, w którym będziemy realizować logikę działania. Klasa ta nie będzie się mocno różniła od utworzonej już ArticleUpdateView - również będziemy musieli w niej zdefiniować pola takie jak: model, template\_name oraz succes\_url (który za chwilę omówię) wraz z metodą weryfikacji, czy zalogowany użytkownik jest autorem posta i może go usunąć - test\_func. Mam nadzieję, że nie muszę również przypominać, iż konieczne będzie również zaimportowanie klasy DeleteView z django.views.generic.



*Potrzebny import*



*Nowo utworzona klasa*

class ArticleDeleteView(LoginRequiredMixin, UserPassesTestMixin, DeleteView):

model = Article

template\_name = 'blog/delete\_confirm.html'

success\_url = '/'

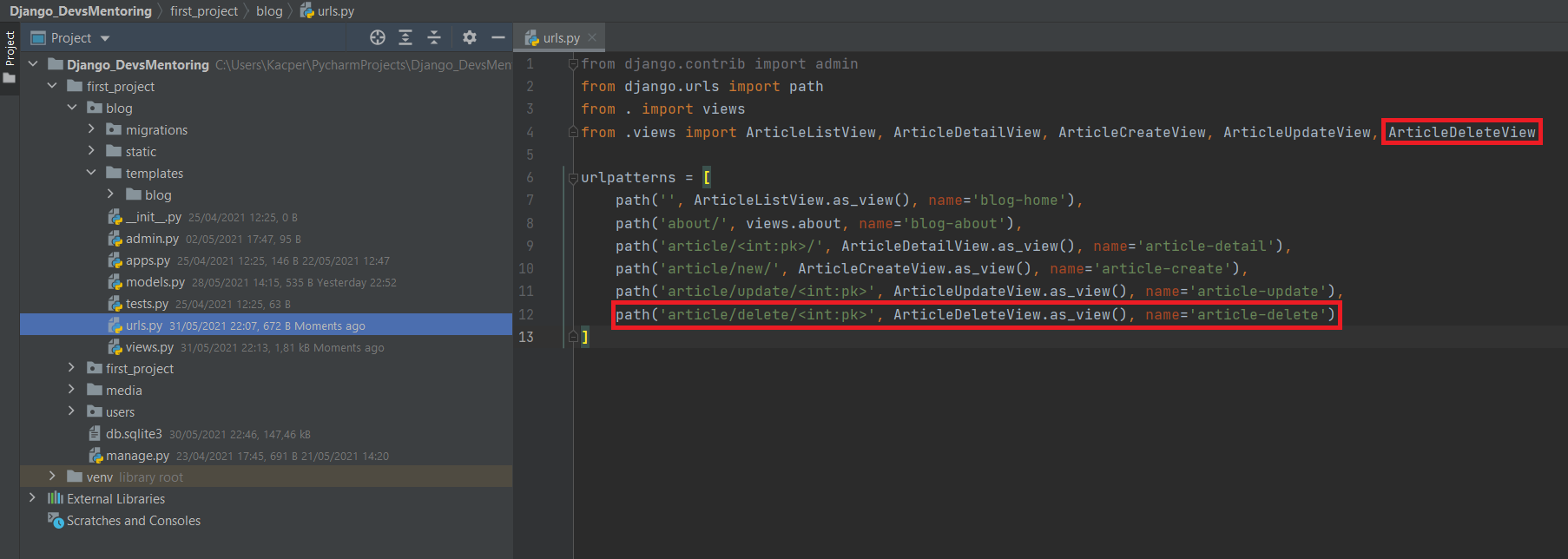
def test\_func(self):

article = self.get\_object()

return self.request.user == article.author

Zwróć uwagę, że pole template\_name odnosi się do jeszcze nieistniejącego szablonu delete\_confirm.html - będziemy musieli go zatem utworzyć i odpowiednio zakodować. Tak jak obiecałem, należą Ci się również wyjaśnienia za co odpowiada pole success\_url. Otóż definiujemy w nim ścieżkę, do której ma zostać przekierowany użytkownik po wykonaniu pewnych operacji w widoku ArticleDeleteView. W gwoli ścisłości - operacja te to potwierdzenie, czy użytkownik jest pewny, że chce usunąć określony artykuł.

