

Chatter – Aplikacja internetowa

Opis: Zastosowanie Pythona i frameworka Django (wersja 1.6.5) do stworzenia aplikacji internetowej Chatter; prostego czata, w którym zarejestrowani użytkownicy będą mogli wymieniać się krótkimi wiadomościami.

Autorzy: Tomasz Nowacki, Robert Bednarz

Czas realizacji: 90 min

Poziom trudności: Poziom 3

Spis treści

| | |
|--|-----------|
| Chatter – Aplikacja internetowa..... | 1 |
| I. Projekt i aplikacja..... | 2 |
| Terminal I.1..... | 2 |
| Kod I.1..... | 2 |
| II. Model – Widok – Kontroler..... | 3 |
| III. Model danych i baza..... | 3 |
| Kod III.1..... | 3 |
| Terminal III.1..... | 4 |
| IV. Panel administracyjny..... | 4 |
| Kod IV.1..... | 4 |
| V. Strona główna – widoki i szablony..... | 5 |
| Kod V.1..... | 6 |
| Kod V.2..... | 6 |
| Terminal V.1..... | 6 |
| Kod V.3..... | 6 |
| VI. Logowanie użytkowników..... | 7 |
| Kod VI.1..... | 7 |
| Kod VI.2..... | 8 |
| Kod VI.3..... | 8 |
| Kod VI.4..... | 8 |
| VII. Dodawanie i wyświetlanie wiadomości..... | 9 |
| Kod VII.1..... | 9 |
| Kod VII.2..... | 10 |
| Kod VII.3..... | 10 |
| Kod VII.4..... | 10 |
| JAK TO DZIAŁA..... | 11 |
| VIII. Rejestrowanie użytkowników..... | 11 |
| Kod VIII.1..... | 11 |
| Kod VIII.2..... | 12 |
| Kod VIII.3..... | 12 |
| Kod VIII.4..... | 12 |
| JAK TO DZIAŁA..... | 12 |
| IX. Wylogowywanie użytkowników..... | 13 |
| Kod IX.1..... | 13 |
| Kod IX.2..... | 13 |
| Kod IX.3..... | 13 |

I. Projekt i aplikacja

Tworzymy nowy projekt Django, a następnie uruchamiamy lokalny serwer, który pozwoli śledzić postęp pracy. W katalogu domowym wydajemy polecenia w terminalu:

Terminal I.1

```
~ $ django-admin.py startproject chatter
~ $ cd chatter
~/chatter $ python manage.py runserver 127.0.0.1:8080
```

Powstanie katalog projektu **chatter** i aplikacja o nazwie **chatter**¹. Pod adresem **127.0.0.1:8080** w przeglądarce zobaczymy stronę powitalną².

It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Teraz zmodyfikujemy ustawienia projektu, aby korzystał z polskiej wersji językowej oraz lokalnych ustawień daty i czasu. Musimy również zarejestrować naszą aplikację w projekcie. W pliku **setting.py** zmieniamy następujące linie:

Kod I.1

```
# chatter/chatter/settnigs.py

# rejestrujemy aplikacje
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'chatter', # nasza aplikacja
)

LANGUAGE_CODE = 'pl' # ustawienia jezyka

TIME_ZONE = 'Europe/Warsaw' # ustawienia daty i czasu
```

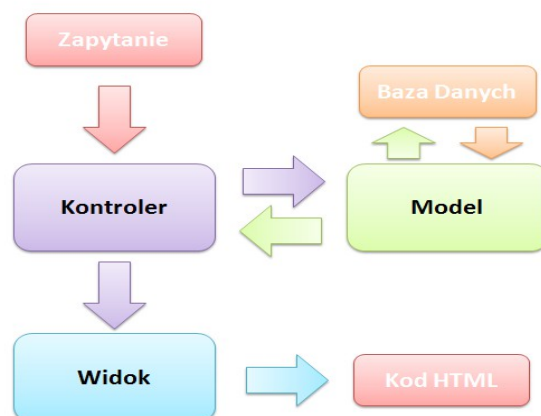
1 Jeden projekt może zawierać wiele aplikacji zapisywanych w osobnych podkatalogach katalogu projektu.

