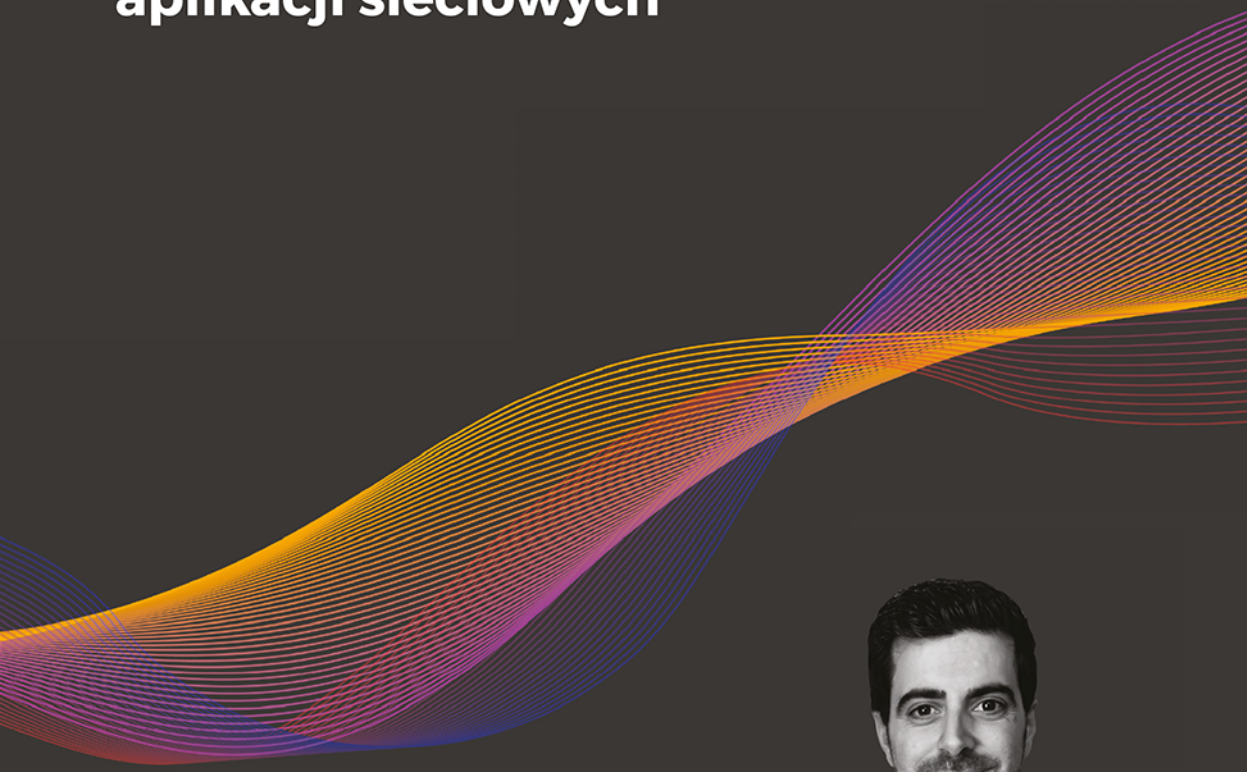
Wydanie IV

Django 4

Praktyczne tworzenie aplikacji sieciowych



Antonio Melé

Helion

Packt

Tytuł oryginału: Django 4 By Example: Build powerful and reliable Python web applications from scratch, 4th Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-8322-370-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/djan44
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 4

Utworzenie witryny społecznościowej

W poprzednim rozdziale nauczyłeś się, jak zaimplementować system tagowania i polecać podobne posty. Zaimplementowałeś niestandardowe tagi szablonów i filtry. Nauczyłeś się również, jak tworzyć mapy witryn i kanały wiadomości, a także zbudowałeś — za pomocą PostgreSQL — pełnotekstową wyszukiwarkę.

W tym rozdziale dowiesz się, jak opracować funkcjonalności związane z obsługą kont — w tym rejestrację użytkownika, zarządzanie hasłami, edycję profilu i uwierzytelnianie — w celu stworzenia serwisu społecznościowego. W następnych kilku rozdziałach zaimplementujemy w tej witrynie funkcje społecznościowe, pozwalające użytkownikom współdzielenie zdjęć i interakcje między sobą. Użytkownicy będą mogli oznaczyć w internecie dowolne zdjęcie 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).