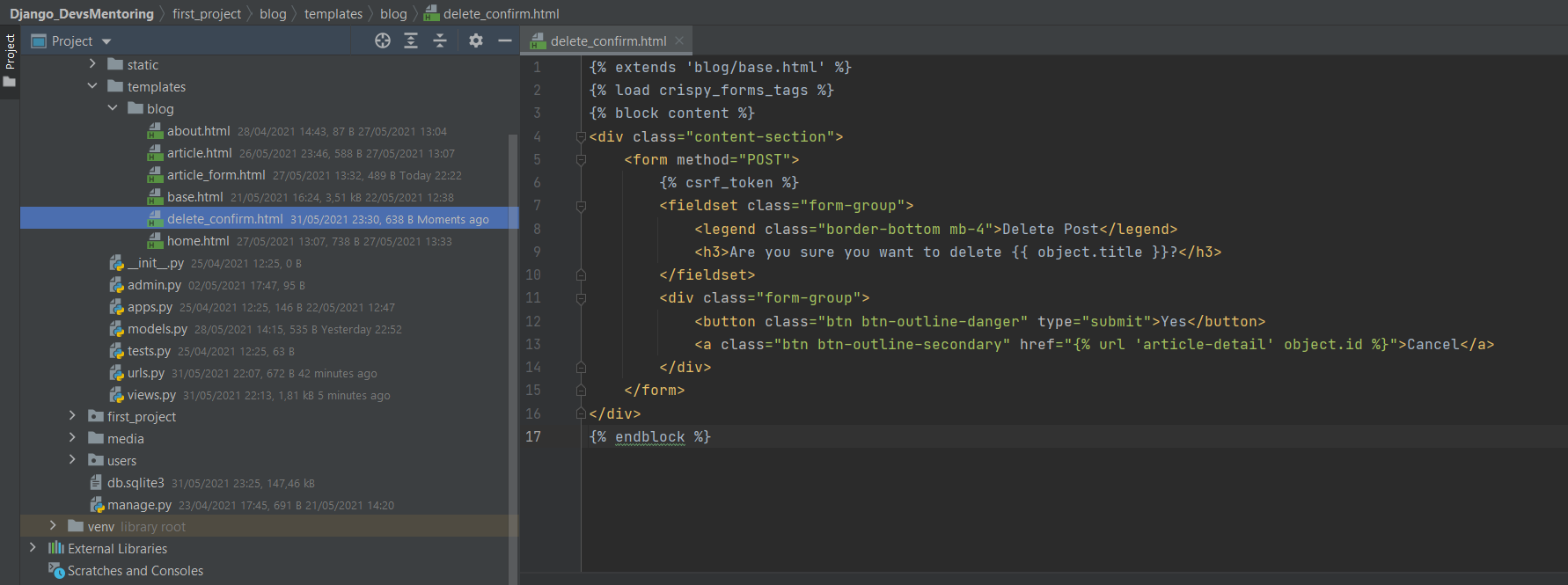
Aby zapewnić prawidłowe działanie aplikacji, musimy jeszcze dodać odpowiednie referencje w pliku blog/urls.py. Nie powinna to być jednak dla Ciebie nowość, ponieważ podobne kroki przeprowadzaliśmy również przy dodawaniu widoków ArticleUpdateView, ArticleCreateView etc.