2 Lokalny serwer deweloperski można zatrzymać za pomocą skrótu Ctrl+C.

II. Model – Widok – Kontroler

W projektowaniu aplikacji internetowych za pomocą Django odwołujemy się do wzorca M(odel)V(iew)C(ontroller), czyli Model–Widok–Kontroler³, co pozwala na oddzielenie danych od ich prezentacji oraz logiki aplikacji. Funkcje kolejnych elementów są następujące:

- ✓ Modele – w Django reprezentują źródło informacji, są to klasy Pythona, które zawierają pola, właściwości i zachowania danych, odwzorowują pojedyncze tabele w bazie danych⁴. Definiowane są w pliku **models.py**.
- ✓ Widoki – w Django są to funkcje Pythona, które na podstawie żądań www (dla danych adresów URL) zwracają odpowiedź w postaci kodu HTML generowanego w szablonach (templates), przekierowania, dokumentu XML czy obrazka. Definiowane są w pliku **views.py**.
- ✓ Kontroler – to mechanizm kierujący kolejne żądania do odpowiednich widoków na podstawie konfiguracji adresów URL zawartej w pliku **urls.py**.



III. Model danych i baza

Model, jak zostało powiedziane, jest klasą Pythona opisującą dane naszej aplikacji, czyli wiadomości. Instancje tej klasy będą konkretnymi wiadomościami napisanymi przez użytkowników systemu. Każda wiadomość będzie zawierała treść, datę dodania oraz autora wiadomości (użytkownika).

W katalogu **chatter/chatter** w pliku **models.py** wpisujemy:

Kod III.1

```
# -*- coding: utf-8 -*-  
  
# chatter/chatter/models.py  
  
from django.db import models
```

³ Twórcy Django traktują jednak ten wzorzec elastycznie, mówiąc że ich framework wykorzystuje wzorzec MTV, czyli model (model), szablon (template), widok (view).

⁴ Takie odwzorowanie nosi nazwę mapowania obiektowo-relacyjnego (ORM). ORM odwzorowuje strukturę bazy na obiekty Pythona.

```
from django.contrib.auth.models import User

class Message(models.Model):
    """Klasa reprezentująca wiadomość w systemie."""
    text = models.CharField(u'wiadomość', max_length=250)
    pub_date = models.DateTimeField(u'data publikacji')
    user = models.ForeignKey(User)

    class Meta: # to jest klasa w klasie
        verbose_name = u'wiadomość' # nazwa modelu w języku polskim
        verbose_name_plural = u'wiadomości' # nazwa modelu w l. mnogiej
        ordering = ['pub_date'] # porządkowanie danych względem daty
```

Po skonfigurowaniu projektu i zdefiniowaniu modelu danych możemy utworzyć bazę danych⁵ dla naszej aplikacji, czyli wszystkie potrzebne tabele. Podczas tworzenia bazy Django pyta o nazwę, email i hasło administratora. Podajemy te dane po wydaniu w katalogu projektu w terminalu polecenia:

Terminal III.1

```
~/chatter $ python manage.py syncdb
```

IV. Panel administracyjny

Django pozwala szybko utworzyć panel administratora dla naszego projektu. Rejestrujemy więc model danych jako element panelu w nowo utworzonym pliku **admin.py** w katalogu **chatter/chatter**:

Kod IV.1

```
# chatter/chatter/admin.py

from django.contrib import admin
from chatter.models import Message # importujemy nasz model

admin.site.register(Message) # rejestrujemy nasz model w panelu administracyjnym
```

Po ewentualnym ponownym uruchomieniu serwera wchodzimy na adres **127.0.0.1:8080/admin/**. Otrzymamy dostęp do panelu administracyjnego, w którym możemy dodawać nowych użytkowników i wiadomości⁶.