Oto zagadnienia, na których skoncentruję się w tym rozdziale.

- Utworzenie widoku logowania.
- Użycie frameworka uwierzytelniania w Django.
- Tworzenie w Django szablonów widoków logowania, wylogowania, zmiany hasła i resetowania hasła.
- Rozbudowa modelu `User` o obsługę niestandardowego profilu.
- Utworzenie widoków pozwalających na rejestrację użytkowników.
- Konfigurowanie projektu do przesyłania plików multimedialnych.
- Użycie frameworka `messages`.
- Implementacja własnego mechanizmu uwierzytelniania.
- Uniemożliwianie użytkownikom korzystania z istniejącego adresu e-mail.

Pracę rozpoczynamy od utworzenia nowego projektu.

Kod źródłowy przykładów z tego rozdziału można znaleźć pod adresem *https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04*.

Wszystkie pakiety Pythona użyte w tym rozdziale są zawarte w pliku *requirements.txt* dołączonym do plików z kodem źródłowym przykładów z tego rozdziału. Aby zainstalować każdy pakiet Pythona, możesz postępować zgodnie z instrukcjami zamieszczonymi w poniższych podrozdziałach. Możesz także zainstalować wszystkie wymagania jednocześnie za pomocą polecenia `pip install -r requirements.txt`.

Utworzenie projektu witryny społecznościowej

Przystępujemy teraz do budowy aplikacji społecznościowej umożliwiającej użytkownikom udostępnianie zdjęć 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 zdjęć 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
python -m venv env/bookmarks
```

Jeśli używasz systemu Linux lub macOS, aby aktywować środowisko wirtualne, uruchom następujące polecenie.

```
source env/bookmarks/bin/activate
```

Jeśli używasz systemu Windows, użyj następującego polecenia.

```
.\env\bookmarks\Scripts\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~=4.1.0
```

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*.

Przeprowadź edycję pliku *settings.py* i dodaj do listy `INSTALLED_APPS` przed jakąkolwiek inną zainstalowaną aplikacją następujący wiersz (wyróżniony na poniższym listingu pogrubioną czcionką).

```
INSTALLED_APPS = [
    'account.apps.AccountConfig',
    'django.contrib.admin',
    'django.contrib.admin',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Django szuka szablonów według kolejności występowania aplikacji w ustawieniu `INSTALLED_APPS`. Aplikacja `django.contrib.admin` zawiera standardowe szablony uwierzytelniania, które zastąpimy w aplikacji `account`. Umieszczając aplikację na pierwszym miejscu w ustawieniu `INSTALLED_APPS`, zapewniamy domyślne wykorzystywanie naszych szablonów uwierzytelniania, a nie szablonów uwierzytelniania zawartych w innych aplikacjach.

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 aplikacji `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 middleware (nazywane także oprogramowaniem pośredniczącym) 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 17., „Wdrażanie".

Framework uwierzytelniania obejmuje również poniższe modele, które są zdefiniowane w modelach aplikacji `django.contrib.auth.models`.

- `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. Stworzymy widok, który w celu zalogowania użytkownika wykona następujące czynności.

- Wyświetlenie użytkownikowi formularza logowania.
- 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 `Password` `↪Input` do wygenerowania elementu HTML `password`. Zawiera on atrybut `type="pass` `↪word"`, 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(request,
```

```
                        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})
```

Oto jakie działania realizuje podstawowy widok logowania.

Po wywołaniu widoku `user_login` przez żądanie `GET` za pomocą wywołania `form = Login ↪Form()` tworzymy nowy egzemplarz formularza logowania. Następnie formularz jest przekazywany do szablonu.

Kiedy użytkownik wyśle formularz przy użyciu żądania `POST`, przeprowadzane są następujące działania:

- 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”.

Ostrzeżenie

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 = [
    path('login/', views.user_login, name='login'),
]
```

Przeprowadź edycję głównego pliku *urls.py* znajdującego się katalogu projektu `book ↪marks` i dodaj wzorzec adresu URL aplikacji `account`, co przedstawiłem poniżej. Nowy kod został wyróżniony pogrubioną czcionką.