*Plik blog/urls.py*

path("article/delete/<int:pk>", ArticleDeleteView.as\_view(), name='article-delete'),

Następnie utworzymy plik deleteconfirm.html i zakodujemy go w następujący sposób:



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

{% load crispy\_forms\_tags %}

{% block content %}

<div class="content-section">

<form method="POST">

{% csrf\_token %}

<fieldset class="form-group">

<legend class="border-bottom mb-4">Delete post</legend>

<h3>Are you sure you want to delete {{ object.title }}</h3>

</fieldset>

<div class="form-group">

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

<a class="btn btn-outline-secondary" href="{% url 'blog:article-detail' object.id %}">Cancel</a>

</div>

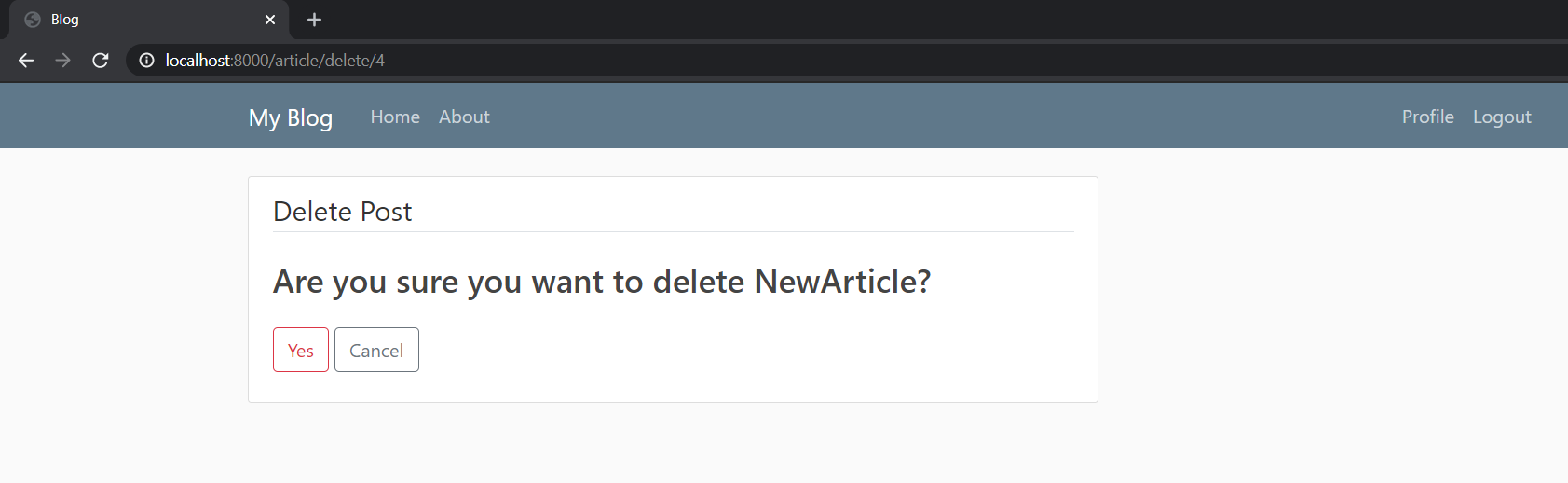
</form>

</div>

{% endblock %}

Zawartość przekopiowałem z pliku blog/templates/blog/article\_form.html i nieco ją zmodyfikowałem. Zwróć szczególnie uwagę na użyte różne klasy Bootstrapa (btn-outline-danger, btn-outline-secondary) - tak, aby odpowiednio dostosować wygląd przycisków. Zauważ również, jaką wartość przyjmuje atrybut href znajdujący się w znaczniku <a> dla Cancel. Jest to {% url ‘article-detail’ object.id %}, bowiem chcemy, aby anulowanie operacji prowadziło z powrotem do artykułu, który zdecydowaliśmy się usunąć.

Przy próbie usunięcia artykułu o id 4, efekt wprowadzonych zmian będzie następujący:

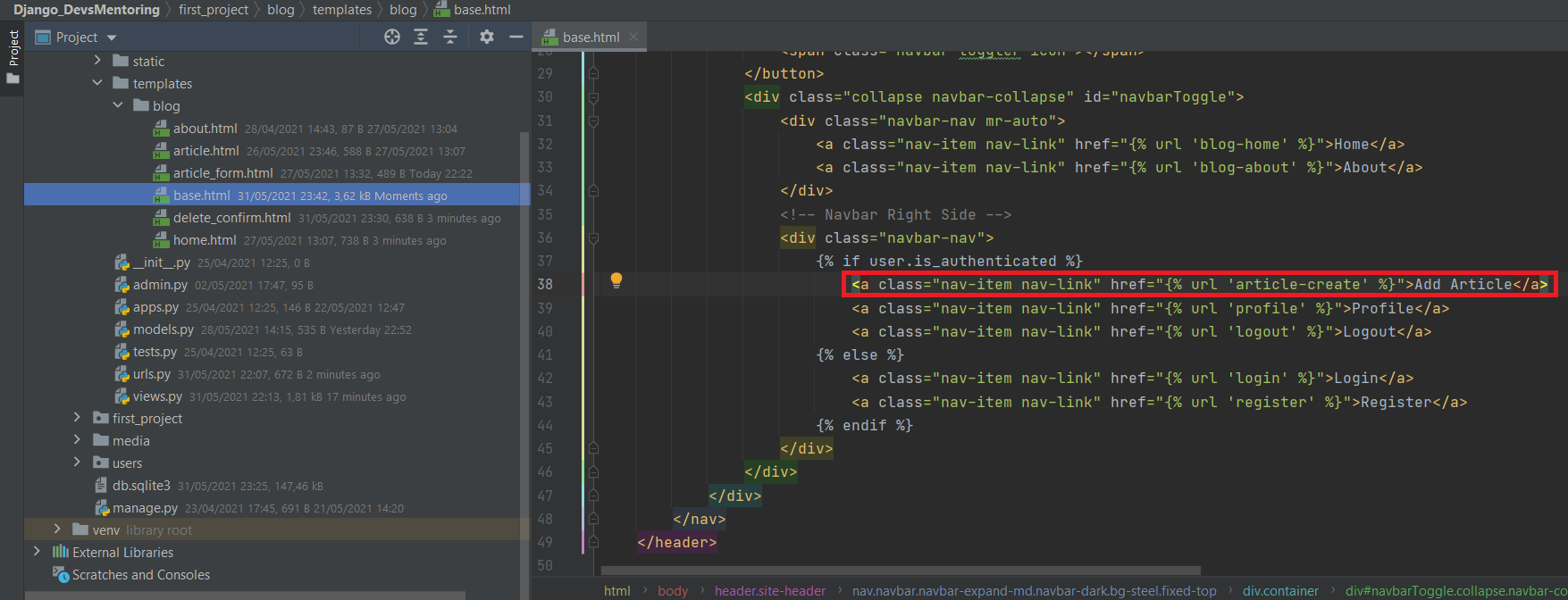


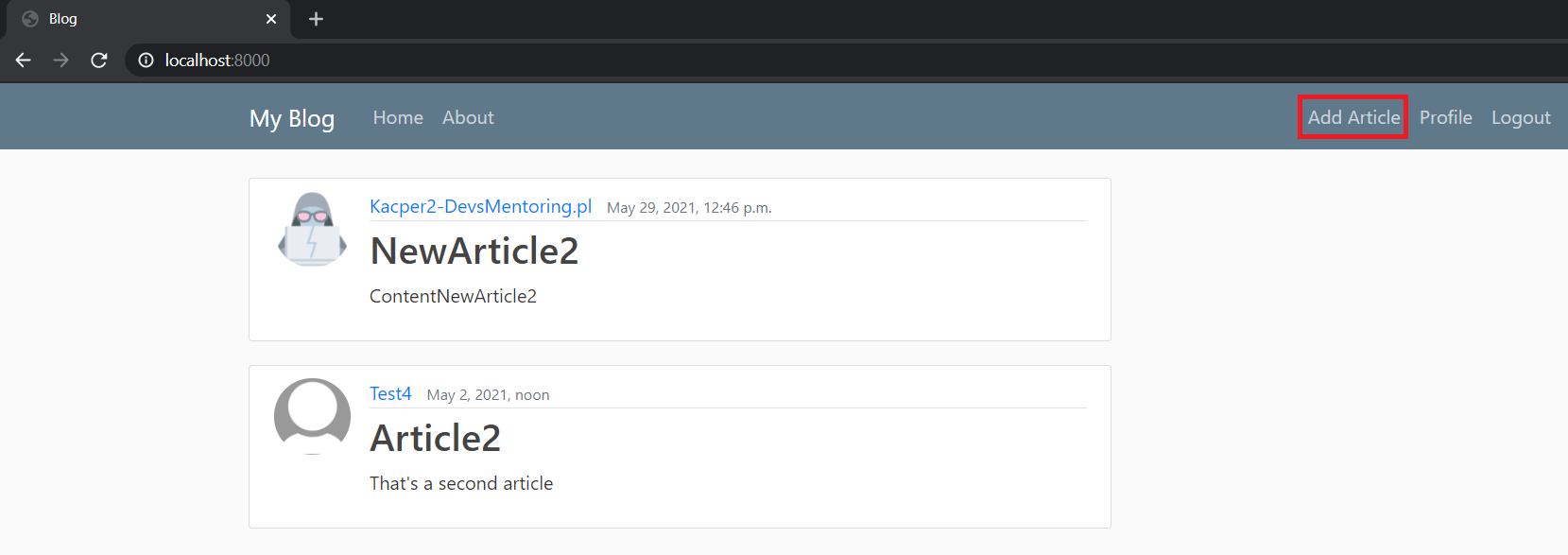
Gdy klikniemy przycisk Yes, zostaniemy przekierowani do strony głównej serwisu (stąd konieczność umieszczenia pola success\_url wewnątrz klasy ArticleDeleteView). Cancel natomiast spowoduje poprawne anulowanie operacji i powrócenie do artykułu, który chcemy usunąć.