5 Domyślnie Django korzysta z bazy SQLite, która przechowywana jest w jednym pliku **db.sqlite3** w katalogu aplikacji

6 Bezpieczna aplikacja powinna dysponować osobnym mechanizmem rejestracji użytkowników i dodawania wiadomości, tak by nie trzeba było udostępniać panelu administracyjnego osobom postronnym.

Administracja Django Witaj, root. Z

Administracja stroną

| Auth | |
|-------------|---|
| Grupy | + Dodaj Zmień |
| Użytkownicy | + Dodaj Zmień |

| Chatter | |
|------------|---|
| Wiadomości | + Dodaj Zmień |

Ostatnie akcje
Moje akcje
[Message](#)
[Message](#)
[+ Message](#)
[Message](#)

Początek > Chatter > Wiadomości > Dodaj wiadomość

Dodaj wiadomość

Wiadomość:

Data publikacji:
Data: [Dzisiaj](#)
Czas: [Teraz](#)

User: [+](#)

[Zapisz i dodaj nowe](#) [Zapisz i kończ](#)

V. Strona główna – widoki i szablony

Dodawanie stron w Django polega na tworzeniu **widoków**, czyli funkcji Pythona powiązanych z określonymi **adresami url**. Widoki najczęściej zwracają kod HTML wyrenderowany na podstawie **szablonów**, do których możemy przekazywać dodatkowe dane⁷, np. z bazy. Dla przejrzystości przyjęto, że w katalogu aplikacji (**chatter/chatter**):

1. plik **views.py** zawiera definicję widoków, w tym wywołania szablonów,
2. plik **url.py** zawiera reguły łączące widoki z adresami url,
3. w katalogu **chatter/chatter/templates/chatter** zapisujemy **szablony** (templatki) pod nazwami określonymi w wywołujących je widokach, np. **index.html**.

Aby utworzyć stronę główną, stworzymy pierwszy **widok**, czyli funkcję **index()**⁸, którą powiążemy z adres URL głównej strony (/). Widok zwracał będzie kod wyrenderowany na podstawie szablonu **index.html**. W pliku **views.py** umieszczamy:

⁷ Danych z bazy przekazywane są do szablonów za pomocą Pythonowego słownika. Renderowanie polega na odszukaniu pliku szablonu, zastąpieniu przekazanych zmiennych danymi i odesłaniu całości (HTML + dane) do użytkownika.

⁸ Nazwa **index()** jest przykładowa, funkcja mogłaby się nazywać inaczej.

Kod V.1

```
# -*- coding: utf-8 -*-
# chatter/chatter/views.py

# HttpResponse pozwala zwracać proste wiadomości tekstowe
from django.http import HttpResponse
# render pozwala zwracać szablony
from django.shortcuts import render

def index(request):
    """Strona główna aplikacji."""
    # aby wyświetlić (zwrócić) prostą wiadomość tekstową wystarczy instrukcja poniżej:
    # return HttpResponse("Hello, SWOI")
    # my zwrócimy szablon index.html, uwaga (!) ścieżkę podajemy względną do katalogu templates :
    return render(request, 'chatter/index.html')
```

Widok **index()** łączymy z adresem URL strony głównej: **127.0.0.1:8000/** w pliku **urls.py**:

Kod V.2

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

from chatter import views # importujemy zdefiniowane w pliku views.py widoki

admin.autodiscover()

urlpatterns = patterns('',
    # główny adres (!) o nazwie index łączymy z widokiem index
    url(r'^$', views.index, name='index'),

    url(r'^admin/', include(admin.site.urls)),
)
```

Tworzymy katalog dla szablonów wydając polecenie:

Terminal V.1

```
~/chatter/chatter $ mkdir -p templates/chatter
```

Tworzymy szablon, plik **chatter/chatter/templates/chatter/index.html**, który zawiera:

Kod V.3

