Wydanie III

Django 3

Praktyczne tworzenie aplikacji sieciowych

Antonio Melé

Helion

Packt

Tytuł oryginału: Django 3 By Example: Build powerful and reliable Python web applications from scratch, 3rd Edition

Tłumaczenie: Radosław Meryk, z wykorzystaniem fragmentów książki
Django. Praktyczne tworzenie aplikacji sieciowych w tłumaczeniu Roberta Górczyńskiego

ISBN: 978-83-283-7250-4

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/dj3pt3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/dj3pt3.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 4

Utworzenie witryny społecznościowej

W poprzednim rozdziale dowiedziałeś się, jak opracować mapę witryny i kanał wiadomości dla postów bloga oraz zaimplementować silnik wyszukiwania w naszej aplikacji bloga. W tym rozdziale przechodzimy do opracowania aplikacji społecznościowej. Użytkownicy będą mogli dołączyć do platformy online i komunikować się pomiędzy sobą, współdzieląc treści. W kilku kolejnych rozdziałach zajmiemy się budowaniem platformy do współdzielenia zdjęć. Użytkownicy będą mogli oznaczyć dowolne zdjęcie w internecie i udostępnić je innym. Będą również mogli zobaczyć aktywność w internecie użytkowników, których obserwują, oraz polubić udostępniane przez nich zdjęcia (lub wyrazić dezaprobatę na ich temat).

W tym rozdziale zaczniemy od przygotowania funkcjonalności pozwalającej użytkownikom na logowanie, wylogowanie oraz edytowanie i zerowanie hasła. Zobaczysz, jak można utworzyć niestandardowe profile dla użytkowników i jak zaimplementować uwierzytelnianie za pomocą innej witryny społecznościowej.

Oto zagadnienia, na których skoncentruję się w tym rozdziale.

- Użycie frameworka uwierzytelniania Django.
- Utworzenie widoków pozwalających na rejestrację użytkowników.
- Rozbudowa modelu `User` o obsługę niestandardowego profilu.
- Implementacja uwierzytelnienia społecznościowego za pomocą modułu `python-social-auth`.

Pracę rozpoczynamy od utworzenia nowego projektu.

Utworzenie projektu witryny społecznościowej

Przystępujemy teraz do budowy aplikacji społecznościowej umożliwiającej użytkownikom udostępnianie obrazów znalezionych w internecie. Na potrzeby tego projektu konieczne jest opracowanie pewnych komponentów. Oto one.

- System uwierzytelniania pozwalający użytkownikowi na rejestrowanie, logowanie, edycję profilu oraz zmianę i zerowanie hasła.
- System obserwacji pozwalający użytkownikom na śledzenie swoich poczynań.
- Funkcjonalność pozwalająca na wyświetlanie udostępnianych obrazów oraz implementacja bookmarkletu umożliwiającego użytkownikowi udostępnianie obrazów z praktycznie każdej witryny internetowej.
- Strumień aktywności dla każdego użytkownika pozwalający użytkownikom śledzić treść dodawaną przez obserwowanych użytkowników.

W tym rozdziale zajmiemy się realizacją pierwszego z wymienionych punktów.

Rozpoczęcie pracy nad aplikacją społecznościową

Przejdź do powłoki i wydaj poniższe polecenia w celu utworzenia środowiska wirtualnego dla projektu, a następnie jego aktywacji.

```
mkdir env
python3 -m venv env/bookmarks
source env/bookmarks/bin/activate
```

Znak zachęty w powłoce wyświetla nazwę aktywnego środowiska wirtualnego, co pokazałem poniżej.

```
(bookmarks)laptop:~ zenx$
```

W przygotowanym środowisku wirtualnym zainstaluj framework Django, wydając poniższe polecenie.

```
pip install "Django==3.0.*"
```

Aby utworzyć nowy projekt, wydaj poniższe polecenie.

```
django-admin startproject bookmarks
```

W ten sposób zbudujemy nowy projekt Django o nazwie `bookmarks` wraz z początkową strukturą plików i katalogów. Teraz przejdź do nowego katalogu projektu i utwórz nową aplikację o nazwie `account`, wydając poniższe polecenia.

```
cd bookmarks/
django-admin startapp account
```

Pamiętaj, by dodać aplikację do projektu; zrobisz to, wpisując ją na listę `INSTALLED_APPS` w pliku *settings.py*. Naszą aplikację umieść na początku listy, przed pozostałymi zainstalowanymi aplikacjami, tak jak pokazałem poniżej.

```
INSTALLED_APPS = (
    'account.apps.AccountConfig',
    #…
)
```

Szablony uwierzytelniania Django zdefiniujemy nieco później. Umieszczając aplikację na pierwszym miejscu w ustawieniu `INSTALLED_APPS`, zapewniamy domyślne wykorzystywanie naszych szablonów uwierzytelniania, a nie szablonów uwierzytelniania zawartych w innych aplikacjach. Django szuka szablonów według kolejności występowania aplikacji w ustawieniu `INSTALLED_APPS`.

Uruchom poniższe polecenie, aby przeprowadzić synchronizację bazy danych z modelami aplikacji domyślnych wskazanymi na liście `INSTALLED_APPS`.

```
python manage.py migrate
```

Zobaczysz, że zostaną zastosowane wszystkie początkowe migracje bazy danych Django. Teraz przystępujemy do budowy systemu uwierzytelniania w projekcie z wykorzystaniem frameworka uwierzytelniania Django.

Użycie frameworka uwierzytelniania w Django

Django jest dostarczany wraz z wbudowanym frameworkiem uwierzytelniania, który może obsługiwać uwierzytelnianie użytkowników, sesje, uprawnienia i grupy użytkowników. System uwierzytelniania oferuje widoki dla działań najczęściej podejmowanych przez użytkowników, takich jak logowanie, wylogowanie, zmiana hasła i zerowanie hasła.

Wspomniany framework uwierzytelniania znajduje się w `django.contrib.auth` i jest używany także przez inne pakiety Django, typu `contrib`. Framework uwierzytelniania wykorzystaliśmy już w rozdziale 1., „Budowa aplikacji bloga" do utworzenia superużytkownika dla aplikacji bloga, aby mieć dostęp do witryny administracyjnej.

Kiedy tworzysz nowy projekt Django za pomocą polecenia `startproject`, framework uwierzytelniania zostaje wymieniony w domyślnych ustawieniach projektu. Składa się z aplikacji `django.contrib.auth` oraz przedstawionych poniżej dwóch klas wymienionych w opcji `MIDDLEWARE` projektu.

- `AuthenticationMiddleware`. Wiąże użytkowników z żądaniami za pomocą mechanizmu sesji.
- `SessionMiddleware`. Zapewnia obsługę bieżącej sesji między poszczególnymi żądaniami.

Oprogramowanie pośredniczące to klasy wraz z metodami wykonywanymi globalnie w trakcie fazy przetwarzania żądania lub udzielania odpowiedzi na nie. W tej książce klasy oprogramowania pośredniczącego będziemy wykorzystywać w wielu sytuacjach. Temat tworzenia oprogramowania pośredniczącego zostanie dokładnie omówiony w rozdziale 14., „Wdrażanie".

Framework uwierzytelniania zawiera również wymienione poniżej modele.

- `User`. Model użytkownika wraz z podstawowymi kolumnami, takimi jak `username`, `password`, `email`, `first_name`, `last_name` i `is_active`.
- `Group`. Model grupy do nadawania kategorii użytkownikom.
- `Permission`. Uprawnienia pozwalające na wykonywanie określonych operacji.

Opisywany framework zawiera także domyślne widoki uwierzytelniania i formularze, z których będziemy korzystać nieco później.

Utworzenie widoku logowania

Rozpoczynamy od użycia wbudowanego w Django frameworka uwierzytelniania w celu umożliwienia użytkownikom zalogowania się w witrynie. Aby zalogować użytkownika, widok powinien wykonywać poniższe akcje.

- Pobranie nazwy użytkownika i hasła z wysłanego formularza logowania.
- Uwierzytelnienie użytkownika na podstawie danych przechowywanych w bazie danych.
- Sprawdzenie, czy konto użytkownika jest aktywne.
- Zalogowanie użytkownika w witrynie i rozpoczęcie uwierzytelnionej sesji.

Najpierw musimy przygotować formularz logowania. Utwórz nowy plik *forms.py* w katalogu aplikacji `account` i umieść w nim poniższy fragment kodu.

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

Formularz będzie używany do uwierzytelnienia użytkownika na podstawie informacji przechowywanych w bazie danych. Zwróć uwagę na wykorzystanie widżetu `PasswordInput` do wygenerowania elementu HTML `<input>` wraz z atrybutem `type="password"` po to, aby przeglądarka interpretowała wprowadzane dane jako hasło.

Przeprowadź edycję pliku *views.py* aplikacji `account` i umieść w nim poniższy fragment kodu.

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from .forms import LoginForm

def user_login(request):
```

```
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            user = authenticate(username=cd['username'],
                                password=cd['password'])
            if user is not None:
                if user.is_active:
                    login(request, user)
                    return HttpResponse('Uwierzytelnienie zakończyło się sukcesem.')
                else:
                    return HttpResponse('Konto jest zablokowane.')
            else:
                return HttpResponse('Nieprawidłowe dane uwierzytelniające.')
    else:
        form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

To jest kod podstawowego widoku logowania użytkownika. Po wywołaniu widoku `user_login` przez żądanie `GET` za pomocą wywołania `form = LoginForm()` tworzymy nowy egzemplarz formularza logowania i wyświetlamy go w szablonie. Kiedy użytkownik wyśle formularz przy użyciu żądania `POST`, przeprowadzane są następujące akcje.

- Utworzenie egzemplarza formularza wraz z wysłanymi danymi. Do tego celu służy polecenie `form = LoginForm(request.POST)`.
- Sprawdzenie za pomocą wywołania `form.is_valid()`, czy formularz jest prawidłowy. Jeżeli formularz jest nieprawidłowy, w szablonie wyświetlamy błędy wykryte podczas weryfikacji formularza (np. użytkownik nie wypełnił jednego z pól).
- Jeżeli wysłane dane są prawidłowe, za pomocą metody `authenticate()` uwierzytelniamy użytkownika na podstawie informacji przechowywanych w bazie danych. Wymieniona metoda pobiera `username` i `password`, a zwraca obiekt `User`, gdy użytkownik zostanie uwierzytelniony, lub `None` w przeciwnym przypadku. Ponadto jeśli użytkownik nie będzie uwierzytelniony, zwracamy także obiekt `HttpResponse` wraz z komunikatem „Nieprawidłowe dane uwierzytelniające”.
- W przypadku pomyślnego uwierzytelnienia użytkownika za pomocą atrybutu `is_active` sprawdzamy, czy jego konto użytkownika jest aktywne. Wymieniony atrybut pochodzi z modelu `User` dostarczanego przez Django. Gdy konto użytkownika jest nieaktywne, zwracamy obiekt `HttpResponse` wraz z komunikatem „Konto jest zablokowane”.
- Gdy konto użytkownika jest aktywne, logujemy go w witrynie internetowej. Rozpoczynamy także sesję dla użytkownika: wywoływana jest metoda `login()` i zwracany komunikat „Uwierzytelnienie zakończyło się sukcesem.”.

> Zwróć uwagę na różnice między metodami `authenticate()` i `login()`. Metoda `authenticate()` sprawdza dane uwierzytelniające użytkownika i jeśli są prawidłowe, zwraca obiekt użytkownika. Natomiast metoda `login()` umieszcza użytkownika w bieżącej sesji.

Teraz musimy opracować wzorzec adresu URL dla nowo zdefiniowanego widoku. Utwórz nowy plik *urls.py* w katalogu aplikacji `account` i umieść w nim poniższy fragment kodu.

```
from django.urls import path
from . import views

urlpatterns = [
    # Widoki logowania.
    path('login/', views.user_login, name='login')
]
```

Przeprowadź edycję głównego pliku *urls.py* znajdującego się katalogu projektu `bookmarks` i dodaj wzorzec adresu URL aplikacji `account`, co przedstawiłem poniżej.

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

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

Widok logowania jest teraz dostępny za pomocą adresu URL. Przechodzimy więc do przygotowania szablonu dla tego widoku. Ponieważ w projekcie nie mamy jeszcze żadnych szablonów, najpierw musimy utworzyć szablon bazowy, który następnie będzie mógł być rozszerzony przez szablon logowania. Wymienioną poniżej strukturę plików i katalogów utwórz w katalogu aplikacji `account`.

```
templates/
    account/
        login.html
    base.html
```

Przeprowadź edycję pliku *base.html* i umieść w nim poniższy fragment kodu.

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <span class="logo">Bookmarks</span>
  </div>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
</body>
</html>
```

W ten sposób przygotowaliśmy szablon bazowy dla budowanej witryny internetowej. Podobnie jak w poprzednim projekcie, także w tym style CSS dołączamy w szablonie głównym. Niezbędne pliki statyczne znajdziesz w materiałach przygotowanych dla książki. Wystarczy skopiować

podkatalog *static* z katalogu *account* we wspomnianych materiałach i umieścić go w tym samym położeniu budowanego projektu. Zawartość katalogów można znaleźć pod adresem *https://github.com/PacktPublishing/Django-3-by-Example/tree/master/Chapter04/bookmarks/account/static*.

Szablon bazowy definiuje bloki `title` i `content`, które mogą być wypełniane przez treść szablonów rozszerzających szablon bazowy.

Przechodzimy do utworzenia szablonu dla formularza logowania. W tym celu otwórz plik *account/login.html* i umieść w nim poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Logowanie{% endblock %}

{% block content %}
  <h1>Logowanie</h1>
  <p>Wypełnij poniższy formularz, aby się zalogować:</p>
  <form action="." method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Zaloguj"></p>
  </form>
{% endblock %}
```

Ten szablon zawiera formularz, którego egzemplarz jest tworzony w widoku. Ponieważ formularz zostanie wysłany za pomocą metody `POST`, dołączamy znacznik szablonu `{% csrf_token %}` w celu zapewnienia ochrony przed atakami typu CSRF. Więcej informacji na temat ataków CSRF przedstawiłem w rozdziale 2., „Usprawnienie bloga za pomocą funkcji zaawansowanych".

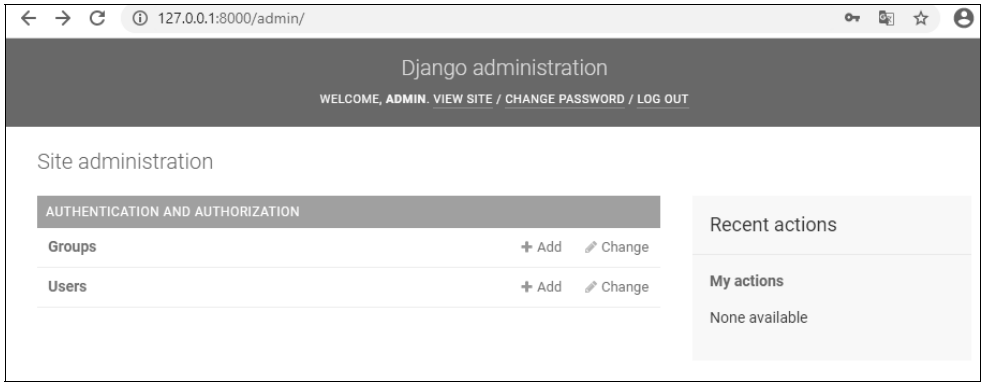
W bazie danych nie ma jeszcze żadnych kont użytkowników. Konieczne jest utworzenie najpierw superużytkownika, aby zapewnić sobie dostęp do witryny administracyjnej i zarządzać pozostałymi użytkownikami. Przejdź do powłoki i wydaj polecenie `python manage.py createsuperuser`. Podaj wybraną nazwę użytkownika, adres e-mail i hasło. Następnie uruchom serwer programistyczny przez wydanie polecenia `python manage.py runserver` i w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/admin/*. Dostęp do witryny administracyjnej uzyskasz po podaniu ustalonej przed chwilą nazwy użytkownika i hasła. Gdy znajdziesz się już w witrynie administracyjnej Django, zobaczysz modele `Users` (łącze *Użytkownicy*) i `Group` (łącze *Grupy*) dla wbudowanego w Django frameworka uwierzytelniania.

Strona wygląda następująco (rysunek 4.1)[1].