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

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.

W katalogu aplikacji `account` utwórz wymienioną poniżej strukturę plików i katalogów.

```
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 jest dostępna pod adresem *https://github.com/PacktPublishing/Django-4-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 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.

W wierszu polecenia powłoki uruchom następujące polecenie.

```
python manage.py createsuperuser
```

Wyświetli się wynik pokazany poniżej. Wprowadź żądaną nazwę użytkownika, adres e-mail i hasło w następujący sposób.

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
```

Następnie wyświetli się poniższy komunikat o powodzeniu operacji.

```
Superuser created successfully.
```

Uruchom serwer programistyczny za pomocą następującego polecenia.

```
python manage.py runserver
```

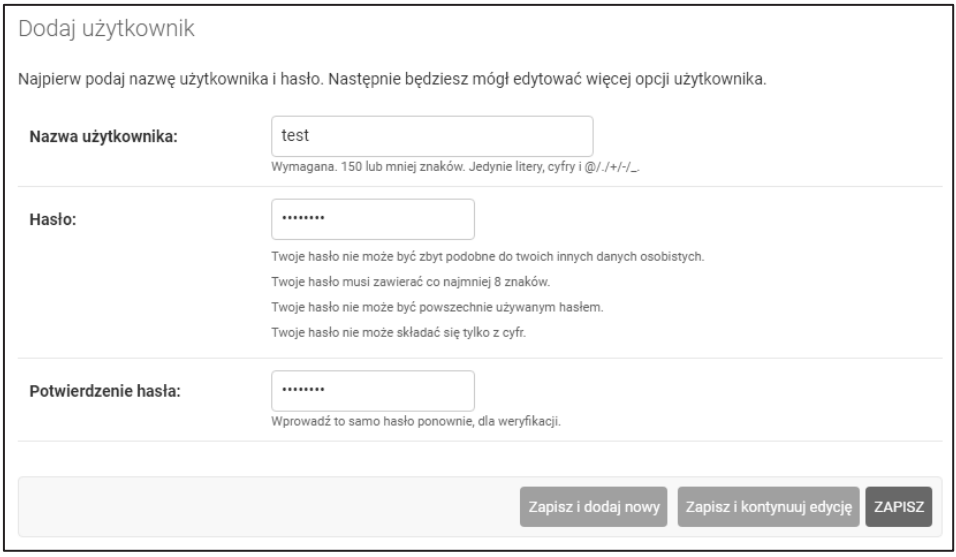
W przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/admin/*. Dostęp do witryny administracyjnej uzyskasz po podaniu ustalonej przed chwilą nazwy użytkownika i hasła. Gdy znajdziesz się już w witrynie administracyjnej Django, zobaczysz modele `Users` (łącze *Użytkownicy*) i `Group` (łącze *Grupy*) dla wbudowanego w Django frameworka uwierzytelniania.

Strona wygląda następująco (rysunek 4.1).

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

W wierszu *Użytkownicy* kliknij łącze *Dodaj*.

Utwórz nowego użytkownika, używając do tego witryny administracyjnej (rysunek 4.2).



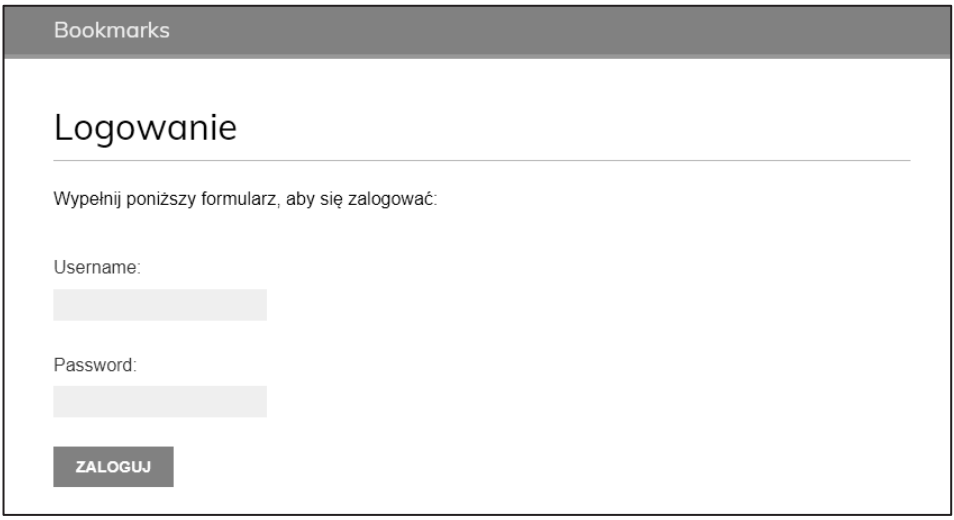
Rysunek 4.2. Strona logowania użytkownika