```
<!-- chatter/chatter/templates/chatter/index.html -->

<html>
  <head></head>
  <body>
    <h1>Witaj w systemie Chatter</h1>
  </body>
</html>
```

Po wpisaniu adresu **127.0.0.1:8080/** zobaczymy tekst, który zwróciliśmy z widoku, czyli "Hello, SWOI".

Witaj w systemie Chatter

VI. Logowanie użytkowników

Dodanie formularza logowania dla użytkowników polega na:

1. dodaniu w pliku **views.py** nowego widoku **my_login()**, który wywoływać będzie szablon **login.html** zapisany w **templates/chat**,
2. powiązaniu w pliku **urls.py** nowego widoku z adresem **/login**,

Django upraszcza zadanie, ponieważ zawiera odpowiednie formularze i model reprezentujący użytkowników w systemie, z którego – *nota bene* – skorzystaliśmy już podczas tworzenia bazy danych.

Widok **my_login()** wyświetli formularz logowania i obsłuży żądania typu POST (wysłanie danych z formularza na serwer), sprawdzi więc poprawność przesłanych danych (nazwa użytkownika i hasło). Jeżeli dane będą poprawne, zaloguje użytkownika i wyświetli spersonalizowaną stronę główną (**index.html**), w przeciwnym wypadku zwrócona zostanie informacja o błędzie.

Importujemy potrzebne moduły, tworzymy widok **my_login()** i uzupełniamy widok **index()** w pliku **views.py**:

Kod VI.1

```
# -*- coding: utf-8 -*-
# chatter/chat/views.py

# dodajemy nowe importy
from django.shortcuts import render, redirect
from django.contrib.auth import forms, authenticate, login, logout
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse

def index(request):
    """Strona glowna aplikacji."""
    # tworzymy zmienną (słownik), zawierającą informacje o użytkowniku
    context = {'user': request.user}
    # zmienna context przekazujemy do szablonu index.html
    return render(request, 'chat/index.html', context)

def my_login(request):
    """Logowanie uzytkownika w sytemie."""
    form = forms.AuthenticationForm() # ustawiamy formularz logowania

    if request.method == 'POST': # sprawdzamy, czy ktos probuje sie zalogowac
        # przypisujemy nadeslane dane do formularza logowania
        form = forms.AuthenticationForm(request, request.POST)
        # sprawdzamy poprawnosc formularza lub zwracamy informacje o bledzie
        if form.is_valid(): # jezeli wszystko jest ok - logujemy uzytkownika
            user = form.get_user()
            login(request, user)
            return redirect(reverse('index')) # przekierowujemy uzytkownika na strone glowna

    context = {'form': form} # ustawiamy zmienne przekazywane do templatki
    # renderujemy templatke logowania
    return render(request, 'chat/login.html', context)
```

W pliku **urls.py** dopisujemy regułę łączącą url **/login** z widokiem **my_login()**:

```
# adres logowania (/login) o nazwie login powiazany z widokiem my_login
url(r'^login/$', views.my_login, name='login'),
```

Kod VI.2

Tworzymy nowy szablon **login.html** w katalogu **templates/chat**ter/:

```
<!-- chatter/chatter/templates/login.html -->
<html>
  <body>
    <h1>Zaloguj się w systemie Chatter</h1>

    <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Zaloguj</button>
    </form>
  </body>
</html>
```

Kod VI.3

Zmieniamy również szablon **index.html** głównego widoku, aby uwzględnił status użytkownika (zalogowany/niezalogowany):