[1] Aby witryna administracyjna miała polski interfejs użytkownika, należy w pliku settings.py projektu wprowadzić następujące ustawienia:

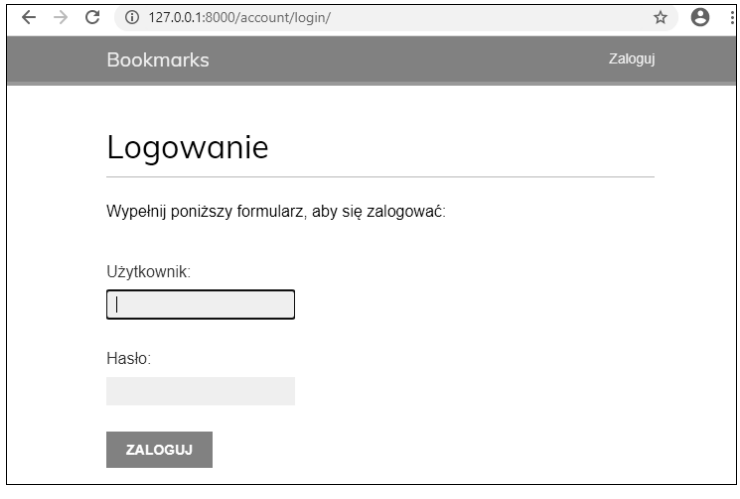
```
LANGUAGE_CODE = 'pl'
TIME_ZONE = 'Europe/Warsaw'
USE_I18N = True
USE_L10N = True
USE_TZ = False
```

— *przyp. tłum.*



Rysunek 4.1. Główna strona witryny administracyjnej Django włącznie z modelami Users i Group

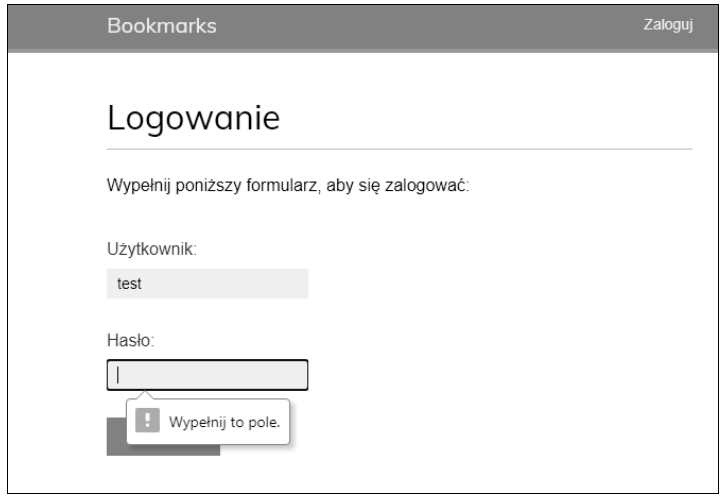
Utwórz nowego użytkownika, używając do tego witryny administracyjnej, a następnie w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/*. Powinieneś zobaczyć wygenerowany szablon wraz z formularzem logowania (rysunek 4.2).



Rysunek 4.2. Strona logowania użytkownika

Spróbuj teraz wysłać formularz, pozostawiając niewypełnione jedno z pól. W takim przypadku formularz jest uznawany za nieprawidłowy i zostanie wyświetlony komunikat o błędzie (rysunek 4.3).

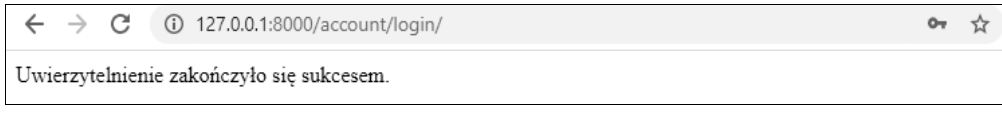
Warto wiedzieć, że niektóre nowoczesne przeglądarki uniemożliwiają przesyłanie formularzy z pustymi lub błędnymi polami. Dzieje się tak dlatego, że przeglądarka sprawdza poprawność formularza na podstawie typów pól i ograniczeń na poziomie pól. W takim przypadku formularz nie zostanie przesłany, a przeglądarka wyświetli komunikat o błędzie dla pól, które są wypełnione nieprawidłowo.



Rysunek 4.3. Formularz logowania z błędami w polach

Jeżeli podasz dane nieistniejącego użytkownika lub błędne hasło, Django wygeneruje komunikat o nieudanym logowaniu.

Natomiast po podaniu prawidłowych danych uwierzytelniających Django wyświetli komunikat o zakończonym sukcesem logowaniu, co pokazałem poniżej (rysunek 4.4).



Rysunek 4.4. Komunikat tekstowy z informacją o pomyślnym uwierzytelnianiu

Użycie widoków uwierzytelniania w Django

Framework uwierzytelniania w Django zawiera wiele formularzy i widoków gotowych do natychmiastowego użycia. Utworzony przed chwilą widok logowania to dobre ćwiczenie pomagające w zrozumieniu procesu uwierzytelniania użytkowników w Django. Jednak w większości przypadków możesz wykorzystać wspomniane domyślne widoki uwierzytelniania.

Do obsługi uwierzytelniania Django oferuje wymienione poniżej widoki. Wszystkie są dostępne w module `django.contrib.auth.views`.

- `LoginView`. Obsługa formularza logowania oraz proces zalogowania użytkownika.
- `LogoutView`. Obsługa wylogowania użytkownika.

Do obsługi zmiany hasła Django oferuje wymienione poniżej widoki.

- `PasswordChangeView`. Obsługa formularza pozwalającego użytkownikowi na zmianę hasła.
- `PasswordChangeDoneView`. Strona informująca o sukcesie operacji; zostanie wyświetlona użytkownikowi, gdy zmiana hasła zakończy się powodzeniem.

Natomiast do obsługi operacji zerowania hasła Django oferuje następujące widoki.

- `PasswordResetView`. Umożliwienie użytkownikowi wyzerowania hasła. Generowane jest przeznaczone tylko do jednokrotnego użycia łącze wraz z tokenem, które następnie będzie wysłane na adres e-mail danego użytkownika.
- `PasswordResetDoneView`. Wyświetlenie użytkownikowi strony z informacją o wysłaniu wiadomości e-mail wraz z łączem pozwalającym na wyzerowanie hasła.
- `PasswordResetConfirmView`. Widok umożliwiający użytkownikowi zdefiniowanie nowego hasła.
- `PasswordResetCompleteView`. Strona informująca o sukcesie operacji; zostanie wyświetlona użytkownikowi, gdy wyzerowanie hasła zakończy się powodzeniem.

Zastosowanie wymienionych wyżej widoków może zaoszczędzić sporą ilość czasu podczas tworzenia witryny internetowej obsługującej konta użytkowników. W widokach tych używane są wartości domyślne, które oczywiście można nadpisać. Przykładem może być wskazanie położenia szablonu przeznaczonego do wygenerowania lub formularza wyświetlanego przez widok.

Więcej informacji na temat wbudowanych widoków uwierzytelniania znajdziesz na stronie *https://docs.djangoproject.com/en/3.0/topics/auth/default/#module-django.contrib.auth.views*.

Widoki logowania i wylogowania

Przeprowadź edycję pliku *urls.py* aplikacji `account` i dodaj kolejne wzorce adresów URL. Po wprowadzeniu zmian zawartość wymienionego pliku powinna przedstawiać się następująco.

```
from django.urls import path
from django.contrib.auth import views as auth_views
from . import views

urlpatterns = [
    # poprzedni widok login view
    # path('login/', views.user_login, name='login'),
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

Umieściliśmy znak komentarza na początku wiersza wzorca adresu URL dla utworzonego wcześniej widoku `LoginView`. Teraz wykorzystamy widok `login` oferowany przez wbudowany w Django framework uwierzytelniania. Dodaliśmy także wzorzec adresu URL dla widoku `LogoutView`.

Utwórz nowy podkatalog w katalogu szablonów aplikacji `account` i nadaj mu nazwę *registration*. Podkatalog ten to domyślna lokalizacja, w której widoki uwierzytelniania Django spodziewają się znaleźć szablony.

Moduł `django.contrib.admin` zawiera kilka szablonów uwierzytelniania, które są używane w witrynie administracyjnej. Aplikację `account` umieściliśmy na początku ustawienia `INSTALLED_APPS`, aby Django domyślnie korzystał z naszych szablonów zamiast szablonów uwierzytelniania zdefiniowanych w innych aplikacjach.

W katalogu *templates/registration* utwórz plik *login.html* i umieść w nim poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Logowanie{% endblock %}

{% block content %}
  <h1>Logowanie</h1>
  {% if form.errors %}
    <p>
      Nazwa użytkownika lub hasło są nieprawidłowe.
      Spróbuj ponownie.
    </p>
  {% else %}
    <p>Wypełnij poniższy formularz, aby się zalogować:</p>
  {% endif %}
  <div class="login-form">
    <form action="{% url 'login' %}" method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="hidden" name="next" value="{{ next }}" />
      <p><input type="submit" value="Zaloguj"></p>
    </form>
  </div>
{% endblock %}
```

Ten szablon logowania jest bardzo podobny do utworzonego wcześniej. Domyślnie Django używa formularza `AuthenticationForm` pochodzącego z modułu `django.contrib.auth.forms`. Formularz próbuje uwierzytelnić użytkownika i zgłasza błąd weryfikacji, gdy logowanie zakończy się niepowodzeniem. W takim przypadku za pomocą znacznika szablonu `{% if form.errors %}` można przeanalizować te błędy, aby sprawdzić, czy podane zostały nieprawidłowe dane uwierzytelniające.

Zwróć uwagę na dodanie ukrytego elementu HTML `<input>` przeznaczonego do wysłania wartości zmiennej o nazwie `next`. Zmienna jest ustawiania przez widok logowania, gdy w żądaniu będzie przekazany parametr `next` (np. *http://127.0.0.1:8000/account/login/?next=/account/*).

Wartością parametru `next` musi być adres URL. Jeżeli ten parametr zostanie podany, widok logowania w Django przekieruje użytkownika po zalogowaniu do podanego adresu URL.

Teraz utwórz szablon *logged_out.html* w katalogu *registration* i umieść w nim następujący fragment kodu.

```
{% extends "base.html" %}

{% block title %}Wylogowanie{% endblock %}

{% block content %}
  <h1>Wylogowanie</h1>
  <p>Zostałeś pomyślnie wylogowany. Możesz <a href="{% url
"login" %}">zalogować się ponownie</a>.</p>
{% endblock %}
```

Ten szablon zostanie przez Django wyświetlony po wylogowaniu użytkownika.

Po dodaniu wzorców adresu URL oraz szablonów dla widoków logowania i wylogowania budowana tutaj witryna internetowa jest gotowa na obsługę logowania użytkowników za pomocą oferowanych przez Django widoków uwierzytelniania.

Przystępujemy teraz do utworzenia nowego widoku przeznaczonego do wyświetlenia użytkownikowi panelu głównego (ang. *dashboard*) po tym, jak już zaloguje się w aplikacji. Otwórz plik *views.py* aplikacji `account` i umieść w nim poniższy fragment kodu.

```
from django.contrib.auth.decorators import login_required

@login_required
def dashboard(request):
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard'})
```

Widok został oznaczony dekoratorem `login_required` frameworka uwierzytelniania. Zadanie dekoratora `login_required` polega na sprawdzeniu, czy bieżący użytkownik został uwierzytelniony. Jeżeli użytkownik jest uwierzytelniony, następuje wykonanie udekorowanego widoku. Gdy natomiast użytkownik nie jest uwierzytelniony, zostaje przekierowany na stronę logowania, a adres URL, do którego próbował uzyskać dostęp, będzie podany jako wartość parametru `next` żądania `GET`.

Tym samym po udanym logowaniu użytkownik powróci na stronę, do której wcześniej próbował uzyskać dostęp. Pamiętaj, że do obsługi tego rodzaju sytuacji dodaliśmy w szablonie logowania ukryty element `<input>`.

Zdefiniowaliśmy również zmienną `section`. Wykorzystamy ją do ustalenia, którą sekcję witryny obserwuje użytkownik. Wiele widoków może odpowiadać tej samej sekcji. To jest prosty sposób na zdefiniowanie, której sekcji odpowiadają poszczególne widoki.

Teraz należy utworzyć szablon dla widoku panelu głównego. Utwórz nowy plik w katalogu *templates/account/*, nadaj mu nazwę *dashboard.html* i umieść w nim przedstawiony poniżej kod.

```
{% extends "base.html" %}

{% block title %}Panel główny{% endblock %}

{% block content %}
  <h1>Panel główny</h1>
  <p>Witaj w panelu głównym.</p>
{% endblock %}
```

Kolejnym krokiem jest dodanie poniższego wzorca adresu URL dla nowego widoku. To zadanie przeprowadzamy w pliku *urls.py* aplikacji `account`.

```
urlpatterns = [
    #...
    path('', views.dashboard, name='dashboard'),
]
```

Teraz przeprowadź edycję pliku *settings.py* projektu `bookmarks` i dodaj poniższy fragment kodu.

```
LOGIN_REDIRECT_URL = 'dashboard'
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
```

Oto wyjaśnienie działania poszczególnych opcji.

- `LOGIN_REDIRECT_URL`. Wskazujemy Django adres URL, do którego ma nastąpić przekierowanie, gdy widok `contrib.auth.views.login` nie otrzymuje parametru `next`.
- `LOGIN_URL`. Adres URL, do którego ma nastąpić przekierowanie po zalogowaniu użytkownika (np. za pomocą dekoratora `login_required`).
- `LOGOUT_URL`. Adres URL, do którego ma nastąpić przekierowanie po wylogowaniu użytkownika.

Używamy nazw wzorców adresów URL, które wcześniej zdefiniowaliśmy przy użyciu atrybutu `name` funkcji `path()`. Do tych ustawień zamiast nazw adresów URL można również użyć zakodowanych „na sztywno" adresów URL.

Oto krótkie podsumowanie przeprowadzonych dotąd działań.

- Do projektu dodaliśmy wbudowane we frameworku uwierzytelniania Django widoki logowania i wylogowania.
- Przygotowaliśmy własne szablony dla obu widoków i zdefiniowaliśmy prosty widok, do którego użytkownik zostanie przekierowany po zalogowaniu.
- Na koniec skonfigurowaliśmy ustawienia Django, aby wspomniane adresy URL były używane domyślnie.

Teraz do szablonu bazowego dodamy łącza logowania i wylogowania, co pozwoli na zebranie wszystkiego w całość. Aby to zrobić, koniecznie trzeba ustalić, czy bieżący użytkownik jest zalogowany. Na tej podstawie zostanie wyświetlone prawidłowe łącze (logowania lub wylogowania). Bieżący użytkownik jest przez oprogramowanie pośredniczące ustawiony w obiekcie `HttpRequest`. Dostęp do niego uzyskujesz za pomocą `request.user`. Użytkownika znajdziesz w wymienionym obiekcie nawet wtedy, gdy nie został uwierzytelniony. W takim przypadku użytkownik będzie zdefiniowany w postaci egzemplarza obiektu `AnonymousUser`. Najlepszym sposobem zweryfikowania, czy użytkownik został uwierzytelniony, jest sprawdzenie wartości jego atrybutu „tylko do odczytu" `is_authenticated`.

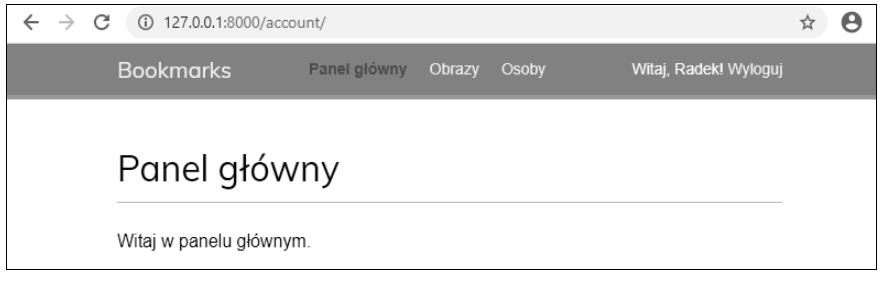
Przeprowadź edycję pliku *base.html* i zmodyfikuj element `<div>` o identyfikatorze `header`, tak jak przedstawiłem poniżej.

```
<div id="header">
  <span class="logo">Bookmarks</span>
  {% if request.user.is_authenticated %}
    <ul class="menu">
      <li {% if section == "dashboard" %}class="selected"{% endif %}>
        <a href="{% url "dashboard" %}">Panel główny</a>
      </li>
      <li {% if section == "images" %}class="selected"{% endif %}>
```

```
        <a href="#">Obrazy</a>
      </li>
      <li {% if section == "people" %}class="selected"{% endif %}>
        <a href="#">Osoby</a>
      </li>
    </ul>
  {% endif %}

  <span class="user">
    {% if request.user.is_authenticated %}
      Witaj, {{ request.user.first_name }}!
      <a href="{% url "logout" %}">Wyloguj</a>
    {% else %}
      <a href="{% url "login" %}">Zaloguj</a>
    {% endif %}
  </span>
</div>
```

Jak możesz zobaczyć, menu witryny internetowej będzie wyświetlane jedynie uwierzytelnionym użytkownikom. Sprawdzana jest także bieżąca sekcja witryny, aby dodać klasę atrybutu `selected` do odpowiedniego elementu `<li>` i tym samym za pomocą CSS podświetlić nazwę aktualnej sekcji. Wyświetlane jest również imię uwierzytelnionego użytkownika i łącze pozwalające mu na wylogowanie. Jeżeli użytkownik nie jest uwierzytelniony, wyświetlone będzie jedynie łącze pozwalające mu na zalogowanie.

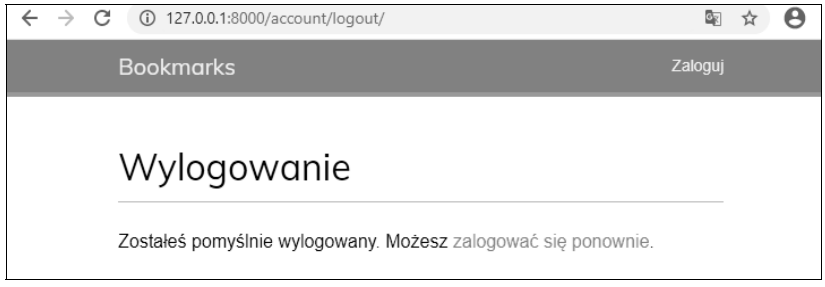
Teraz w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/*. Powinieneś zobaczyć stronę logowania. Podaj prawidłowe dane uwierzytelniające i kliknij przycisk *Zaloguj*. Po udanym logowaniu znajdziesz się na stronie pokazanej poniżej (rysunek 4.5).