**Ostateczne poprawki**

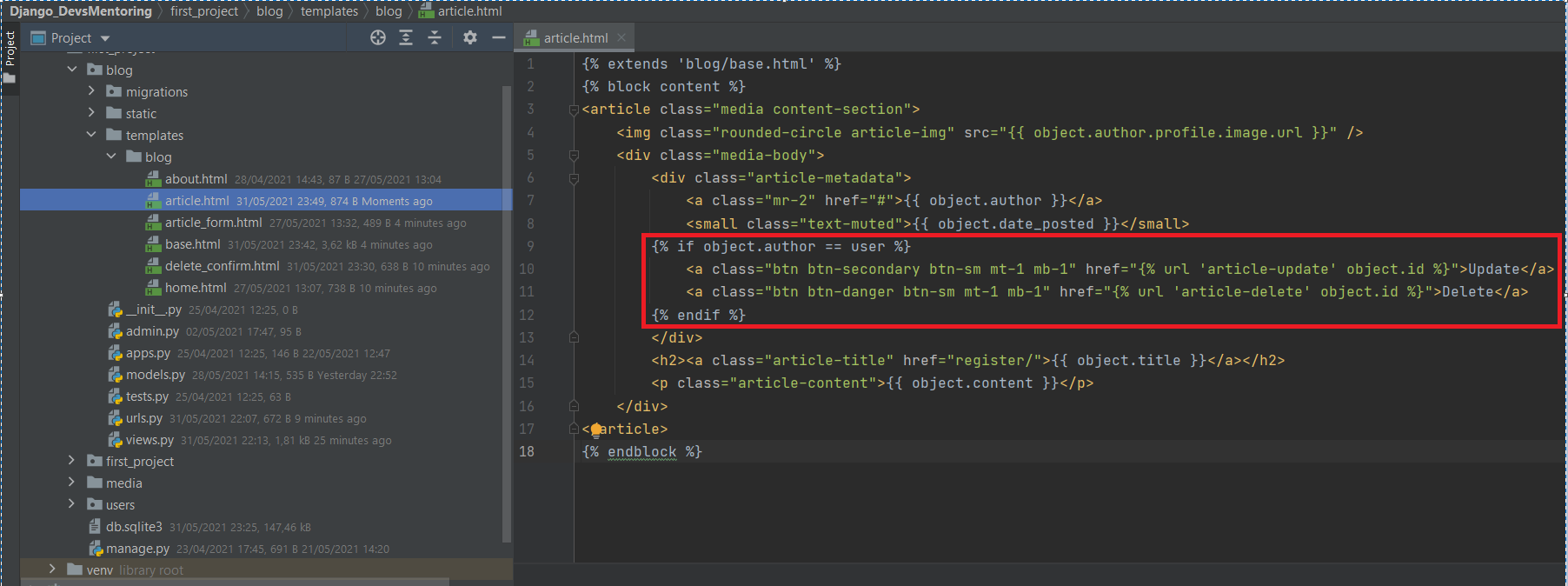
Gruntowną funkcjonalność aplikacji mamy już zaimplementowaną! Pozostało jedynie dodać pojedyncze drobne zmiany postaci, np. przycisk do tworzenia nowego artykułu, a z poziomu konkretnego już artykułu przyciski do jego usunięcia i edycji.