```
<! -- chatter/chatter/templates/chatter/index.html -->
<html>
  <head></head>
  <body>
    {% if not user.is_authenticated %}
      <h1>Witaj w systemie Chatter</h1>
      <p><a href="{% url 'login' %}">Zaloguj się</a></p>
    {% else %}
      <h1>Witaj, {{ user.username }}</h1>
    {% endif %}
  </body>
</html>
```

Kod VI.4

Zwróćmy uwagę, jak umieszczamy linki w szablonach. Mianowicie kod **{% url 'login' %}** wykorzystuje wbudowaną funkcję **url()**, która na podstawie nazwy adresu określonej w regułach pliku **urls.py** (parametr *name*) generuje skojarzony z nią adres.

JAK TO DZIAŁA: Po przejściu pod adres **127.0.0.1:8080/login/**, powiązany z widokiem **my_login()**, przeglądarka wysyła żądanie GET do serwera. Widok **my_login()** przygotowuje formularz autoryzacji (**AuthenticationForm**), przekazuje go do szablonu **login.html** i zwraca do klienta. Efekt jest taki:

Zaloguj się w systemie Chatter

Użytkownik:

Hasło:

Po wypełnieniu formularza danymi i kliknięciu przycisku „Zaloguj”, do serwera zostanie wysłane żądanie typu POST. W widoku `my_login()` obsługujemy taki przypadek za pomocą instrukcji if. Sprawdzamy poprawność przesłanych danych (walidacja), logujemy użytkownika w systemie i zwracamy przekierowanie na stronę główną, która wyświetla nazwę zalogowanego użytkownika. Jeżeli dane nie są poprawne, zwracana jest informacja o błędach. Przetestuj!

VII. Dodawanie i wyświetlanie wiadomości

Chcemy, by zalogowani użytkownicy mogli przeglądać wiadomości od innych użytkowników i dodawać własne. Utworzymy widok `messages()`, który wyświetli wszystkie wiadomości (żądanie GET) i ewentualnie zapisze nową wiadomość nadesłaną przez użytkownika (żądanie POST). Widok skorzysta z nowego szablonu `messages.html` i powiązany zostanie z adresem `/messages`. Zaczynamy od zmian w `views.py`.

Kod VII.1

```
# -*- coding: utf 8 -*-

# chatter/chatter/views.py

# dodajemy nowe importy
from chatter.models import Message
from django.utils import timezone
from django.contrib.auth.decorators import login_required

# pozostałe widoki

# dekorator, który "chroni" nasz widok przed dostępem przez osoby niezalogowane, jeżeli użytkownik niezalogowany
# będzie próbował odwiedzić ten widok, to zostanie przekierowany na stronę logowania
@login_required(login_url='/login')
def messages(request):
    """Widok wiadomości."""
    error = None

    # zadanie POST oznacza, że ktoś próbuje dodać nową wiadomość w systemie
    if request.method == 'POST':
        text = request.POST.get('text', '') # pobieramy treść przesłanej wiadomości
        # sprawdzamy, czy nie jest ona dłuższa od 250 znaków:
        # – jeżeli jest dłuższa, to zwracamy błąd, jeżeli jest krótsza lub równa, to zapisujemy ją w systemie
        if not 0 < len(text) <= 250:
            error = u'Wiadomość nie może być pusta i musi mieć co najwyżej
250 znaków'
        else:
            # ustawiamy dane dla modelu Message
            msg = Message(text=text, pub_date=timezone.now(),
user=request.user)
            msg.save() # zapisujemy nową wiadomość
            return redirect(reverse('messages')) # przekierowujemy na stronę wiadomości

    user = request.user # informacje o aktualnie zalogowanym użytkowniku
    messages = Message.objects.all() # pobieramy wszystkie wiadomości
    # ustawiamy zmienne przekazywane do szablonu
    context = {'user': user, 'messages': messages, 'error': error}
    # renderujemy templatę wiadomości
    return render(request, 'chatter/messages.html', context)
```

Teraz tworzymy nową templatkę **messages.html** w katalogu **templates/chat**ter/.

Kod VII.2

```
<html>
  <body>
    <h1>Witaj, {{ user.username }}</h1>

    <!-- w razie potrzeby wyświetlamy błędy -->
    {% if error %}
      <p>{{ error }}</p>
    {% endif %}

    <form method="POST">
      {% csrf_token %}
      <input name="text" />
      <button type="submit">Dodaj wiadomość</button>
    </form>