Rysunek 4.5. Strona panelu głównego

Jak można zobaczyć, nazwa sekcji *Panel główny* została za pomocą stylów CSS wyświetlona innym kolorem czcionki, ponieważ odpowiadającemu jej elementowi `<li>` przypisaliśmy klasę `selected`. Skoro użytkownik jest uwierzytelniony, jego imię wyświetlamy po prawej stronie nagłówka. Kliknij łącze *Wyloguj*, powinieneś zobaczyć stronę pokazaną na rysunku 4.6.

Na tej stronie został wyświetlony komunikat informujący o udanym wylogowaniu i dlatego nie jest dłużej wyświetlane menu witryny internetowej. Łącze znajdujące się po prawej stronie nagłówka zmienia się na *Zaloguj*.



Rysunek 4.6. Strona wyświetlana po wylogowaniu użytkownika

Jeżeli zamiast przygotowanej wcześniej strony wylogowania zostanie wyświetlona strona wylogowania witryny administracyjnej Django, sprawdź listę `INSTALLED_APPS` projektu i upewnij się, że wpis dotyczący aplikacji `django.contrib.admin` znajduje się po `account`. Oba wymienione szablony są umieszczone na tej samej względnej ścieżce dostępu i mechanizm wczytywania szablonów w Django po prostu użyje pierwszego znalezionego.

Widoki zmiany hasła

Użytkownikom witryny musimy zapewnić możliwość zmiany hasła po zalogowaniu się. Zintegrujemy więc oferowane przez framework uwierzytelniania Django widoki przeznaczone do obsługi procedury zmiany hasła.

Otwórz plik *urls.py* aplikacji `account` i umieść w nim poniższe wzorce adresów URL.

```
# Adresy URL przeznaczone do obsługi zmiany hasła.
path('password_change/',
    auth_views.PasswordChangeView.as_view(),
    name='password_change'),
path('password_change/done/',
    auth_views.PasswordChangeDoneView.as_view(),
    name='password_change_done'),
```

Widok `PasswordChangeView` zapewnia obsługę formularza pozwalającego na zmianę hasła, natomiast `PasswordChangeDoneView` wyświetla komunikat informujący o sukcesie po udanej operacji zmiany hasła przez użytkownika. Przystępujemy więc do przygotowania szablonu dla wymienionych widoków.

Dodaj nowy plik w katalogu *templates/registration* aplikacji `account` i nadaj mu nazwę *password_change_form.html*. Następnie w nowym pliku umieść poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Zmiana hasła{% endblock %}

{% block content %}
  <h1>Zmiana hasła</h1>
  <p>Wypełnij poniższy formularz, aby zmienić hasło.</p>
```

```
  <form action="." method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Zmień"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

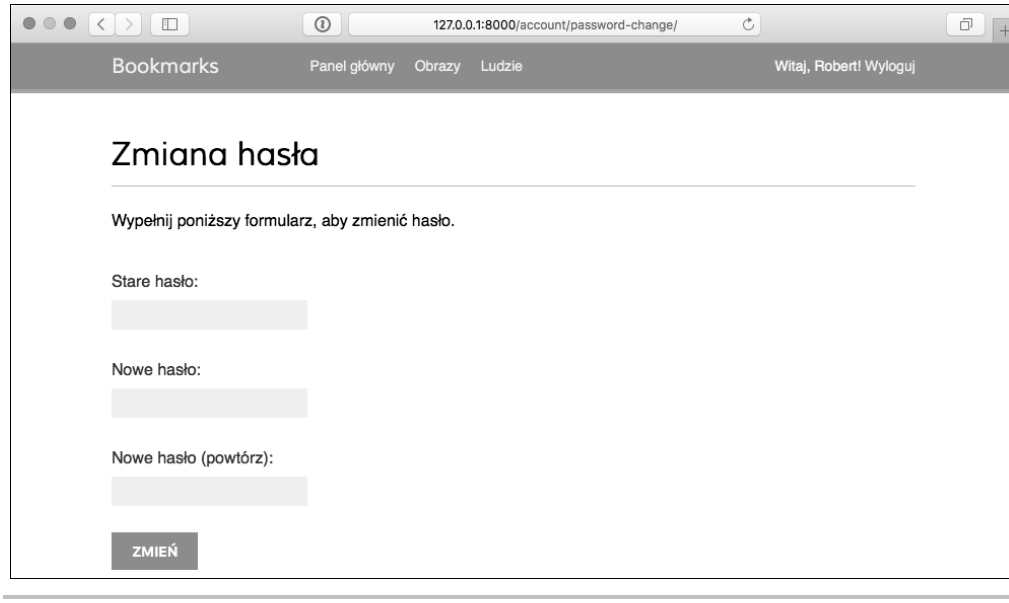
Przedstawiony szablon zawiera formularz przeznaczony do obsługi procedury zmiany hasła.

Teraz w tym samym katalogu utwórz kolejny plik i nadaj mu nazwę *password_change_done.html*. Następnie w nowym pliku umieść poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Hasło zostało zmienione{% endblock %}

{% block content %}
  <h1>Hasło zostało zmienione</h1>
  <p>Zmiana hasła zakończyła się powodzeniem.</p>
{% endblock %}
```

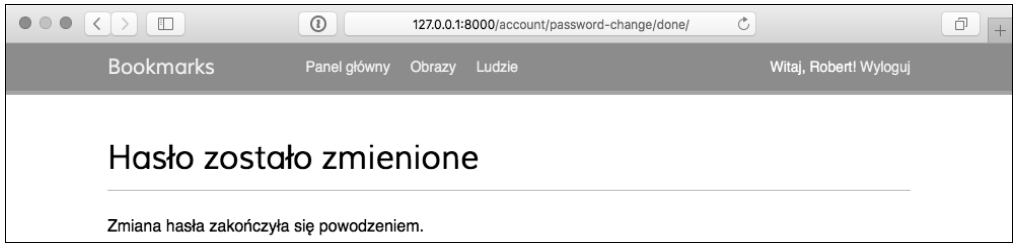
Ten szablon zawiera jedynie komunikat sukcesu wyświetlany, gdy przeprowadzona przez użytkownika operacja zmiany hasła zakończy się powodzeniem.

W przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/password_change/*. Jeżeli użytkownik nie jest zalogowany, nastąpi przekierowanie na stronę logowania. Po udanym uwierzytelnieniu zobaczysz formularz pozwalający na zmianę hasła pokazany na rysunku 4.7.



Rysunek 4.7. Formularz zmiany hasła

W wyświetlonym formularzu należy podać dotychczasowe hasło oraz nowe, a następnie kliknąć przycisk *Zmień*. Jeżeli operacja przebiegnie bez problemów, zostanie wyświetlona pokazana poniżej strona wraz z komunikatem informującym o sukcesie (rysunek 4.8).



Rysunek 4.8. Strona z komunikatem o pomyślnej zmianie hasła

Wyloguj się i zaloguj ponownie za pomocą nowego hasła, aby sprawdzić, że wszystko działa zgodnie z oczekiwaniami.

Widoki zerowania hasła

W pliku *urls.py* aplikacji `account` dodaj poniższe wzorce adresów URL dla widoków przeznaczonych do obsługi procedury zerowania hasła.

```
# Adresy URL przeznaczone do obsługi procedury zerowania hasła.
path('password_reset/',
    auth_views.PasswordResetView.as_view(),
    name='password_reset'),
path('password_reset/done/',
    auth_views.PasswordResetDoneView.as_view(),
    name='password_reset_done'),
path('reset/<uidb64>/<token>/',
    auth_views.PasswordResetConfirmView.as_view(),
    name='password_reset_confirm'),
path('reset/done/',
    auth_views.PasswordResetCompleteView.as_view(),
    name='password_reset_complete'),
```

Dodaj nowy plik w katalogu *templates/registration/* aplikacji `account` i nadaj mu nazwę *password_reset_form.html*. Następnie w utworzonym pliku umieść poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Zerowanie hasła{% endblock %}

{% block content %}
  <h1>Zapomniałeś hasła?</h1>
  <p>Podaj adres e-mail, aby zdefiniować nowe hasło.</p>
  <form action="." method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Wyślij e-mail"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

Teraz utwórz w tym samym katalogu kolejny plik, tym razem o nazwie *password_reset_email.html*. Następnie umieść w nim poniższy fragment kodu.

```
Otrzymaliśmy żądanie wyzerowania hasła dla użytkownika używającego adresu e-mail {{
↪email }}. Kliknij poniższe łącze:
{{ protocol }}://{{ domain }}{% url "password_reset_confirm"
uidb64=uid token=token %}
Twoja nazwa użytkownika: {{ user.get_username }}
```

Szablon *password_reset_email.html* zostanie użyty do wygenerowania wiadomości e-mail wysyłanej użytkownikowi, który chce przeprowadzić operację wyzerowania hasła. Wiadomość ta zawiera wygenerowany przez widok token, który jest potrzebny do wyzerowania hasła.

Utwórz w tym samym katalogu kolejny plik i nadaj mu nazwę *password_reset_done.html*. Następnie umieść w nim poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Zerowanie hasła{% endblock %}

{% block content %}
  <h1>Zerowanie hasła</h1>
  <p>Wysłaliśmy Ci wiadomość e-mail wraz z instrukcjami pozwalającymi na
  ↪zdefiniowanie nowego hasła.</p>
  <p>Jeżeli nie otrzymałeś tej wiadomości, to upewnij się, że w formularzu zerowania
  ↪hasła wpisałeś adres e-mail podany podczas zakładania konta użytkownika.</p>
{% endblock %}
```

Utwórz kolejny plik szablonu, nadaj mu nazwę *password_reset_confirm.html*, a następnie umieść w nim poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Zerowanie hasła{% endblock %}

{% block content %}
  <h1>Zerowanie hasła</h1>
  {% if validlink %}
    <p>Dwukrotnie podaj nowe hasło:</p>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Zmień hasło" /></p>
    </form>
  {% else %}
    <p>Łącze pozwalające na wyzerowanie hasła jest nieprawidłowe, ponieważ
    ↪prawdopodobnie zostało już wcześniej użyte. Musisz ponownie rozpocząć
    ↪procedurę zerowania hasła.</p>
  {% endif %}
{% endblock %}
```

W kodzie sprawdzamy, czy podane łącze jest prawidłowe. W tym celu weryfikowana jest zmienna `validlink`. Oferowany przez Django widok `PasswordResertConfirmView` ustawia zmienną i umieszcza ją w kontekście szablonu. Jeżeli łącze jest prawidłowe, wtedy wyświetlamy użytkownikowi formularz wyzerowania hasła. Użytkownicy mogą ustawić nowe hasło tylko wtedy, gdy dysponują prawidłowym łączem zerowania hasła.

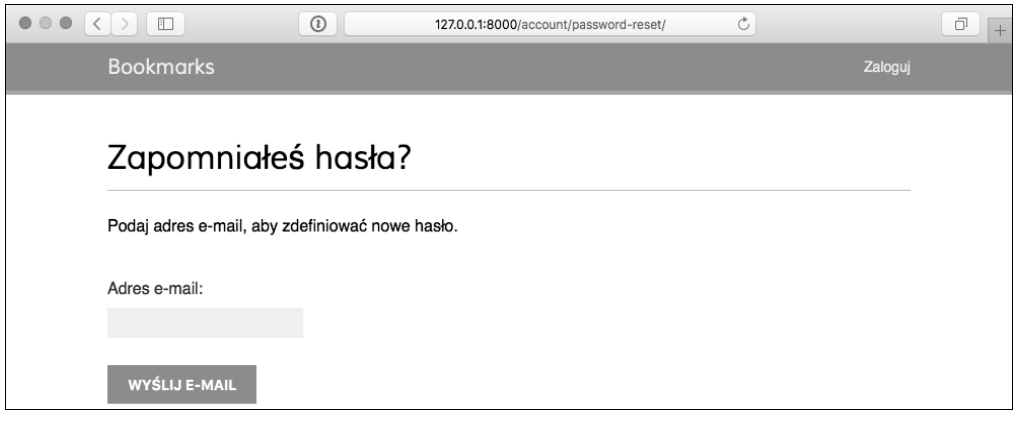
Utwórz kolejny plik szablonu i nadaj mu nazwę *password_reset_complete.html*. Następnie umieść w nim poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Zerowanie hasła{% endblock %}

{% block content %}
  <h1>Zerowanie hasła</h1>
  <p>Hasło zostało zdefiniowane. Możesz się już <a href="{% url "login"
%}">zalogować</a>.</p>
{% endblock %}
```

Na koniec przeprowadź edycję szablonu *registration/login.html* aplikacji `account` i dodaj poniższy fragment kodu po elemencie `<form>`.

```
<p><a href="{% url "password_reset" %}">Zapomniałeś hasła?</a></p>
```

Teraz w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/* i kliknij łącze *Zapomniałeś hasła?*. Powinieneś zobaczyć stronę podobną do pokazanej na rysunku 4.9.



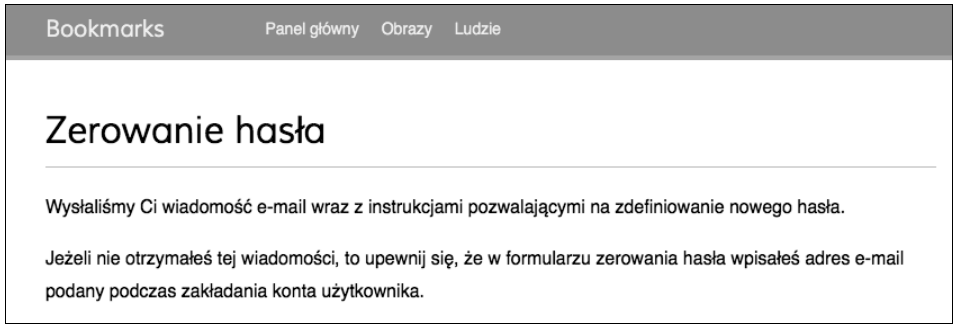
Rysunek 4.9. Formularz przypominania hasła

Na tym etapie w pliku *settings.py* projektu trzeba umieścić konfigurację serwera SMTP, aby umożliwić Django wysyłanie wiadomości e-mail. Procedura dodania tego rodzaju konfiguracji do projektu została omówiona w rozdziale 2., „Usprawnienie bloga za pomocą funkcji zaawansowanych". Jednak podczas pracy nad aplikacją można skonfigurować Django do przekazywania wiadomości e-mail na standardowe wyjście zamiast ich faktycznego wysyłania za pomocą serwera SMTP. Framework Django oferuje mechanizm backend do wyświetlania wiadomości e-mail w konsoli. Przeprowadź edycję pliku *settings.py* projektu i dodaj w nim poniższy wiersz kodu.

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Opcja `EMAIL_BACKEND` wskazuje na użycie klasy przeznaczonej do wysyłania wiadomości e-mail.

Wróć do przeglądarki internetowej, podaj adres e-mail istniejącego użytkownika i kliknij przycisk *Wyślij e-mail*. Powinieneś zobaczyć stronę pokazaną poniżej (rysunek 4.10).



Rysunek 4.10. Strona z informacją o wysłaniu danych potrzebnych do zdefiniowania nowego hasła

Spójrz na konsolę, na której został uruchomiony serwer programistyczny. Powinieneś zobaczyć wygenerowaną wiadomość e-mail.

```
Content-Type: text/plain; charset="utf-8"
MIME-VERSION: 1.0
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: user@domain.com
Date: Fri, 3 Jan 2020 14:35:08 -0000
Message-ID: <20171215143508.62996.55653@zenx.local>

Otrzymaliśmy żądanie wyzerowania hasła dla użytkownika używającego adresu e-mail
↪nazwa_użytkownika@nazwa_domeny.pl. Kliknij poniższe łącze:
http://127.0.0.1:8000/account/password-reset/confirm/MQ/45f9c3f30caafd523055fcc/
Twoja nazwa użytkownika: zenx
```

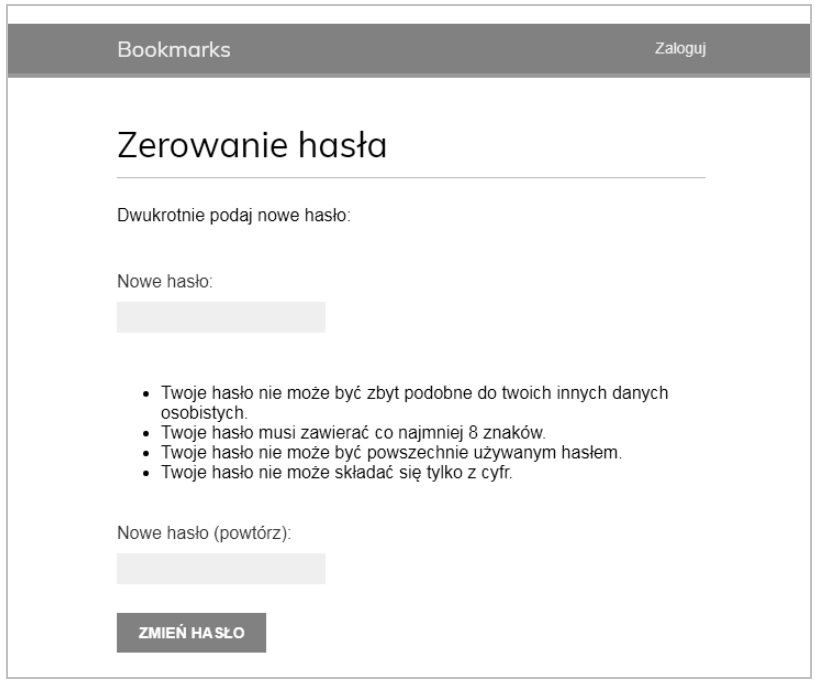
Ta wiadomość e-mail jest generowana za pomocą utworzonego wcześniej szablonu *password_*
↪reset_email.html. Adres URL pozwalający na przejście do strony zerowania hasła zawiera token dynamicznie wygenerowany przez Django.