Zaczniemy od dodania opcji Add Post, którą będzie mógł wybrać zalogowany użytkownik w celu dodania nowego posta. Dokonamy tego w pliku blog/templates/blog/base.html.

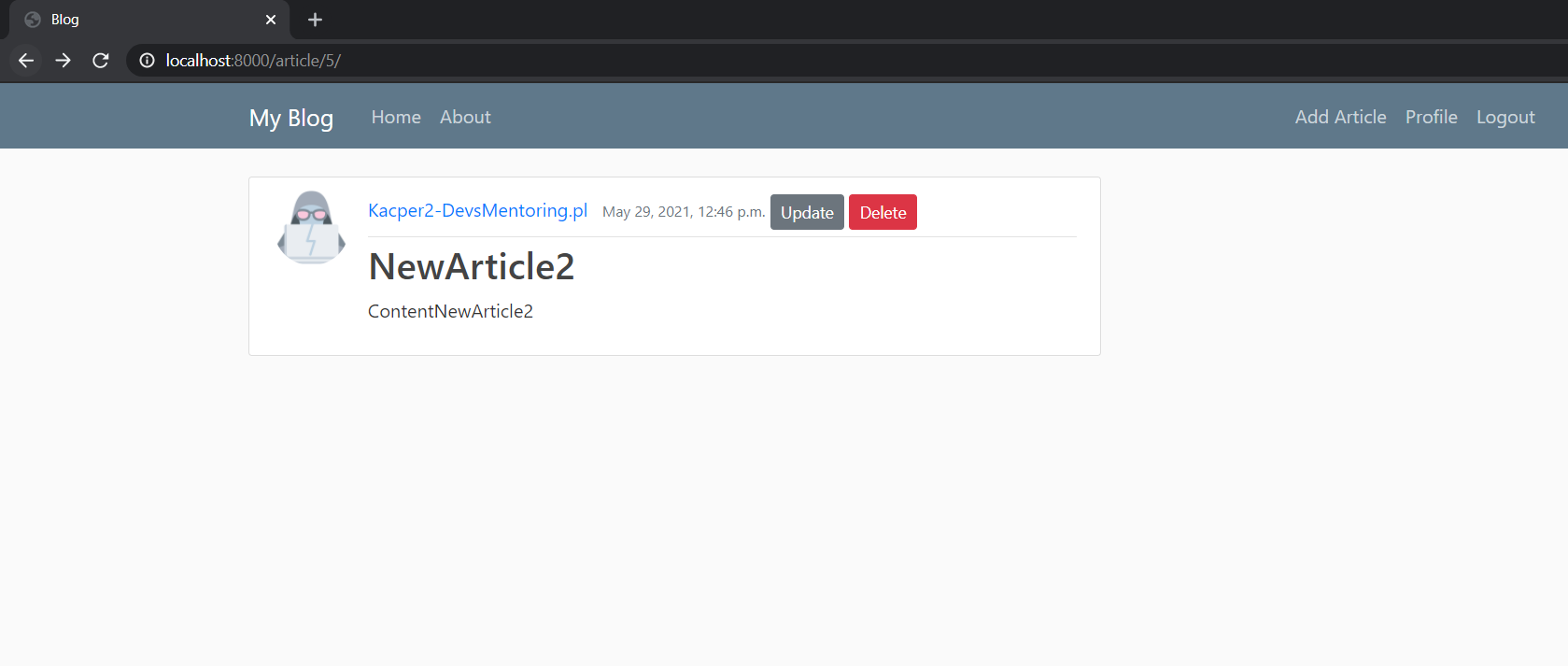




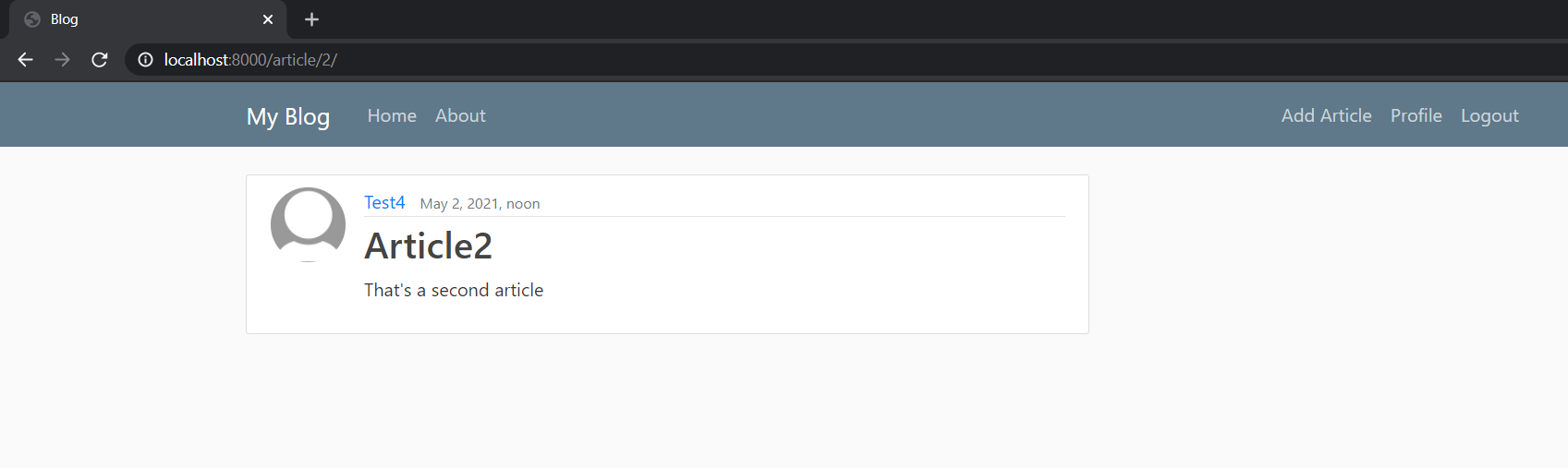
Następnie dodamy przyciski Update i Delete, z których będzie można skorzystać w ramach pojedynczego artykułu. W tym celu dodamy następujący fragment kodu wewnątrz pliku blog/templates/blog/article.html:



Powyższy kod sprawdza, czy autor artykułu jest aktualnie zalogowanym użytkownikiem. Jeżeli tak, to wyświetlamy opcje aktualizacji i usunięcia artykułu. Konieczne jest takie rozwiązanie, ponieważ nie chcielibśmy przecież, aby użytkownik inny niż autor artykułu miał możliwość edycji artykułu. Otrzymamy następujący efekt:



*Artykuł, którego autorem jest zalogowany użytkownik*



*Artykuł, którego autorem nie jest zalogowany użytkownik*

**Zakończenie**

Dobrze, już dobrze. Wiem, że szkolenie nieco się przedłużyło, ale to dlatego, iż zrealizowaliśmy w nim solidną porcję nowych funkcjonalności. Mam nadzieję, że wiesz już, do czego służą widoki oparte na klasach, mixiny, a także jak mniej więcej obsługiwać logikę działania bardziej złożonych serwisów. Nie owijając w bawełnę, możemy śmiało przechodzić do następnych szkoleń. A w nich jedne z ostatnich poprawek w aplikacji!