    <!-- wyświetlamy wszystkie wiadomości -->
    <h2>Wiadomości:</h2>
    <ol>
      {% for message in messages %}
        <li>
          <strong>{{ message.user.username }}</strong>
          ({{ message.pub_date }}):
          <br>
          {{ message.text }}
        </li>
      {% endfor %}
    </ol>
  </body>
</html>
```

Uzupełniamy szablon widoku głównego, aby zalogowanym użytkownikom wyświetlał się link prowadzący do strony z wiadomościami. W pliku **index.html** po klauzuli **{% else %}**, poniżej znacznika **<h1>** wstawiamy:

Kod VII.3

```
<p><a href="{% url 'messages' %}">Zobacz wiadomości</a></p>
```

Na koniec dodajemy nową regułę do **urls.py**:

Kod VII.4

```
url(r'^messages/$', views.messages, name='messages'),
```

Jeżeli uruchomimy serwer deweloperski, zalogujemy się do aplikacji i odwiedzimy adres **127.0.0.1:8080/messages/**, zobaczymy listę wiadomości dodanych przez użytkowników⁹.

Witaj, SWOI

Wiadomości:

1. **root** (11 sierpnia 2014 10:37:06):
nowa wiadomość

⁹ Jeżeli w panelu administracyjnym nie dodałeś żadnej wiadomości, lista będzie pusta.

JAK TO DZIAŁA

W widoku `messages()`, podobnie jak w widoku `login()`, mamy dwie ścieżki postępowania, w zależności od użytej metody HTTP. GET pobiera wszystkie wiadomości (`messages = Message.objects.all()`), przekazuje je do szablonu i renderuje. Django konstruuje odpowiednie zapytanie i mapuje dane z bazy na obiekty klasy `Message` (mapowanie obiektowo-relacyjne (ORM)).

POST zawiera z kolei treść nowej wiadomości, której długość sprawdzamy i jeżeli wszystko jest w porządku, tworzymy nową wiadomość (nowy obiekt) i zapisujemy ją w bazie danych (wywołujemy metodę obiektu). Odpowiada za to fragment:

```
msg = Message(text=text, pub_date=timzone.now(), user=request.user)
msg.save()
```

VIII. Rejestrowanie użytkowników

Tworzymy nowy widok `my_register()`, szablon `register.html` i nowy adres URL `/register`, który skieruje użytkownika do formularza rejestracji, wymagającego podania nazwy i hasła. Zaczynamy od dodania widoku w pliku `views.py`.

Kod VIII.1

```
# chatter/chatter/views.py

# pozostałe widoki

def my_register(request):
    """Rejestracja nowego użytkownika."""
    form = forms.UserCreationForm() # ustawiamy formularz rejestracji

    # POST oznacza, że ktoś próbuje utworzyć nowego użytkownika
    if request.method == 'POST':
        # przypisujemy nadesłane dane do formularza tworzenia użytkownika
        form = forms.UserCreationForm(request.POST)
        # sprawdzamy poprawność nadesłanych danych:
        # – jeżeli wszystko jest w porządku to stworzymy użytkownika
        # – w przeciwnym razie zwracamy formularz wraz z informacją o błędach
        if form.is_valid():
            # zapamiętujemy podaną nazwę użytkownika i hasło
            username = form.data['username']
            password = form.data['password1']
            # zapisujemy formularz tworząc nowego użytkownika
            form.save()
            # uwierzytelniamy użytkownika
            user = authenticate(username=username, password=password)
            login(request, user)
            # po udanej rejestracji, przekierowujemy go na stronę główną
            return redirect(reverse('index'))

    # ustawiamy zmienne przekazywane do szablonu
    context = {'form': form}
    # renderujemy templatę rejestracji
    return render(request, 'chatter/register.html', context)
```

Tworzymy nowy szablon `register.html` w katalogu `templates/chatter`:

```
<!-- chatter/chatter/templates/register.html -->

<html>
  <body>
    <h1>Zarejestruj nowego użytkownika w systemie Chatter</h1>

    <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Zarejestruj</button>
    </form>
  </body>
</html>
```

W szablonie widoku głównego **index.html** po linku „Zaloguj się” wstawiamy kolejny:

```
<p><a href="{% url 'register' %}">Zarejestruj się</a></p>
```

Na koniec uzupełniamy plik **urls.py**.