Po otworzeniu w przeglądarce internetowej otrzymanego łącza przejdziesz na stronę pokazaną poniżej (rysunek 4.11).

To jest strona umożliwiająca użytkownikowi podanie nowego hasła; odpowiada ona szablonowi *password_reset_confirm.html*. W obu polach formularza wpisz nowe hasło, a następnie kliknij przycisk *Zmień hasło*. Django utworzy nowe zaszyfrowane hasło i zapisze je w bazie danych. Następnie zostanie wyświetlona pokazana poniżej strona wraz z komunikatem informującym o sukcesie operacji (rysunek 4.12).

Teraz użytkownik może zalogować się na swoje konto, podając nowe hasło.

Każdy token przeznaczony do ustawienia nowego hasła może być użyty tylko jednokrotnie. Jeżeli ponownie otworzysz w przeglądarce internetowej otrzymane łącze, zostanie wyświetlony komunikat informujący o nieprawidłowym tokenie.



Rysunek 4.11. Formularz z instrukcjami zerowania hasła

Rysunek 4.12. Strona z informacją o pomyślnej zmianie hasła

W ten sposób w projekcie zintegrowałeś widoki oferowane przez framework uwierzytelniania w Django. Wspomniane widoki są odpowiednie do użycia w większości sytuacji. Jednak zawsze możesz utworzyć własne widoki, jeśli potrzebna jest obsługa niestandardowego zachowania.

Wzorce adresów URL uwierzytelniania, które właśnie utworzyliśmy, udostępnia również framework Django. Możesz ująć w komentarz wzorce adresów URL uwierzytelniania, które dodaliśmy do pliku *urls.py* aplikacji `account`, i zamiast tego dodać aplikację `django.contrib.auth.urls` tak, jak pokazałem poniżej.

```
from django.urls import path, include
#...

    urlpatterns = [
        #...
        path('', include('django.contrib.auth.urls')),
]
```

Więcej informacji na temat wzorców adresów URL uwierzytelniania można znaleźć na stronie *https://github.com/django/django/blob/stable/2.0.x/django/contrib/auth/urls.py*.

Rejestracja użytkownika i profile użytkownika

Istniejący użytkownicy mogą się zalogować, wylogować, zmienić hasło lub je wyzerować, jeśli zapomnieli, jakie było. Musimy teraz przygotować widok pozwalający nowym odwiedzającym witrynę na założenie w niej konta użytkownika.

Rejestracja użytkownika

Przystępujemy do utworzenia prostego widoku pozwalającego odwiedzającemu na zarejestrowanie się w naszej witrynie internetowej. Zaczniemy od formularza, w którym nowy użytkownik wprowadzi nazwę użytkownika, swoje imię i nazwisko oraz hasło.

Przeprowadź edycję pliku *forms.py* w katalogu aplikacji `account` i umieść w nim poniższy fragment kodu.

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

class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Hasło',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Powtórz hasło',
                                widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ('username', 'first_name', 'email')

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Hasła nie są identyczne.')
        return cd['password2']
```

Utworzyliśmy formularz modelu (klasa `ModelForm`) dla modelu `User`. W przygotowanym formularzu będą uwzględnione jedynie pola `username`, `first_name` i `email`. Wartości wymienionych pól będą weryfikowane na podstawie odpowiadających im kolumn modelu. Jeśli np. użytkownik

wybierze już istniejącą nazwę użytkownika, otrzyma błąd w trakcie weryfikacji formularza, ponieważ pole `username` jest zdefiniowane z ustawieniem `unique=True`.

Dodaliśmy dwa nowe pola `password` i `password2` przeznaczone do zdefiniowania hasła i jego potwierdzenia. Ponadto zdefiniowaliśmy metodę `clean_password2()` odpowiedzialną za porównanie obu wpisanych haseł. Jeżeli nie są takie same, formularz będzie uznany za nieprawidłowy. Ta operacja sprawdzenia nowego hasła jest przeprowadzana podczas weryfikacji formularza za pomocą jego metody `is_valid()`. Istnieje możliwość dostarczenia metody `clean_<nazwa_pola>()` dla dowolnego pola formularza w celu wyczyszczenia jego wartości lub zgłoszenia błędu weryfikacji formularza dla określonego pola. Formularze zawierają także ogólną metodę `clean()` przeznaczoną do sprawdzenia całego formularza, co okazuje się użyteczne podczas weryfikacji pól zależnych wzajemnie od siebie.

Django oferuje również formularz `UserCreationForm` gotowy do natychmiastowego użycia. Znajdziesz go w `django.contrib.auth.forms`, a sam formularz jest bardzo podobny do utworzonego przez nas wcześniej.

Przeprowadź edycję pliku *views.py* aplikacji `account` i umieść w nim poniższy fragment kodu.

```
from .forms import LoginForm, UserRegistrationForm

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Utworzenie nowego obiektu użytkownika; jednak jeszcze nie zapisujemy go w bazie danych.
            new_user = user_form.save(commit=False)
            # Ustawienie wybranego hasła.
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Zapisanie obiektu User.
            new_user.save()
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

Widok pozwalający na utworzenie nowego konta użytkownika jest całkiem prosty. Zamiast zapisywać wprowadzone przez użytkownika hasło w postaci zwykłego tekstu, wykorzystujemy metodę `set_password()` modelu `User`, która ze względów bezpieczeństwa szyfruje hasło.

Teraz przeprowadź edycję pliku *urls.py* aplikacji `account` i dodaj w nim poniższy wzorzec adresu URL.

```
path('register/', views.register, name='register'),
```

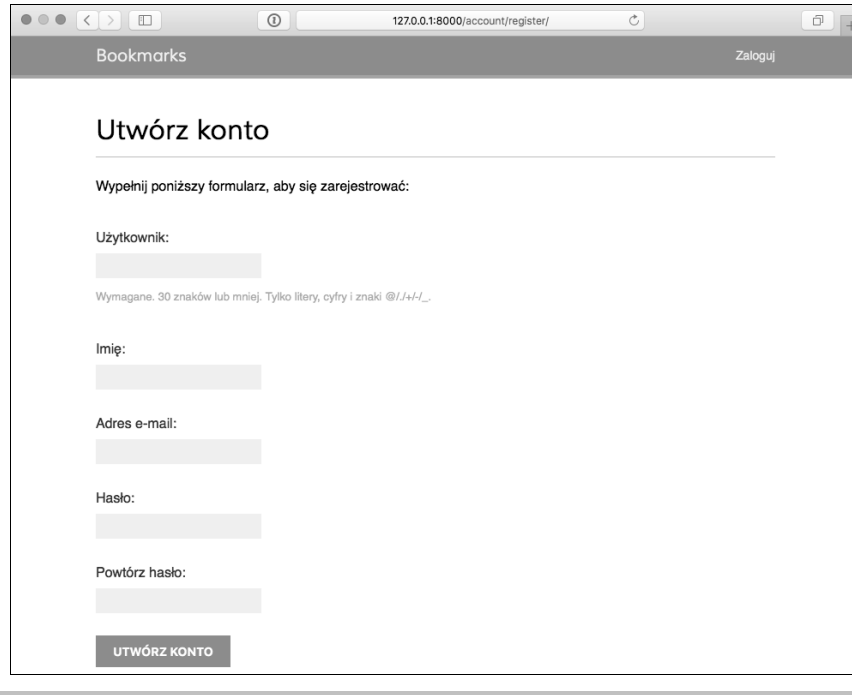
Na koniec utwórz nowy szablon w katalogu szablonów aplikacji `account`, nadaj mu nazwę *register.html*, a następnie umieść w nim poniższy kod.

```
{% extends "base.html" %}

{% block title %}Utwórz konto{% endblock %}

{% block content %}
  <h1>Utwórz konto</h1>
  <p>Wypełnij poniższy formularz, aby się zarejestrować:</p>
<form action="." method="post">
    {{ user_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Utwórz konto"></p>
  </form>
{% endblock %}
```

Do tego samego katalogu dodaj nowy plik szablonu o nazwie *register_done.html*. Następnie umieść w nim poniższy fragment kodu.

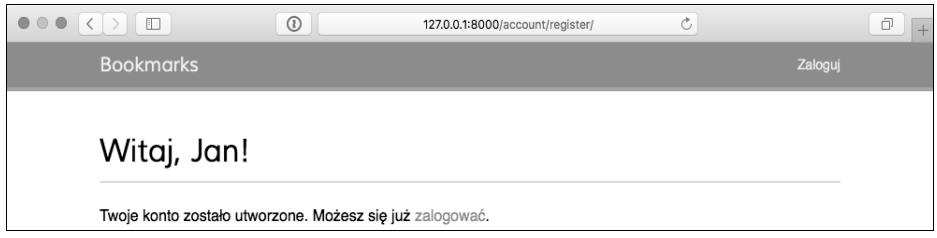
```
{% extends "base.html" %}

{% block title %}Witaj{% endblock %}

{% block content %}
  <h1>Witaj, {{ new_user.first_name }}!</h1>
  <p>Twoje konto zostało utworzone. Możesz się już <a
href="{% url "login" %}">zalogować</a>.</p>
{% endblock %}
```

Teraz w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/register/*. Powinieneś zobaczyć wyświetloną stronę rejestracji nowego użytkownika (rysunek 4.13).



Rysunek 4.13. Formularz tworzenia konta

Podaj informacje potrzebne do utworzenia nowego konta użytkownika i kliknij przycisk *Utwórz konto*. Jeżeli wszystkie pola zostały wypełnione prawidłowo, zostanie wyświetlona strona wraz z komunikatem informującym o pomyślnym utworzeniu nowego konta użytkownika (rysunek 4.14).



Rysunek 4.14. Strona z komunikatem o pomyślnym utworzeniu konta

Kliknij łącze *Zaloguj*, a następnie podaj dane uwierzytelniające utworzonego przed chwilą użytkownika, aby potwierdzić, że możesz uzyskać dostęp do nowego konta.

Teraz do szablonu logowania musimy dodać łącze pozwalające na rejestrację użytkownika. Przeprowadź edycję szablonu *registration/login.html* i poniższy wiersz kodu

```
<p>Wypełnij poniższy formularz, aby się zalogować:</p>
```

zastąp następującym

```
<p>Wypełnij poniższy formularz, aby się zalogować. Jeżeli nie masz jeszcze
konta, możesz je utworzyć <a href="{% url "register" %}">tutaj</a>.</p>
```

W ten sposób do strony rejestracji nowego konta użytkownika można przejść bezpośrednio ze strony logowania.

Rozbudowa modelu User

Podczas pracy z kontami użytkowników przekonasz się, że model `User` oferowany przez framework uwierzytelniania Django sprawdza się w większości przypadków. Jednak model `User` jest dostarczany wraz z jedynie najbardziej podstawowymi kolumnami. Dlatego też prawdopodobnie będzie trzeba niekiedy rozbudować model o możliwość przechowywania dodatkowych danych. Najlepszym sposobem będzie przygotowanie modelu profilu zawierającego wszystkie dodatkowe kolumny oraz związek typu „jeden do jednego" z dostarczanym przez Django modelem `User`. Relacja „jeden do jednego" jest podobna do stosowania pola `ForeignKey` z parametrem `unique = True`. Druga strona relacji to niejawny związek „jeden do jednego" z powiązanym modelem zamiast menedżera wielu elementów. Z każdej strony relacji pobieramy jeden powiązany obiekt.

Przeprowadź edycję pliku *models.py* aplikacji `account` i umieść w nim poniższy fragment kodu.

```
from django.db import models
from django.conf import settings

class Profile(models.Model):
```

```
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                                on_delete=models.CASCADE)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='users/%Y/%m/%d',
                              blank=True)

    def __str__(self):
        return 'Profil użytkownika {}.'.format(self.user.username)
```

Aby kod był generyczny, do pobrania modelu użytkownika używamy funkcji `get_user_model()`. Następnie, podczas definiowania relacji z tym modelem zamiast bezpośredniego odwoływania się do modelu `User` można korzystać z opcji `AUTH_USER_MODEL`. Więcej informacji na ten temat można przeczytać na stronie *https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#django.contrib.auth.get_user_model*.

Kolumna `user` definiuje relację typu „jeden do jednego" i pozwala na powiązanie profilu z użytkownikiem. Użyliśmy klauzuli `CASCADE` dla parametru `on_delete`, aby po usunięciu użytkownika został również usunięty powiązany z nim profil. Zdjęcie użytkownika jest przechowywane w kolumnie `ImageField`. W celu zarządzania obrazami konieczne będzie zainstalowanie pakietu Pythona `Pillow`. Instalacja pakietu `Pillow` zostanie przeprowadzona po wydaniu w powłoce poniższego polecenia.

```
pip install Pillow==7.0.0
```

Aby na serwerze programistycznym Django umożliwić obsługę plików multimedialnych przekazywanych przez użytkowników, w pliku *settings.py* projektu należy dodać poniższe wiersze kodu.

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

Opcja `MEDIA_URL` wskazuje bazowy adres URL określający lokalizację przeznaczoną do przechowywania plików multimedialnych przekazywanych przez użytkowników. Natomiast opcja `MEDIA_ROOT` określa lokalną ścieżkę dostępu dla tych plików. Ścieżka dostępu jest budowana dynamicznie względem projektu, co zapewnia możliwość generycznego działania kodu.

Teraz przeprowadź edycję pliku głównego *urls.py* projektu `bookmarks` i w następujący sposób zmodyfikuj znajdujący się w nim kod.

```
from django.conf.urls import path, include
from django.contrib import admin
from django.conf import settings
from django.conf.urls.static import static

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

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

W ten sposób serwer programistyczny Django stał się odpowiedzialny za obsługę plików multimedialnych podczas prac nad aplikacją (tzn. wtedy, gdy opcja `DEBUG` jest ustawiona na `True`).

> Funkcja pomocnicza `static()` jest odpowiednia do stosowania w środowisku programistycznym, ale na pewno nie w produkcyjnym. Serwowanie statycznych plików przez Django jest bardzo niewydajne. Pamiętaj, aby w środowisku produkcyjnym nigdy nie udostępniać plików statycznych za pomocą Django. Sposób serwowania statycznych plików w środowisku produkcyjnym omówiono w rozdziale 14., „Wdrażanie aplikacji".

Przejdź do powłoki i wydaj następujące polecenie, które spowoduje utworzenie migracji bazy danych dla nowego modelu.

```
python manage.py makemigrations
```

Powinieneś otrzymać następujący wynik.

```
Migrations for 'account':
  0001_initial.py:
    - Create model Profile
```

Kolejnym krokiem jest zsynchronizowanie bazy danych za pomocą poniższego polecenia.

```
python manage.py migrate
```

Wygenerowany wynik będzie zawierał m.in. następujący wiersz.

```
Applying account.0001_initial... OK
```

Przeprowadź edycję pliku *admin.py* aplikacji `account` i zarejestruj model `Profile` w witrynie administracyjnej, co pokazałem poniżej.

```
from django.contrib import admin
from .models import Profile

@admin.register(Profile)
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'date_of_birth', 'photo']
```

Ponownie uruchom serwer programistyczny za pomocą polecenia `python manage.py runserver`. Teraz będziesz mógł zobaczyć model `Profile` w witrynie administracyjnej projektu, co pokazałem na rysunku 4.15.

Rysunek 4.15. Blok Account głównej strony witryny administracyjnej

Teraz zajmiemy się umożliwieniem użytkownikom edycji profilu w witrynie internetowej. Dodaj poniższe formularze modelu do pliku *forms.py* aplikacji `account`.

```
from .models import Profile

class UserEditForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email')

class ProfileEditForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('date_of_birth', 'photo')
```

Oto krótkie omówienie dodanych formularzy.

- `UserEditForm`. Formularz pozwala użytkownikowi na edycję imienia, nazwiska i adresu e-mail. Wymienione informacje są przechowywane we wbudowanym w Django modelu `User`.
- `ProfileEditForm`. Formularz pozwala użytkownikowi na edycję danych dodatkowych, które zostaną zapisane w modelu `Profile`. Użytkownik będzie mógł podać datę urodzenia i wczytać obraz (tzw. awatar) dla swojego profilu.

Przeprowadź edycję pliku *views.py* aplikacji `account` i zaimportuj model `Profile` w następujący sposób.

```
from .models import Profile
```

Następnie dodaj poniższe wiersze kodu do widoku `register`; umieść je pod funkcją `new_user.save()`.

```
# Utworzenie profilu użytkownika.
profile = Profile.objects.create(user=new_user)
```

Kiedy użytkownik rejestruje się w witrynie, tworzymy powiązany z nim pusty model. Dla istniejących użytkowników obiekty `Profile` musimy utworzyć ręcznie za pomocą witryny administracyjnej Django.

Teraz umożliwimy użytkownikowi edycję profilu. Dodaj poniższy fragment kodu do tego samego pliku (*views.py*).