Wprowadź dane użytkownika i aby zapisać nowego użytkownika w bazie danych, kliknij przycisk *ZAPISZ*.

Następnie w obszarze *Informacje osobiste* wypełnij pola *Imię*, *Nazwisko* i *Adres e-mail* w sposób pokazany na rysunku 4.3, po czym kliknij przycisk *Zapisz*, aby zapisać zmiany.

Rysunek 4.3. Formularz edycji danych użytkownika w witrynie administracyjnej Django

W przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/*. Powinieneś zobaczyć wygenerowany szablon wraz z formularzem logowania (rysunek 4.4).

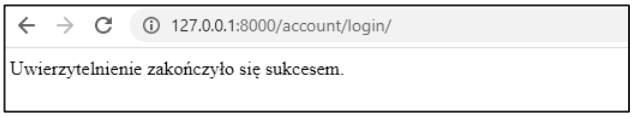


Rysunek 4.4. Strona logowania użytkownika

Wprowadź niepoprawne dane uwierzytelniające i wyślij formularz. Powinieneś otrzymać komunikat `Nieprawidłowe dane uwierzytelniające` (rysunek 4.5).

Rysunek 4.5. Komunikat tekstowy informujący o niepoprawnym logowaniu

Wprowadź poprawne poświadczenia. Wyświetli się komunikat `Uwierzytelnienie zakończyło się sukcesem` (rysunek 4.6).



Rysunek 4.6. Komunikat tekstowy z informacją o pomyślnym uwierzytelnianiu

Nauczyłeś się uwierzytelniać użytkowników i tworzyć własny widok uwierzytelniania. Co prawda możesz tworzyć własne widoki uwierzytelniania, ale Django zapewnia gotowe do użycia widoki uwierzytelniania, z których możesz skorzystać.

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.

Do obsługi operacji resetowania hasła Django oferuje zaś następujące widoki.

- `PasswordResetView`: Umożliwienie użytkownikowi ponownego ustawienia 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 ponowne ustawienie hasła.
- `PasswordResetConfirmView`: widok umożliwiający użytkownikowi zdefiniowanie nowego hasła.
- `PasswordResetCompleteView`: strona informująca o sukcesie operacji; zostanie wyświetlona użytkownikowi, gdy resetowanie 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 można znaleźć pod adresem *https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views*.

Widoki logowania i wylogowania

Przeprowadź edycję pliku *urls.py* aplikacji `account` i dodaj kod wyróżniony pogrubioną czcionką.

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

urlpatterns = [
    # Poprzedni adres URL strony logowania
    # path('login/', views.user_login, name='login'),
    #  login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

W powyższym kodzie oznaczyliśmy jako komentarz wzorzec adresu URL dla utworzonego wcześniej widoku `user_login`. Teraz wykorzystamy widok `LoginView` oferowany przez wbudowany w Django framework uwierzytelniania. Dodaliśmy także wzorzec adresu URL dla widoku `LogoutView`.

Utwórz nowy podkatalog w katalogu *templates/* 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. Podczas konfigurowania projektu 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 niepoprawne.
      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`. Ta zmienna będzie dostarczana do widoku logowania, jeśli przekażesz do żądania parametr o nazwie `next`, na przykład przez otwarcie w przeglądarce strony *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.

Dodaliśmy wzorce i szablony adresów URL dla widoków logowania i wylogowania. Użytkownicy mogą teraz logować się i wylogowywać 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.