```
url(r'^register/$', views.my_register, name='register'),
```

JAK TO DZIAŁA

Zasada działania jest taka sama jak w przypadku pozostałych widoków. Po wpisaniu adresu **127.0.0.1:8080/register/** otrzymujemy formularz rejestracji nowego użytkownika, który podobnie jak formularz logowania, jest wbudowany w Django, więc wystarczy przekazać go do szablonu. Po wypełnieniu i zatwierdzeniu formularza wysyłamy żądanie POST, widok **my_register()** odbiera przekazane dane (nazwę użytkownika, hasło i powtórzone hasło), sprawdza ich poprawność (poprawność i unikalność nazwy użytkownika oraz hasło) oraz tworzy i zapisuje nowego użytkownika. Po rejestracji użytkownik przekierowywany jest na stronę główną.

Zarejestruj nowego użytkownika w systemie Chatter

Nazwa użytkownika: Wymagane. 30 znaków lub mniej. Tylko litery, cyfry i znaki @/./+/_.

Hasło:

Potwierdzenie hasła: Podaj powyższe hasło w celu weryfikacji.

IX. Wylogowywanie użytkowników

Django ma wbudowaną również funkcję wylogowującą. Utworzymy zatem nowy widok `my_logout()` i powiążemy go z adresem `/logout`. Do pliku `views.py` dodajemy:

Kod IX.1

```
def my_logout(request):
    """Wylogowywanie uzytkownika z systemu"""
    logout(request)
    # przekierowujemy na strone glowna
    return redirect(reverse('index'))
```

Następnie dodajemy nową regułę do `urls.py`:

Kod IX.2

```
url(r'^logout/$', views.my_logout, name='logout'),
```

Wylogowywanie nie wymaga osobnego szablonu, dodajemy natomiast link wylogowujący do 1) szablonu `index.html` po linku „Zobacz wiadomości” oraz do 2) szablonu `messages.html` po nagłówku `<h1>`:

Kod IX.3

```
<p><a href="{% url 'logout' %}">Wyloguj</a></p>
```

Witaj, SWOI

[Wyloguj](#)

Wiadomości:

1. **sru** (13 września 2014 12:35:34):
Pierwsza wiadomość
2. **SWOI** (14 września 2014 10:46:23):
Zrobiłem pierwszą aplikację w Django!

Film instruktażowy: http://youtu.be/aHy_GvsQ9uo

Słownik pojęć:

- Aplikacja – program komputerowy.
- Framework – zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji.
- GET – typ żądania HTTP, służący do pobierania zasobów z serwera WWW.
- HTML – język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.
- HTTP – protokół przesyłania dokumentów WWW,
- Logowanie – proces autoryzacji i uwierzytelniania użytkownika w systemie.
- Model – źródło danych aplikacji.
- ORM – mapowanie obiektowo-relacyjne, oprogramowanie służące do przekształcania struktur bazy danych na obiekty klasy danego języka programowania.
- POST – typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW.
- Serwer deweloperski – serwer używany w czasie prac nad oprogramowaniem.
- Serwer WWW – serwer obsługujący protokół HTTP.
- Templatka – szablon strony WWW wykorzystywany przez Django do renderowania widoków.
- URL – ustandaryzowany format adresowania zasobów w internecie (przykład: adres strony WWW).
- Widok – fragment danych, który jest reprezentowany użytkownikowi.

Materiały pomocnicze:

1. O Django [http://pl.wikipedia.org/wiki/Django_\(informatyka\)](http://pl.wikipedia.org/wiki/Django_(informatyka))
2. Strona projektu Django <https://www.djangoproject.com/>
3. Co to jest framework? <http://pl.wikipedia.org/wiki/Framework>
4. Co nieco o HTTP i żądaniach GET i POST <http://pl.wikipedia.org/wiki/Http>