```
from .forms import LoginForm, UserRegistrationForm, UserEditForm, ProfileEditForm

@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile,
                                    data=request.POST,
                                    files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
    else:
        user_form = UserEditForm(instance=request.user)
```

```
        profile_form = ProfileEditForm(instance=request.user.profile)
    return render(request,
                  'account/edit.html',
                  {'user_form': user_form,
                   'profile_form': profile_form})
```

Używamy dekoratora `login_required`, ponieważ w celu przeprowadzenia edycji profilu użytkownik musi być uwierzytelniony. W takim przypadku korzystamy z dwóch modeli formularzy, czyli `UserEditForm` przeznaczonego do przechowywania danych wbudowanego modelu `User` i `ProfileEditForm` przeznaczonego do przechowywania dodatkowych danych profilu. Aby zweryfikować dane wysłane w formularzu, sprawdzamy, czy wartością zwrotną metody `is_valid()` w obu wymienionych formularzach jest `True`. Jeżeli tak, zawartość obu formularzy zapisujemy w celu uaktualnienia odpowiedniego obiektu w bazie danych.

Dodaj poniższy wzorzec adresu URL do pliku *urls.py* aplikacji `account`.

```
path('edit/', views.edit, name='edit'),
```

Na koniec w katalogu *templates/account/* utwórz nowy szablon dla widoku i nadaj mu nazwę *edit.html*. Następnie w tym pliku umieść poniższy fragment kodu.

```
{% extends "base.html" %}

{% block title %}Edycja konta{% endblock %}

{% block content %}
  <h1>Edycja konta</h1>
  <p>Ustawienia konta możesz zmienić za pomocą poniższego formularza:</p>
  <form action="." method="post" enctype="multipart/form-data">
    {{ user_form.as_p }}
    {{ profile_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Zapisz zmiany"></p>
  </form>
{% endblock %}
```

Aby umożliwić przekazywanie plików, w formularzu musi znaleźć się opcja `enctype="multipart/form-data"`. Wykorzystujemy tylko jeden formularz HTML do wysłania obu formularzy Django, czyli `user_form` i `profile_form`.

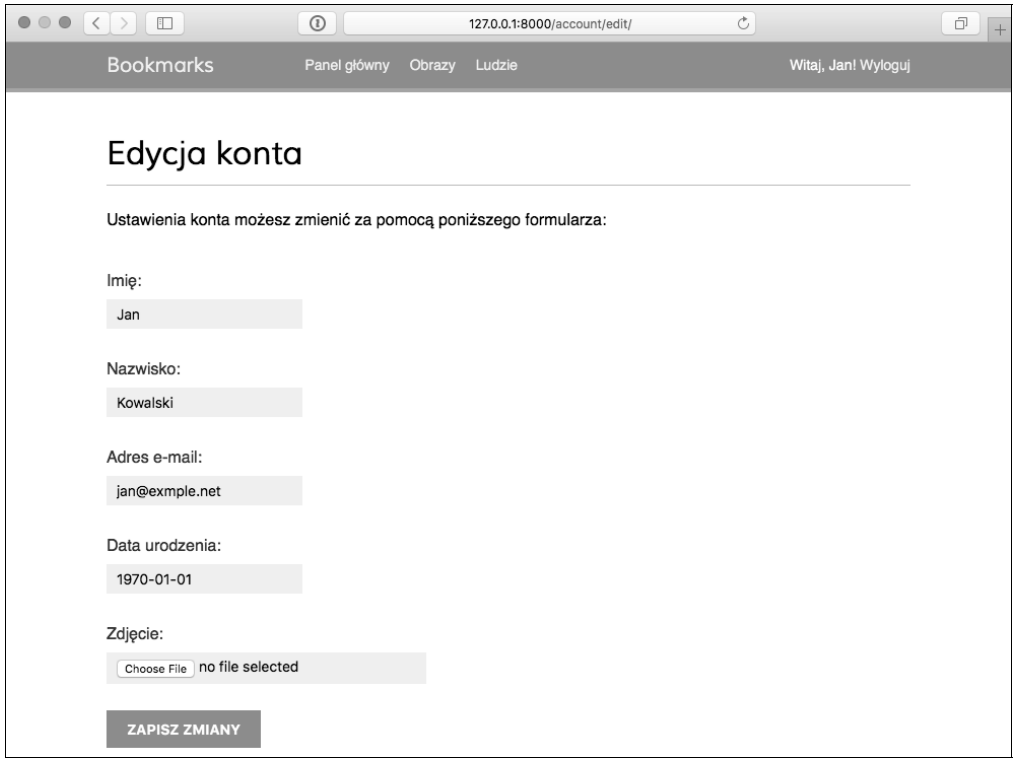
Zarejestruj nowego użytkownika i przejdź pod adres *http://127.0.0.1:8000/account/edit/* w przeglądarce internetowej. Powinieneś zobaczyć stronę pokazaną na rysunku 4.16.

Teraz możemy zmodyfikować stronę panelu głównego i umieścić na niej łącza prowadzące do stron pozwalających na edycję profilu i zmianę hasła. Otwórz szablon *account/dashboard.html* i poniższy wiersz kodu

```
<p>Witaj w panelu głównym.</p>
```

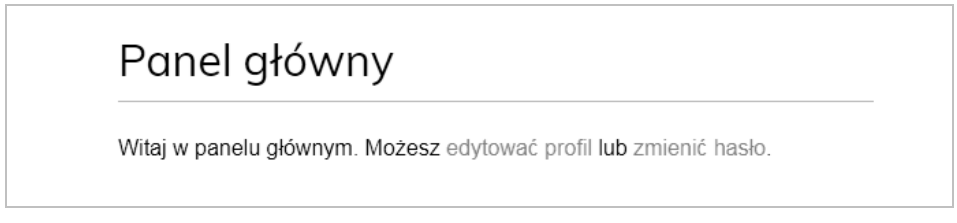
zastąp następującym

```
<p>Witaj w panelu głównym. Możesz <a href="{% url "edit" %}">edytować
profil</a> lub <a href="{% url "password_change" %}">zmienić hasło</a>.</p>
```



Rysunek 4.16. Formularz edycji profilu

Po wprowadzonych zmianach użytkownik będzie miał z poziomu panelu głównego dostęp do formularza umożliwiającego edycję profilu. Otwórz w przeglądarce stronę *http://127.0.0.1:8000/account/* i przetestuj nowe łącze edycji profilu użytkownika (rysunek 4.17).



Rysunek 4.17. Zawartość strony panelu głównego razem z łączami do edycji profilu i zmiany hasła

Użycie własnego modelu User

Django oferuje możliwość całkowitego zastąpienia modelu `User` własnym. Klasa modelu powinna dziedziczyć po klasie `AbstractUser` zapewniającej pełną implementację użytkownika domyślnego jako modelu abstrakcyjnego. Więcej informacji na temat tego rodzaju podejścia znajdziesz na stronie *https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#substit*
↪uting-a-custom-user-model.

Zastosowanie własnego modelu użytkownika daje znacznie większą elastyczność, choć może oznaczać również nieco większą trudność podczas integracji z innymi aplikacjami, które współdziałają ze standardowym modelem `User`.

Użycie frameworka komunikatów

Podczas obsługi akcji podejmowanych przez użytkowników może wystąpić konieczność informowania ich o skutkach podjętych przez nich działań. Django oferuje wbudowany framework pozwalający na wyświetlanie użytkownikom jednorazowych powiadomień.

Framework jest dostarczany przez aplikację `django.contrib.messages`, która została domyślnie umieszczona na liście `INSTALLED_APPS` w pliku *settings.py* podczas tworzenia nowego projektu za pomocą polecenia `python manage.py startproject`. Zwróć uwagę na fakt, że plik ustawień zawiera oprogramowanie pośredniczące o nazwie `django.contrib.messages.middleware.Message` `↪Middleware` umieszczone na liście `MIDDLEWARE` ustawień projektu.

Framework komunikatów zapewnia prosty sposób dodawania komunikatów przeznaczonych do wyświetlenia użytkownikom. Wspomniane komunikaty są przechowywane domyślnie w pliku cookie (jeśli go nie ma, to w pamięci trwałej sesji) i wyświetlane podczas następnego żądania wykonywanego przez danego użytkownika. Framework komunikatów można wykorzystać w widokach — w tym celu należy zaimportować odpowiedni moduł — a następnie można już dodawać nowe komunikaty za pomocą prostych skrótów, co pokazałem poniżej.

```
from django.contrib import messages
messages.error(request, 'Wystąpił pewien problem!')
```

Nowe wiadomości możesz tworzyć z wykorzystaniem metody `add_message()` lub dowolnej z wymienionych poniżej metod skrótów.

- `success()`. Komunikat sukcesu wyświetlany po zakończeniu operacji powodzeniem.
- `info()`. Ogólny komunikat informacyjny.
- `warning()`. Wystąpił pewien problem, ale jeszcze nie mamy do czynienia z niepowodzeniem.
- `error()`. Akcja nie zakończyła się powodzeniem lub doszło do niepowodzenia.
- `debug()`. Komunikaty debugowania, które będą usunięte lub zignorowane w środowisku produkcyjnym.

Przechodzimy teraz do wyświetlania komunikatów użytkownikom. Ponieważ framework komunikatów jest stosowany globalnie w projekcie, komunikaty możemy wyświetlać użytkownikowi za pomocą szablonu bazowego. Otwórz więc plik *base.html* i poniższy fragment kodu umieść pomiędzy elementem `<div>` o identyfikatorze `header` a elementem `<div>` o identyfikatorze `content`.

```
{% if messages %}
  <ul class="messages">
    {% for message in messages %}
      <li class="{{ message.tags }}">
        {{ message|safe }}
          <a href="#" class="close">x</a>
```

```
        </li>
      {% endfor %}
    </ul>
{% endif %}
```

Framework komunikatów zawiera procesor kontekstu `django.contrib.messages.context_processor.messages` dodający zmienną `messages` do kontekstu żądania. Można go znaleźć na liście `context_processors` ustawienia `TEMPLATES` w Twoim projekcie. Zmiennej `messages` można używać w szablonach do wyświetlania użytkownikowi aktualnych komunikatów.

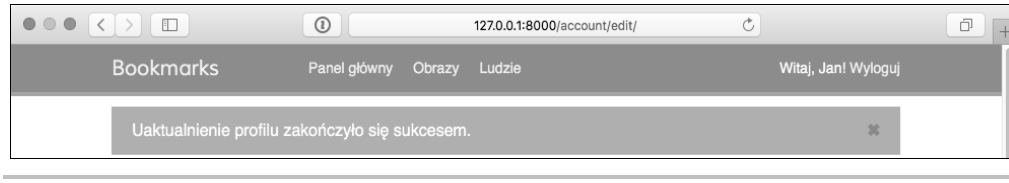
> Procesor kontekstu to funkcja Pythona, która pobiera jako argument obiekt `request` i zwraca słownik, który jest dodawany do kontekstu żądania. Sposób tworzenia własnych procesorów kontekstu omówię w rozdziale 7., „Utworzenie sklepu internetowego".

Teraz zmodyfikujmy widok edycji profilu, aby wykorzystać w nim framework komunikatów. Przeprowadź edycję pliku *views.py* naszej aplikacji i zmień kod widoku `edit` na następujący.

```
from django.contrib import messages

@login_required
def edit(request):
    if request.method == 'POST':
    #...
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, 'Uaktualnienie profilu '\
                                      'zakończyło się sukcesem.')
        else:
            messages.error(request, 'Wystąpił błąd podczas uaktualniania profilu.')
    else:
        user_form = UserEditForm(instance=request.user)
        #...
```

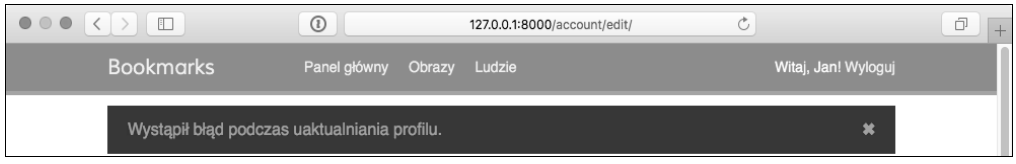
Dodaliśmy komunikat informujący o sukcesie wyświetlany, gdy operacja zmiany profilu zakończy się powodzeniem. Jeżeli którykolwiek formularz będzie wypełniony nieprawidłowo, nastąpi wyświetlenie komunikatu o błędzie.

W przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/edit/*, a następnie przeprowadź edycję profilu. Jeśli profil zostanie pomyślnie zaktualizowany, powinieneś zobaczyć komunikat informujący o sukcesie, co pokazałem na rysunku 4.18.



Rysunek 4.18. Komunikat o pomyślnej edycji profilu

Gdy formularz jest nieprawidłowy, gdy np. format daty wprowadzonej w polu Data urodzenia jest niepoprawny, Django wyświetli komunikat o błędzie (rysunek 4.19).



Rysunek 4.19. Komunikat o błędzie aktualizacji profilu

Więcej informacji na temat frameworka komunikatów można znaleźć na stronie *https://docs.djangoproject.com/en/2.0/ref/contrib/messages*.

Implementacja własnego mechanizmu uwierzytelniania

Django pozwala na uwierzytelnianie względem wielu różnych źródeł. Opcja `AUTHENTICATION_BACKENDS` to lista mechanizmów uwierzytelniania, które mogą być stosowane w projekcie. Domyślnie opcja ta ma następującą wartość.

```
['django.contrib.auth.backends.ModelBackend',]
```

Domyślny model o nazwie `ModelBackend` uwierzytelnia użytkowników na podstawie bazy danych za pomocą modelu `User` dostarczanego przez `django.contrib.auth`. Takie rozwiązanie okazuje się wystarczające w większości projektów. Istnieje jednak możliwość przygotowania własnego mechanizmu uwierzytelniania i wykorzystania innych źródeł, np. usług katalogowych (LDAP) bądź też innego systemu.

Więcej informacji dotyczących dostosowania mechanizmu uwierzytelniania do własnych potrzeb znajdziesz na stronie *https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#other-authentication-sources*.

Jeśli tylko użyjesz funkcji `authenticate()` zdefiniowanej w `django.contrib.auth`, Django spróbuje po kolei uwierzytelnić użytkownika względem każdego mechanizmu zdefiniowanego na liście `AUTHENTICATION_BACKENDS` aż do chwili, gdy którakolwiek próba zakończy się powodzeniem. Jeżeli nie uda się uwierzytelnić użytkownika za pomocą żadnego mechanizmu, wtedy pozostanie on niezalogowany w witrynie.

Django pozwala na bardzo łatwe zdefiniowanie własnego mechanizmu uwierzytelniania. Sama procedura opiera się na klasie dostarczającej dwie wymienione poniżej metody.

- `authenticate()`. Metoda pobiera obiekt `request` i poświadczenia użytkownika jako parametry. Wartością zwrotną jest obiekt `user` odpowiadający przekazanym poświadczeniom, o ile są one prawidłowe, i `None` w przeciwnym przypadku. Parametr `request` to obiekt klasy `HttpRequest` lub `None` (jeśli go nie przekazano do metody `authenticate()`).
- `get_user()`. Metoda pobiera parametr w postaci identyfikatora użytkownika i zwraca odpowiadający mu obiekt `User`.

Przygotowanie własnego mechanizmu uwierzytelnienia jest naprawdę łatwe i sprowadza się do utworzenia klasy Pythona implementującej obie wymienione metody. Zdefiniujemy teraz własny mechanizm uwierzytelniania, aby pozwolić użytkownikom na zalogowanie się do witryny za pomocą adresu e-mail zamiast nazwy użytkownika.

Utwórz nowy plik w katalogu aplikacji `account` i nadaj mu nazwę *authentication.py*. Następnie umieść w nim poniższy fragment kodu.

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

class EmailAuthBackend(object):
    """
    Uwierzytelnia użytkownika na podstawie adresu e-mail.
    """
    def authenticate(self, request, username=None, password=None):
        try:
            user = User.objects.get(email=username)
            if user.check_password(password):
                return user
            return None
        except User.DoesNotExist:
            return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

To jest prosty mechanizm uwierzytelniania. Metoda `authenticate()` pobiera obiekt `request` i parametry opcjonalne `username` i `password`. Wprawdzie można by użyć innych parametrów, ale zdecydowałem się na wymienione, aby ułatwić współpracę tego mechanizmu z widokami frameworka uwierzytelniania. Oto dokładne omówienie sposobu działania powyższego kodu.

- `authenticate()`. Próbujemy pobrać użytkownika na podstawie podanego adresu e-mail oraz sprawdzamy hasło za pomocą wbudowanej metody `check_password()` modelu `User`. Wymieniona metoda dla podanego przez użytkownika hasła generuje wartość hash, którą następnie porównuje z wartością przechowywaną w bazie danych.
- `get_user()`. Metoda pobiera użytkownika na podstawie identyfikatora podanego w parametrze `user_id`. Django wykorzystuje mechanizm, który uwierzytelnił użytkownika, do pobrania odpowiadającego mu obiektu `User` na czas trwania sesji użytkownika.

Przeprowadź edycję pliku *settings.py* projektu i dodaj poniższe ustawienia.

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

Pozostawiamy domyślny mechanizm `ModelBackend` używany do uwierzytelnienia użytkownika na podstawie podanych przez niego nazwy użytkownika i hasła. Opracowany mechanizm uwierzytelnienia na podstawie adresu e-mail dodajemy jako drugi.

Teraz w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/*. Ponieważ Django będzie próbował uwierzytelnić użytkownika za pomocą każdego mechanizmu, więc do witryny możesz się zalogować, podając nazwę użytkownika lub adres e-mail. Poświadczenia użytkownika będą sprawdzane z wykorzystaniem mechanizmu uwierzytelniania `ModelBackend`, a jeśli nie zostanie zwrócony żaden użytkownik, poświadczenia będą sprawdzane przy użyciu zdefiniowanego przed chwilą mechanizmu uwierzytelniania `EmailAuthBackend`.

> Kolejność umieszczenia mechanizmów uwierzytelniania na liście `AUTHENTICATION_BACKENDS` ma znaczenie. Jeżeli te same dane uwierzytelniające są prawidłowe dla wielu mechanizmów, Django zakończy próby na pierwszym mechanizmie, za pomocą którego uda się uwierzytelnić użytkownika.

Dodanie do witryny uwierzytelnienia za pomocą innej witryny społecznościowej

Być może do budowanej witryny internetowej zechcesz dodać obsługę uwierzytelnienia za pomocą innego serwisu społecznościowego, takiego jak Facebook, Twitter lub Google. Dostępny jest moduł Pythona o nazwie `Python-social-auth`, który ułatwia proces implementacji tego rodzaju uwierzytelnienia. Dzięki wykorzystaniu wymienionego modułu możesz zezwolić użytkownikowi na logowanie się do witryny internetowej za pomocą konta, które założył w innym serwisie.

Uwierzytelnianie za pomocą serwisów społecznościowych to powszechnie wykorzystywana własność ułatwiająca przeprowadzanie procesu uwierzytelniania. Kod modułu `Python-social-auth` znajdziesz na stronie *https://github.com/omab/python-social-auth*.

Sam moduł jest dostarczany wraz z różnymi mechanizmami uwierzytelniania dla odmiennych frameworków Pythona, w tym także dla Django. Aby zainstalować moduł za pomocą menedżera `pip`, przejdź do powłoki i wydaj poniższe polecenie.