Przeprowadź edycję pliku *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'})
```

Stworzyliśmy widok pulpitu nawigacyjnego i zastosowaliśmy do niego dekorator `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 HTML `<input>`.

Zdefiniowaliśmy również zmienną `section`. Użyjemy tej zmiennej do zaznaczenia bieżącej sekcji w menu głównym witryny.

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`. Nowy kod został wyróżniony pogrubioną czcionką:

```
urlpatterns = [
    # Poprzedni adres URL strony logowania
    # path('login/', views.user_login, name='login'),

    # login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

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

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

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

Zdefiniowaliśmy następujące ustawienia.

- `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.

Użyliśmy nazw adresów URL, które wcześniej zdefiniowaliśmy we wzorcach adresów URL za pomocą 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. 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`.

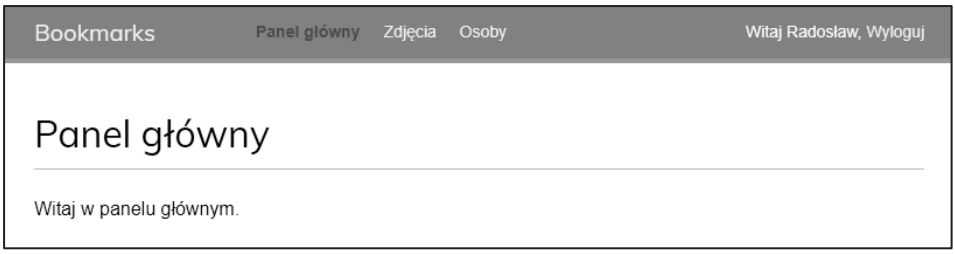
Przeprowadź edycję pliku *base.html* i dodaj poniższe wiersze wyróżnione pogrubioną czcionką.

```
{% 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>
    {% 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="#">Zdjęcia</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|default:request.user.username }},
        <a href="{% url "logout" %}">Wyloguj</a>
      {% else %}
        <a href="{% url "login" %}">Zaloguj </a>
      {% endif %}
    </span>
  </div>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
</body>
</html>
```

Menu witryny internetowej będzie wyświetlane jedynie uwierzytelnionym użytkownikom. Sprawdzana jest także zmienna reprezentująca sekcję witryny, w celu dodania klasy atrybutu `selected` do odpowiedniego elementu `<li>` i tym samym zaznaczenia

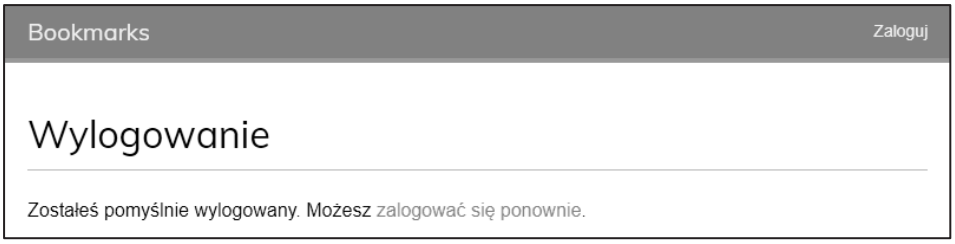
za pomocą CSS nazwy bieżącej sekcji. Jeśli użytkownik jest zalogowany, wyświetlane jest imię uwierzytelnionego użytkownika i łącze pozwalające mu na wylogowanie. Jeżeli użytkownik nie jest uwierzytelniony, wyświetlone będzie łącze pozwalające mu na zalogowanie. Jeśli nazwa użytkownika jest pusta, jest wyświetlana za pomocą wywołania `request.user.first_name|default:request.user.username`.

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*. Wyświetli się ekran pokazany na rysunku 4.7.



Rysunek 4.7. Strona panelu głównego

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 podobną do pokazanej na rysunku 4.8.



Rysunek 4.8. Strona wyświetlana po wylogowaniu użytkownika

Na tej stronie widać, że użytkownik jest wylogowany, w związku z czym menu witryny nie jest wyświetlane. Łącze znajdujące się po prawej stronie nagłówka zmienia się na *Zaloguj*.

Ostrzeżenie

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`. Obie aplikacje zawierają szablony wylogowania znajdujące się w tej samej ścieżce względnej. Z tego powodu mechanizm ładujący szablony Django przejrzy różne aplikacje na liście `INSTALLED_APPS` i użyje pierwszego znalezionego szablonu.