```
pip install social-auth-app-django==3.1.0
```

Następnie umieść `social.apps.django_app.default` na liście `INSTALLED_APPS` w pliku *settings.py* projektu, tak jak pokazałem poniżej.

```
INSTALLED_APPS = [
    #...
    'social_django',
]
```

W ten sposób dodajemy do projektu Django aplikację `default` modułu `python-social-auth`. Teraz wydaj poniższe polecenie, aby przeprowadzić synchronizację modeli `python-social-auth` z bazą danych.

```
$ python manage.py migrate
```

Powinieneś zobaczyć zastosowanie migracji dla aplikacji `default`.

```
Applying social_django.0001_initial... OK
Applying social_django.0002_add_related_name... OK
...
Applying social_django.0008_partial_timestamp... OK
```

Moduł `Python-social-auth` oferuje obsługę mechanizmu uwierzytelniania wielu usług. Pełną listę znajdziesz na stronie *https://python-social-auth.readthedocs.org/en/latest/backends/index.html#supported-backends*.

W rozdziale zaimplementujemy uwierzytelnienie w naszej aplikacji za pomocą konta użytkownika w serwisach Facebook, Twitter i Google.

Na początek konieczne jest dodanie do projektu wzorca adresu URL dla logowania za pomocą serwisu społecznościowego. Otwórz główny plik *urls.py* w projekcie `bookmarks` i umieść w nim wzorce URL `social_django` tak, jak pokazałem poniżej.

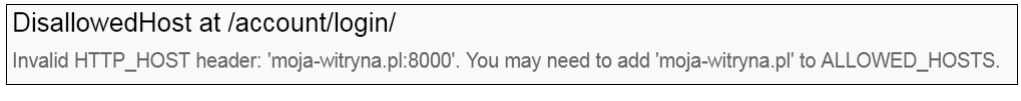
```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
        include('social_django.urls', namespace='social')),
]
```

Wiele usług społecznościowych nie pozwala po pomyślnym uwierzytelnieniu na przekierowanie na adres `127.0.0.1` lub `localhost`. Oczekują one podania nazwy domeny. W systemach Linux lub OS X rozwiązaniem jest edycja pliku */etc/hosts* polegająca na dodaniu wiersza mapującego dowolnie wybraną domenę na adres komputera lokalnego. W omawianym przykładzie decydujemy się na domenę *moja-witryna.pl*, więc polecenie ma następującą postać.

```
127.0.0.1 moja-witryna.pl
```

Powyższe polecenie oznacza, że po podaniu adresu *moja-witryna.pl* zostanie wskazany komputer lokalny. Jeżeli używasz Windows, plik *hosts* znajdziesz w *C:\Winwows\System32\Drivers\etc\hosts*.

Aby sprawdzić, czy zdefiniowane przekierowanie działa zgodnie z oczekiwaniami, uruchom serwer programistyczny za pomocą polecenia `python manage.py runserver` i w przeglądarce internetowej przejdź pod adres *http://moja-witryna.pl:8000/account/login/*. Wyświetli się poniższy komunikat o błędzie (rysunek 4.20).



Rysunek 4.20. Komunikat nagłówka o nieprawidłowym hoście

Django zarządza hostami, które mogą obsługiwać Twoją aplikację, używając ustawienia ALLOWED_HOSTS. Jest to środek bezpieczeństwa zapobiegający atakom z wykorzystaniem nagłówków hosta HTTP. Django zezwoli na serwowanie aplikacji tylko hostom z wymienionej listy. Więcej informacji na temat ustawienia ALLOWED_HOSTS można znaleźć na stronie *https://docs.djangoproject.com/en/3.0/ref/setting/#allowed-hosts*.

Zmodyfikuj plik *settings.py* swojego projektu w taki sposób, aby ustawienie ALLOWED_HOSTS miało następującą postać.

```
ALLOWED_HOSTS = ['moja-witryna.pl', 'localhost', '127.0.0.1']
```

Oprócz hosta *moja-witryna.pl* jawnie dołączyliśmy hosty *localhost* i *127.0.0.1*. Zrobiliśmy to, aby można było uzyskać dostęp do witryny za pośrednictwem hosta *localhost*, który jest domyślnym zachowaniem Django, gdy opcja DEBUG ma wartość True, a lista ALLOWED_HOSTS jest pusta. Teraz powinieneś otworzyć w przeglądarce stronę *http://moja-witryna.pl:8000/account/login/*.

Uruchomienie serwera programistycznego za pośrednictwem HTTPS

Niektóre metody uwierzytelniania za pośrednictwem serwisów społecznościowych, z których skorzystamy, wymagają połączenia HTTPS. Protokół TLS (ang. *Transport Layer Security*) jest standardem dla serwowania stron internetowych za pośrednictwem bezpiecznego połączenia. Poprzednikiem TLS jest SSL (ang. *Secure Sockets Layer*).

Chociaż protokół SSL jest obecnie uważany za przestarzały, w wielu bibliotekach i dokumentacji online można znaleźć odwołania zarówno do TLS, jak i SSL. Serwer programistyczny Django nie potrafi serwować witryn przy użyciu HTTPS, ponieważ to nie jest docelowe miejsce, w którym ma działać witryna. Aby przetestować funkcjonalność uwierzytelniania przez serwis społecznościowy, który serwuje naszą witrynę z pomocą HTTPS, skorzystamy z rozszerzenia RunServerPlus pakietu Django Extensions. Django Extensions to zbiór niestandardowych rozszerzeń Django tworzonych przez firmy zewnętrzne. Należy pamiętać, że tej metody nie należy stosować do serwowania witryn w środowisku produkcyjnym. Służy ona wyłącznie do stosowania na serwerach programistycznych.

Aby zainstalować Django Extensions, skorzystaj z następującego polecenia.

```
pip install django-extensions==2.2.5
```

Teraz należy zainstalować Werkzeug — pakiet zawierający warstwę debuggera wymaganą przez rozszerzenie RunServerPlus. Aby go zainstalować, skorzystaj z następującej komendy.

```
pip install werkzeug==0.16.0
```

Na koniec użyj poniższego polecenia, aby zainstalować pyOpenSSL — pakiet, który jest wymagany do korzystania z funkcjonalności SSL (TLS) rozszerzenia RunServerPlus.

```
pip install pyOpenSSL==19.0.0
```

Przeprowadź edycję pliku *settings.py* w swoim projekcie i dodaj do ustawienia `INSTALLED_APPS` rozszerzenie Django Extensions tak, jak pokazałem poniżej.

```
INSTALLED_APPS = [
    #...
    'django_extensions',
]
```

Aby uruchomić serwer programistyczny, skorzystaj z polecenia `runserver_plus` rozszerzeń Django Extensions.

```
python manage.py runserver_plus --cert-file cert.crt
```

W poleceniu `runserver_plus` należy podać nazwę pliku zawierającego certyfikat SSL (TLS). Rozszerzenie Django Extensions automatycznie wygeneruje klucz wraz z certyfikatem.

W przeglądarce przejdź pod adres *https://moja-witryna.pl:8000/account/login/*. Teraz korzystasz z witryny za pośrednictwem HTTPS. Ponieważ używasz certyfikatu z podpisem własnym, Twoja przeglądarka może wyświetlić ostrzeżenie o zabezpieczeniach. Jeśli tak się stanie, kliknij łącze *Zaawansowane* i zaakceptuj certyfikat z podpisem własnym, aby przeglądarka ufała certyfikatowi.

Zauważysz, że adres URL zaczyna się od *https://*, a ikona zabezpieczeń ma postać kłódki, co wskazuje, że połączenie jest bezpieczne (rysunek 4.21).

Rysunek 4.21. Adres URL z ikoną bezpiecznego połączenia

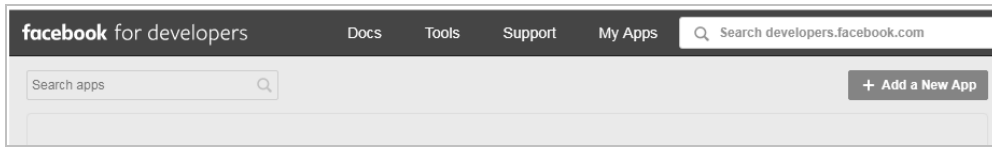
Możesz teraz podczas prac programistycznych serwować swoją witrynę przez HTTPS. W ten sposób możesz przetestować uwierzytelnianie za pośrednictwem serwisów społecznościowych Facebook, Twitter i Google.

Uwierzytelnienie za pomocą serwisu Facebook

Jeżeli chcesz pozwolić użytkownikowi na zalogowanie się do naszej witryny internetowej za pomocą jego konta w serwisie Facebook, poniższy wiersz kodu umieść na liście `AUTHENTICA ↪TION_BACKENDS` w pliku *settings.py* projektu.

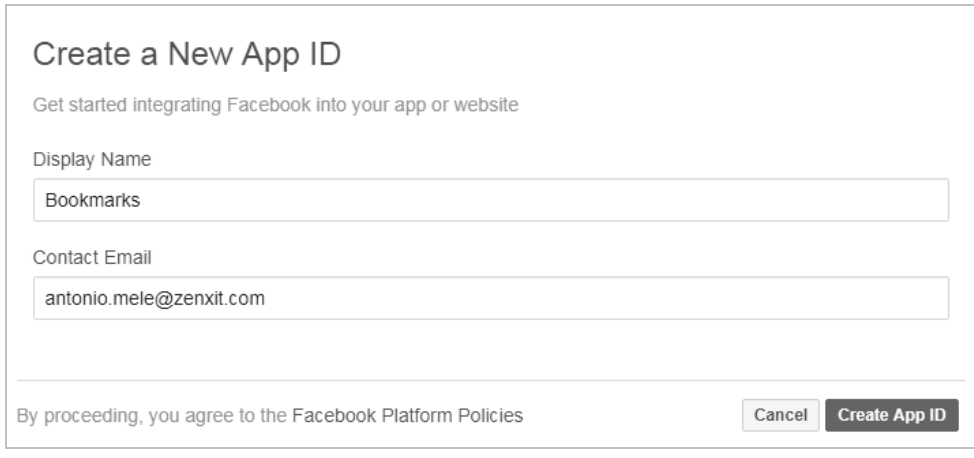
```
'social.backends.facebook.Facebook2OAuth2',
```

Dodanie uwierzytelnienia społecznościowego za pomocą serwisu Facebook wymaga konta programistycznego Facebook oraz utworzenia nowej aplikacji Facebook. W przeglądarce internetowej przejdź pod adres *https://developers.facebook.com/apps*. Po utworzeniu konta programistycznego Facebook zobaczysz w przeglądarce następujący nagłówek (rysunek 4.22).



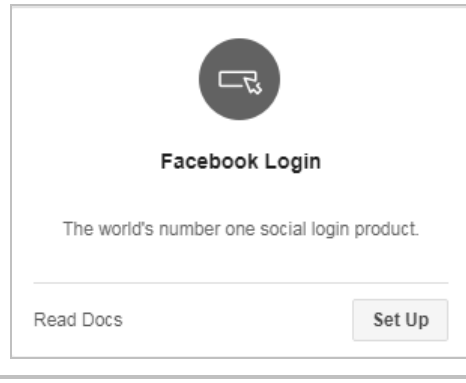
Rysunek 4.22. Menu portalu programisty serwisu Facebook

Teraz kliknij przycisk *Create App* z menu *My Apps*. Wyświetli się formularz tworzenia nowej aplikacji podobny do pokazanego poniżej (rysunek 4.23).



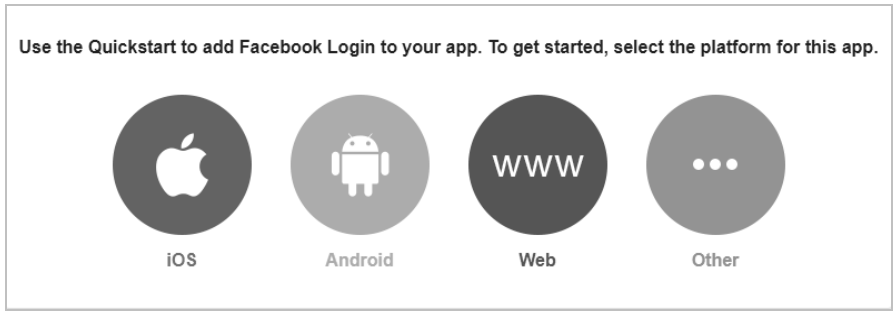
Rysunek 4.23. Formularz tworzenia nowego identyfikatora aplikacji Facebook

W polu *Display Name* podaj *Bookmarks*, dodaj kontaktowy adres e-mail i kliknij przycisk *Create App ID*. Po utworzeniu aplikacji przejdź do jej panelu (ang. *dashboard*). Wyświetlają się w nim różne funkcjonalności, które możesz skonfigurować dla swojej aplikacji. Poszukaj ramki oznaczonej *Facebook Login* (rysunek 4.24) i kliknij *Set Up*.



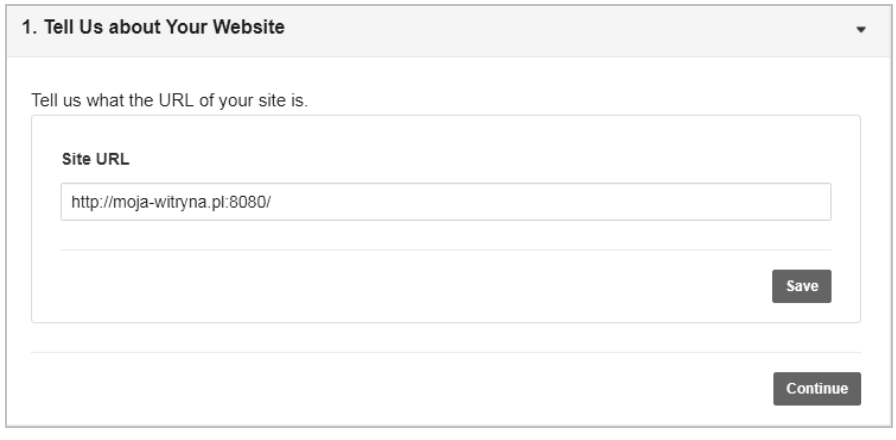
Rysunek 4.24. Blok logowania przy użyciu aplikacji Facebook

Wyświetli się okno wyboru platformy (rysunek 4.25).



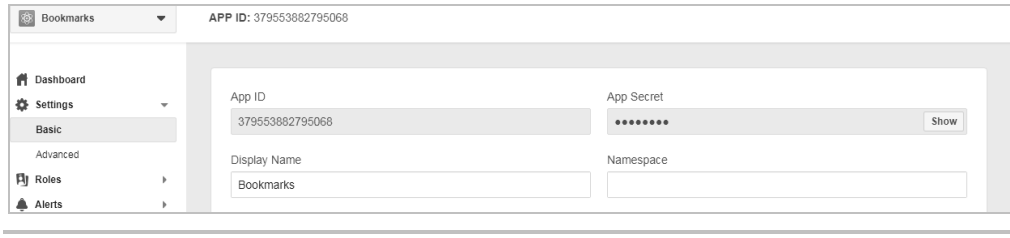
Rysunek 4.25. Okno wyboru platformy dla logowania przez Facebook

Wybierz platformę *Web*. Powinieneś zobaczyć następujący formularz (rysunek 4.26).



Rysunek 4.26. Konfiguracja platformy Web dla logowania przez Facebook

W polu *Site URL* wpisz *http://moja-witryna.pl:8000/* i kliknij przycisk *Save*. Pozostałą część procesu tworzenia aplikacji możesz pominąć. W menu po lewej stronie kliknij *Dashboard*. Poszukaj ustawień aplikacji (*Settings/Basic*). Powinieneś zobaczyć ekran podobny do pokazanego poniżej (rysunek 4.27).



Rysunek 4.27. Szczegóły aplikacji Facebook

Skopiuj wartości *App ID* i *App Secret*, a następnie dodaj je do pliku *settings.py* projektu, tak jak pokazałem poniżej.

```
SOCIAL_AUTH_FACEBOOK_KEY = 'XXX' # Wartość App ID pobrana z serwisu Facebook.
SOCIAL_AUTH_FACEBOOK_SECRET = 'XXX' # Wartość App Secret pobrana z serwisu Facebook.
```

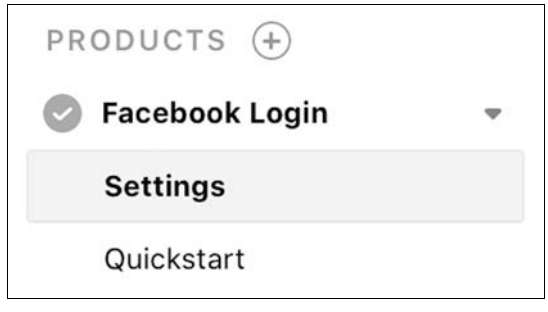
Opcjonalnie możesz dodać opcję `SOCIAL_AUTH_FACEBOOK_SCOPE` i zdefiniować dodatkowe uprawnienia dla użytkowników serwisu Facebook.

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

Teraz wróć do Facebooka i kliknij *Settings*. Zobaczysz formularz z wieloma ustawieniami dla Twojej aplikacji. W polu *App Domains* dodaj domenę *moja-witryna.pl* tak, jak pokazałem poniżej (rysunek 4.28).

Rysunek 4.28. Dozwolone domeny dla aplikacji Facebook

Kliknij *Save Changes*. Następnie w menu po lewej stronie, pod łączem *Products*, kliknij *Facebook Login*, a następnie *Settings*, tak jak pokazałem poniżej (rysunek 4.29).

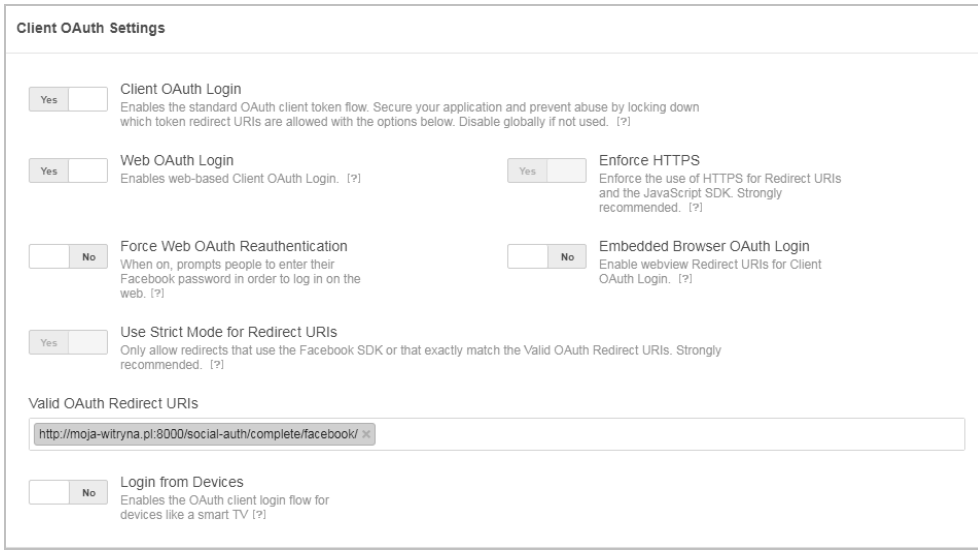


Rysunek 4.29. Menu logowania przez Facebook

Upewnij się, że są aktywne następujące opcje:

- Client OAuth Login
- Web OAuth Login
- Enforce HTTPS
- Embedded Browser OAuth Login

W polu *Valid OAuth redirect URIs* wprowadź adres *http://moja-witryna.pl:8000/social-auth/complete/facebook/*. Ekran ustawień powinien wyglądać następująco (rysunek 4.30).



Rysunek 4.30. Ustawienia klienta OAuth dla logowania przez Facebook

Teraz otwórz szablon *registration/login.html* aplikacji `account` i w bloku `content` dodaj poniższy fragment kodu.

```
<div class="social">
  <ul>
    <li class="facebook"><a href="{% url "social:begin" "facebook"
%}">Zaloguj się za pomocą konta Facebook</a></li>
  </ul>
</div>
```

W przeglądarce internetowej przejdź pod adres *https://moja-witryna.pl:8000/account/login/*. Teraz zmodyfikowana strona logowania powinna wyglądać tak, jak pokazałem poniżej (rysunek 4.31).

Rysunek 4.31. Strona logowania z przyciskiem do uwierzytelniania przez Facebook

Kliknij przycisk *Zaloguj się za pomocą konta Facebook*. Zostaniesz przekierowany do serwisu Facebook i zobaczysz okno modalne z pytaniem o udzielenie uprawnień aplikacji `Bookmarks` na uzyskanie dostępu do Twojego publicznego profilu w serwisie Facebook (rysunek 4.32).

Rysunek 4.32. Okno modalne do udzielenia uprawnień aplikacji

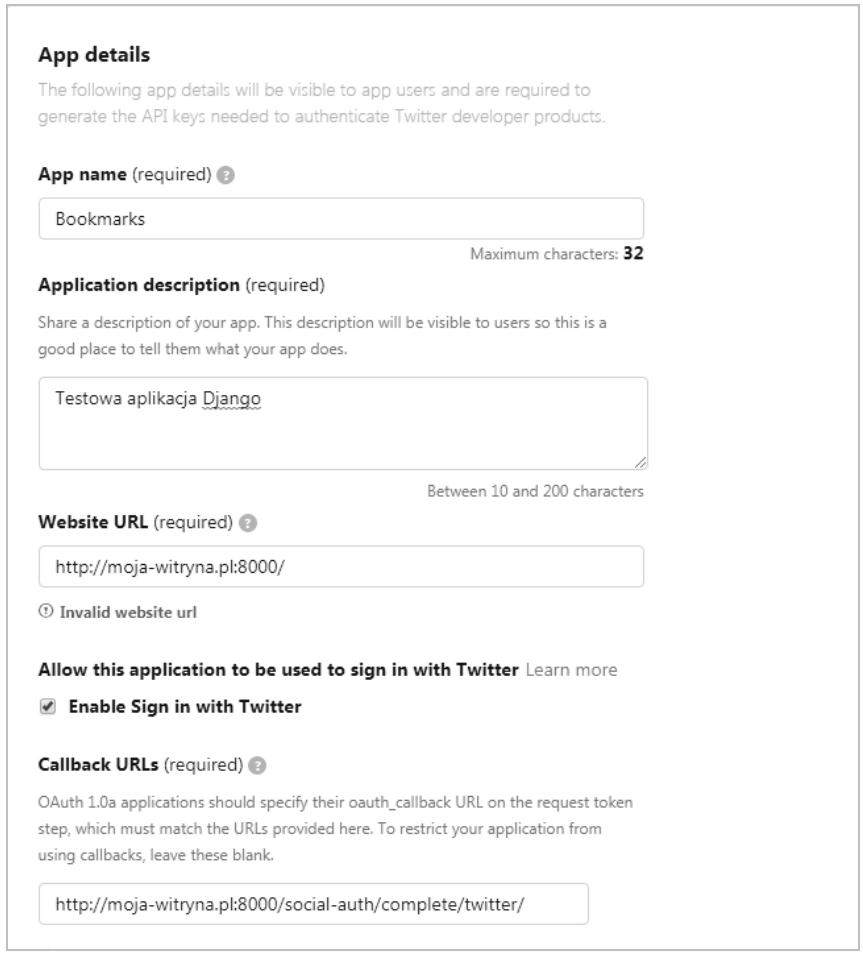
Kliknij przycisk *Kontynuuj jako*. Jeżeli wszystko przebiegnie bez problemów, zostaniesz zalogowany i przeniesiony na stronę panelu głównego w budowanej witrynie internetowej. Pamiętaj, że ten adres URL został podany w opcji `LOGIN_REDIRECT_URL`. Jak możesz zobaczyć, implementacja w aplikacji uwierzytelnienia za pomocą innego serwisu społecznościowego jest naprawdę prosta.

Uwierzytelnienie za pomocą serwisu Twitter

W przypadku uwierzytelnienia z wykorzystaniem serwisu Twitter konieczne jest dodanie poniższego wiersza kodu do listy `AUTHENTICATION_BACKENDS` w pliku *settings.py* aplikacji.

```
'social_core.backends.twitter.TwitterOAuth',
```

Musisz utworzyć nową aplikację z poziomu konta w serwisie Twitter. W przeglądarce internetowej przejdź pod adres *https://developer.twitter.com/en/apps/create*. Jeśli dotąd nie miałeś konta programistycznego, wyświetli się kilka pytań wymaganych do utworzenia takiego konta. Gdy już masz konto programisty, podczas tworzenia nowej aplikacji wyświetli się poniższy formularz (rysunek 4.33).



Rysunek 4.33. Konfigurowanie aplikacji Twitter

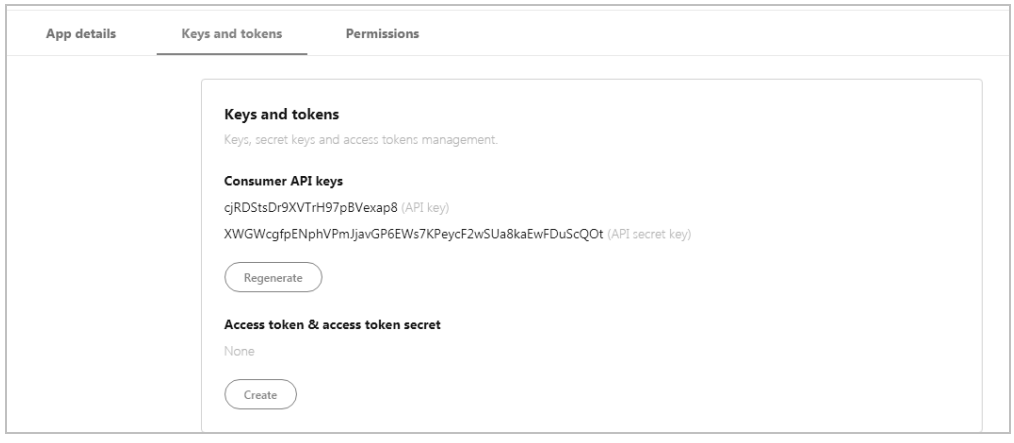
Podaj informacje szczegółowe dotyczące aplikacji. Oto m.in. one.

- *Website*. `http://moja-witryna.pl:8000/`
- *Callback URL*. `http://moja-witryna.pl:8000/social-auth/complete/twitter/`

Zaznacz opcję *Enable Sign in with Twitter*. Następnie kliknij *Create*. Zobaczysz szczegóły aplikacji. Kliknij kartę *Keys and tokens*. Powinieneś zobaczyć stronę podobną do pokazanej na rysunku 4.34.

Dane wymienione w polach *API Key* i *API secret key* skopiuj do przedstawionych poniżej ustawień w pliku *settings.py* projektu.

```
SOCIAL_AUTH_TWITTER_KEY = 'XXX' # Wartość Consumer Key pobrana z serwisu Twitter.
SOCIAL_AUTH_TWITTER_SECRET = 'XXX' # Wartość Consumer Secret pobrana z serwisu Twitter.
```

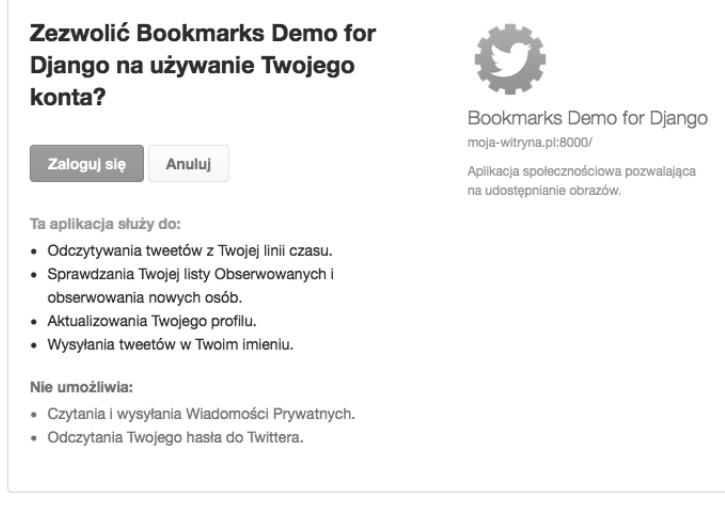


Rysunek 4.34. Klucze API aplikacji Twitter

Teraz przeprowadź edycję szablonu *registration/login.html* i poniższy fragment kodu umieść w elemencie `<ul>`.

```
<li class="twitter"><a href="{% url "social:begin" "twitter"
%}">Zaloguj się za pomocą konta Twitter</a></li>
```

W przeglądarce internetowej przejdź pod adres *http://moja-witryna.pl:8000/account/login/* i kliknij łącze *Zaloguj za pomocą konta Twitter*. Zostaniesz przekierowany do serwisu Twitter i poproszony o autoryzowanie aplikacji, co pokazałem na rysunku 4.35.



Rysunek 4.35. Okno modalne pozwalające na udzielenie uprawnień aplikacji

Kliknij przycisk *Zaloguj się*. Zostaniesz zalogowany, a następnie przeniesiony na stronę panelu głównego budowanej witryny internetowej.

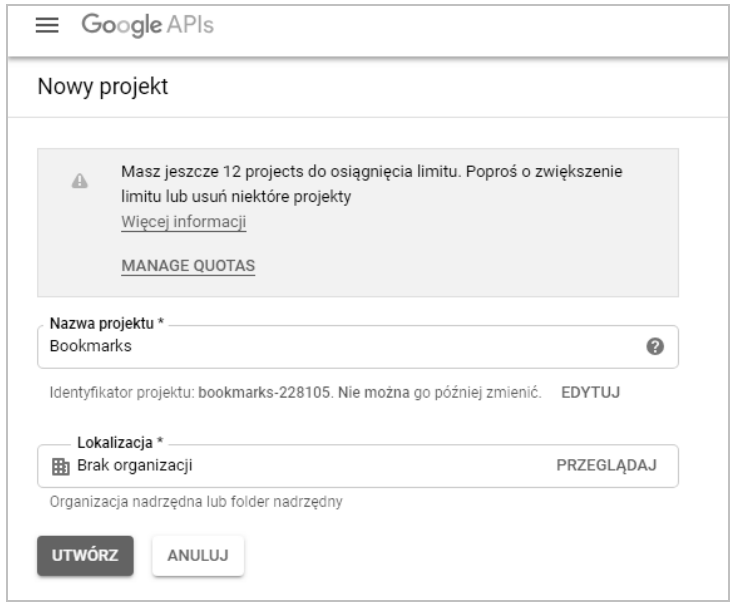
Uwierzytelnienie za pomocą serwisu Google

Serwis Google oferuje możliwość uwierzytelnienia za pomocą OAuth2. Więcej informacji na temat tej implementacji znajdziesz na stronie *https://developers.google.com/identity/protocols/OAuth2*.

Aby zaimplementować uwierzytelnienie za pomocą Google, dodaj do ustawienia `AUTHENTICATION_BACKENDS` w pliku *settings.py* Twojego projektu następujący wiersz.

```
'social_core.backends.google.GoogleOAuth2',
```

Pracę trzeba zacząć od utworzenia klucza API w Konsoli interfejsów API Google. W przeglądarce internetowej przejdź pod adres *https://console.developers.google.com/apis/credentials*. Następnie na pasku menu w górnym lewym rogu strony wskaż opcję wyboru projektu, po czym kliknij *NOWY PROJEKT*. Wpisz nazwę projektu i kliknij przycisk *Utwórz* (rysunek 4.36).

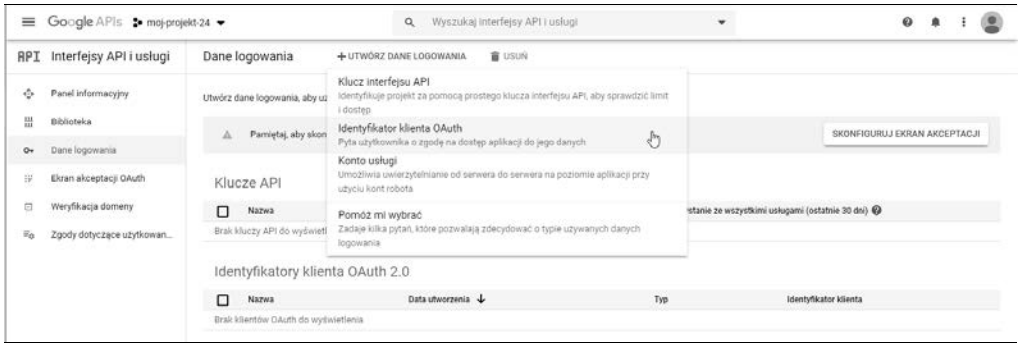


Rysunek 4.36. Formularz tworzenia projektu Google

Po utworzeniu projektu w obszarze *Dane logowania* kliknij *UTWÓRZ DANE LOGOWANIA* i wybierz *Identyfikator klienta OAuth* w następujący sposób (rysunek 4.37).

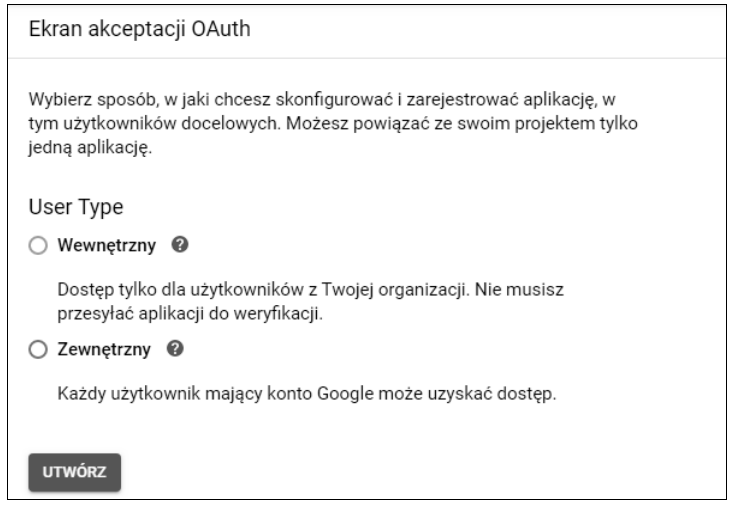
Google poprosi Cię najpierw o skonfigurowanie ekranu akceptacji (rysunek 4.38).

Na pokazanej stronie wyświetlana jest ramka, w której użytkownicy wyrażają zgodę na dostęp do witryny za pomocą konta Google. Kliknij przycisk ekranu *Skonfiguruj ekran akceptacji*. Zostaniesz przekierowany na stronę pokazaną na rysunku 4.39.



Rysunek 4.37. Tworzenie danych logowania dla Google API

Rysunek 4.38. Komunikat informujący o konieczności skonfigurowania ekranu akceptacji OAuth



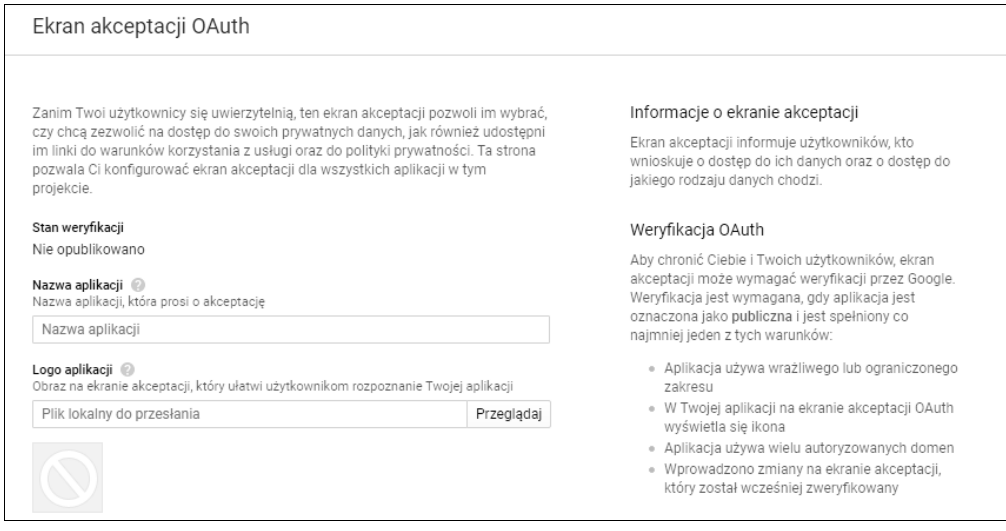
Rysunek 4.39. Wybór typu użytkownika na ekranie konfigurowania ekranu akceptacji OAuth

W obszarze *User Type* wybierz *Zewnętrzny*, po czym kliknij przycisk *UTWÓRZ*. Wyświetli się ekran pokazany na rysunku 4.40.

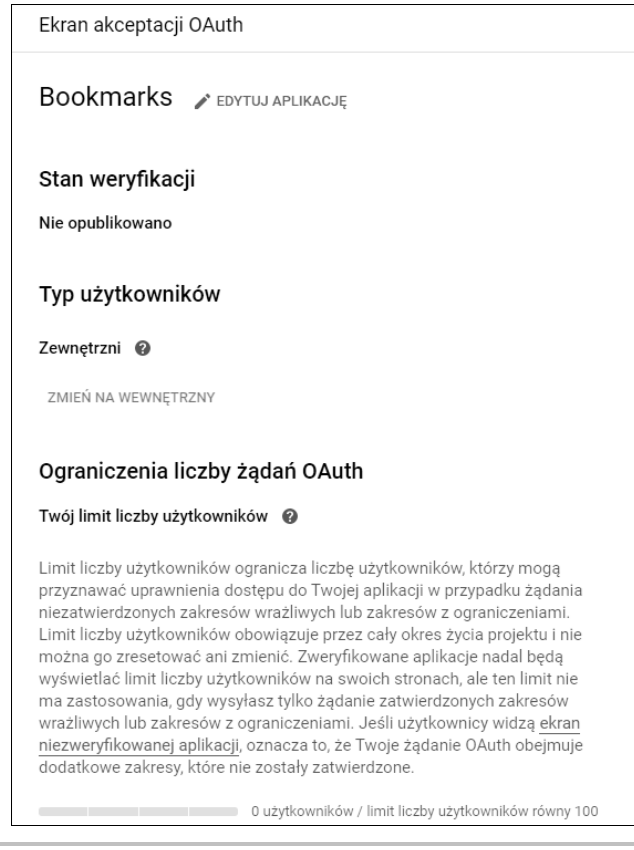
W wyświetlonym formularzu podaj następujące informacje.

- **Nazwa aplikacji**. Podaj nazwę *Bookmarks*.
- **Autoryzowane domeny**. Podaj *moja-witryna.pl*.

Kliknij *Zapisz*. Ekran akceptacji dla Twojej aplikacji został skonfigurowany. Wyświetlą się szczegóły ekranu akceptacji aplikacji (rysunek 4.41).



Rysunek 4.40. Konfigurowanie ekranu akceptacji OAuth



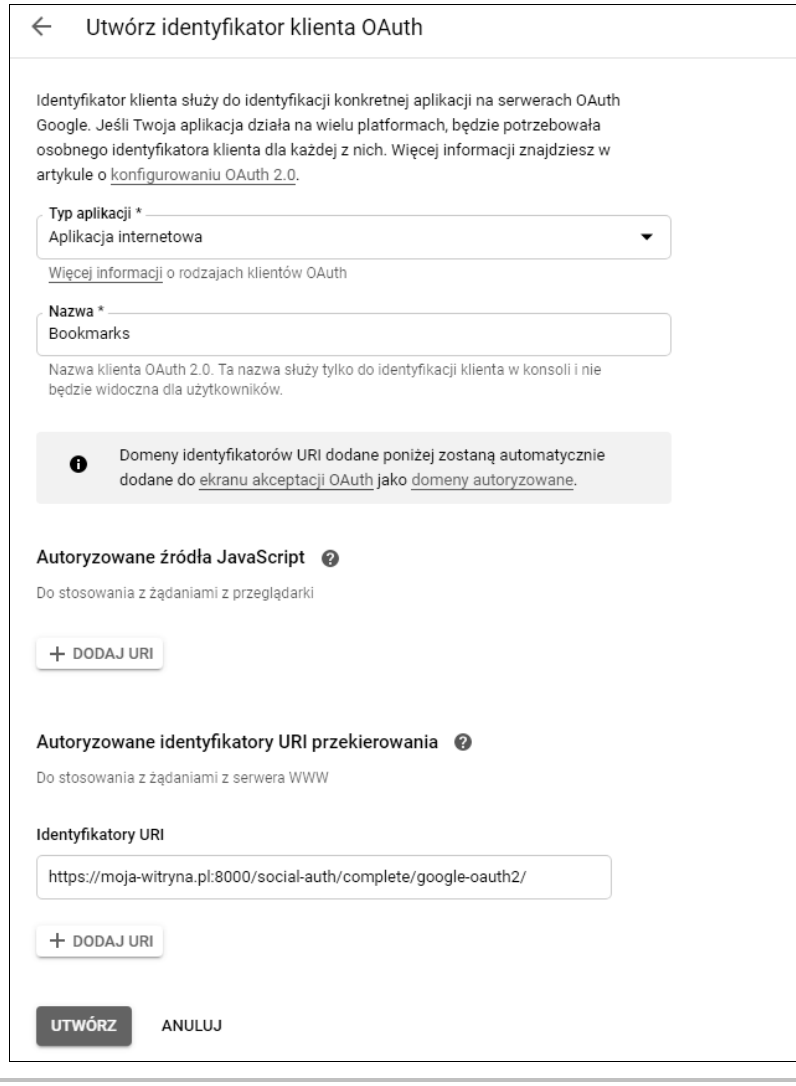
Rysunek 4.41. Szczegóły ekranu akceptacji Google OAuth

W menu w ramce po lewej stronie kliknij *Dane logowania*. Następnie ponownie kliknij łącze *UTWÓRZ DANE LOGOWANIA*, a następnie pozycję *Identyfikator klienta OAuth*.

W kolejnym kroku wprowadź następujące informacje:

- **Typ aplikacji**: wybierz *Aplikacja internetowa*;
- **Nazwa**: wpisz *Bookmarks*;
- **Autoryzowane identyfikatory URI przekierowania**: dodaj *https://moja-witryna.pl:8000/social-auth/complete/google-oauth2/*.

Wypełniony formularz powinien wyglądać tak, jak pokazałem na rysunku 4.42.

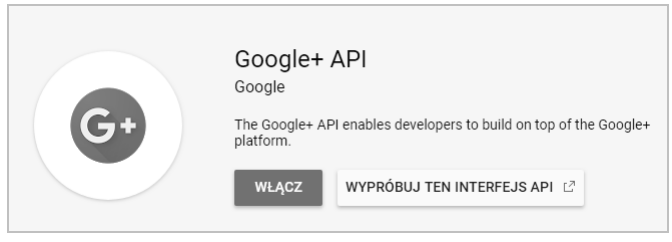


Rysunek 4.42. Formularz tworzenia identyfikatora klienta OAuth

Kliknij przycisk *Utwórz*. Otrzymasz wartości identyfikatora klienta i jego tajnego klucza, które musisz umieścić w pliku *settings.py* w następujący sposób.

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = '' # Wartość Consumer Key pobrana z serwisu Google.
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = '' # Wartość Consumer Secret pobrana z serwisu Google.
```

W menu po lewej stronie konsoli programisty Google, w obszarze *Interfejsy API i usługi*, kliknij łącze *Biblioteka*. Zobaczysz wyświetloną listę wszystkich API udostępnianych przez Google. W sekcji *Sieci społecznościowe* wybierz *Google+ API*, a następnie kliknij przycisk *Włącz* na wyświetlonej stronie, co pokazałem na rysunku 4.43.

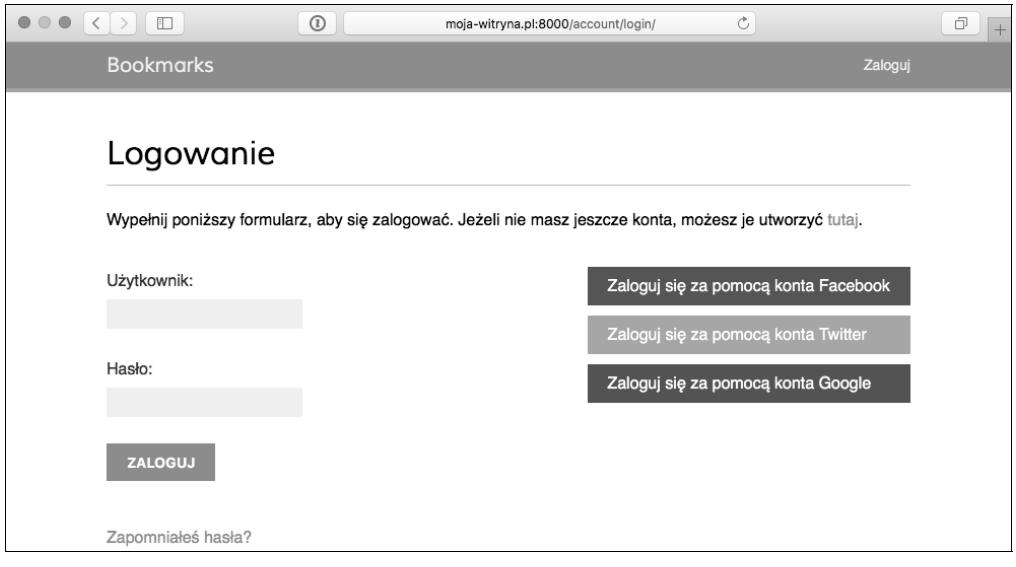


Rysunek 4.43. Blok Google+ API

Przeprowadź edycję szablonu *login.html* i dodaj poniższy fragment kodu do elementu `<ul>`.

```
<li class="google"><a href="{% url "social:begin" "google-oauth2" %}">Zaloguj się
↪za pomocą konta Google</a></li>
```

W przeglądarce internetowej przejdź pod adres *http://moja-witryna.pl:8000/account/login/*. Teraz zmodyfikowana strona logowania powinna wyglądać tak, jak pokazałem poniżej (rysunek 4.44).



Rysunek 4.44. Strona logowania zawierająca przyciski do uwierzytelniania za pośrednictwem serwisów Facebook, Twitter i Google

Kliknij przycisk *Zaloguj się za pomocą konta Google*. Zostaniesz zalogowany i przekierowany do panelu głównego w budowanej witrynie internetowej.

W ten sposób do budowanego tutaj projektu dodaliśmy możliwość uwierzytelnienia za pomocą innych serwisów społecznościowych. Moduł `Python-social-auth` zawiera wbudowane mechanizmy obsługi innych popularnych serwisów społecznościowych.

Podsumowanie

W tym rozdziale dowiedziałeś się, jak zaimplementować system uwierzytelniania w budowanej witrynie internetowej. Zaimplementowałeś wszystkie widoki potrzebne do rejestracji użytkownika, logowania i wylogowania oraz edycji i zerowania hasła. Utworzyłeś modele dla niestandardowych profili użytkownika oraz opracowałeś niestandardowy backend uwierzytelniania pozwalający na logowanie się użytkowników do Twojej witryny za pomocą adresu e-mail. Ponadto dodałeś możliwość zalogowania się do witryny za pomocą kont na serwisach Facebook, Twitter i Google.

W następnym rozdziale zobaczysz, jak zbudować system katalogowania obrazów, generowania miniatur obrazów oraz tworzenia widoków w technologii AJAX.

Django — wypróbowany framework dla profesjonalnych projektantów!

Django jest potężnym frameworkiem służącym do tworzenia aplikacji internetowych w Pythonie. Pozwala na pełne wykorzystywanie zalet tego języka, takich jak przejrzystość, elastyczność, wszechstronność i łatwość uczenia się. Użycie Django do budowania aplikacji w Pythonie jest atrakcyjną możliwością zarówno dla początkujących, jak i zaawansowanych programistów. Aby jednak zapewnić tworzonym projektom odpowiednią, profesjonalną jakość, trzeba się nauczyć rozwiązywać problemy powstające podczas tworzenia aplikacji internetowych, stosować najlepsze praktyki programistyczne, a także skutecznie wdrażać i testować aplikację.

To trzecie wydanie praktycznego przewodnika po budowie aplikacji internetowych. Krok po kroku opisano w nim wszystkie istotne elementy procesu projektowania i wdrażania aplikacji: bloga, witryny społecznościowej, sklepu internetowego oraz platformy e-learningowej. Zawarte tu szczegółowe wskazówki pomogą integrować popularne technologie, usprawniać aplikacje z wykorzystaniem technik AJAX, tworzyć API REST oraz konfigurować środowisko produkcyjne dla projektów Django. Dzięki tej książce niepostrzeżenie opanujesz najistotniejsze zasady pracy w Django — i czym prędzej zaczniesz od podstaw budować praktyczne projekty. Nowością w tym wydaniu jest rozdział poświęcony projektowaniu serwera czatu z wykorzystaniem Django Channels.

W książce:

- praktyczna strona projektowania aplikacji internetowych
- podstawy Django, w tym ORM, szablony, adresy URL, formularze i uwierzytelnianie
- funkcje zaawansowane: niestandardowe pola modelu i oprogramowanie pośredniczące
- wykorzystanie technik AJAX, system płatności, CMS, API RESTful
- integracja projektu z takimi technologiami jak Redis, RabbitMQ, PostgreSQL i Channels
- wdrażanie projektów Django za pomocą NGINX, uWSGI i Daphne

Antonio Melé

od blisko 15 lat rozwija projekty Django dla klientów z różnych branż. W 2009 roku założył Zenx IT — firmę programistyczną specjalizującą się w budowaniu produktów cyfrowych. Pracował jako CTO i konsultant techniczny wielu start-upów. Zarządzał także zespołami programistycznymi pracującymi dla dużych firm informatycznych. Komputery i programowanie są jego pasją od dzieciństwa.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7250-4

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł

Packt