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.

```
urlpatterns = [
    # Poprzedni adres URL strony logowania
    # path('login/', views.user_login, name='login'),

    # login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # 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'),

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

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 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 4.9.

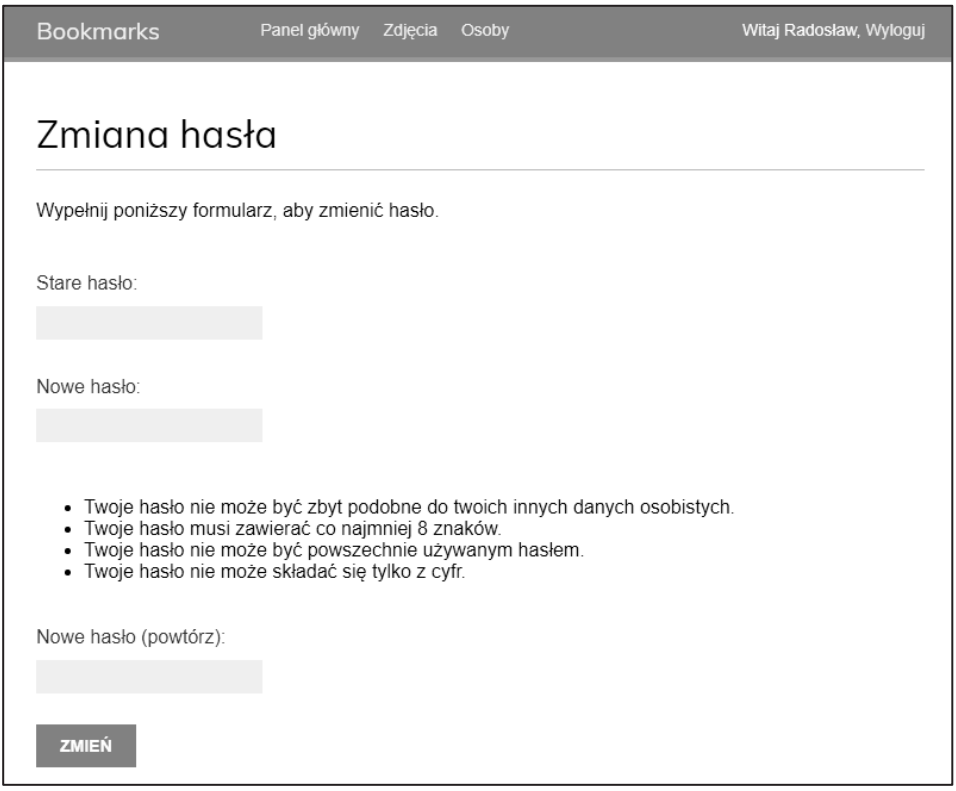
W wyświetlonym formularzu należy podać dotychczasowe hasło oraz nowe, a następnie kliknąć przycisk *Zmień*. Jeżeli operacja przebiegnie bez problemów, zostanie wyświetlona pokazana poniżej strona wraz z komunikatem informującym o sukcesie (rysunek 4.10).

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

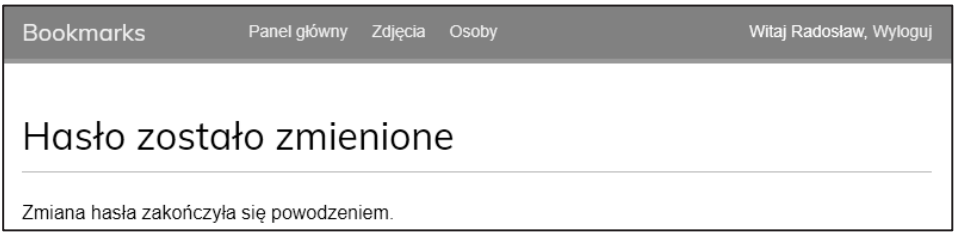
Widoki odzyskiwania hasła

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

```
urlpatterns = [
    # Poprzedni adres URL strony logowania
    # path('login/', views.user_login, name='login'),
```



Rysunek 4.9. Formularz zmiany hasła



Rysunek 4.10. Strona z komunikatem o pomyślnej zmianie hasła

```
# Adresy URL logowania / wylogowania
path('login/', auth_views.LoginView.as_view(), name='login'),
path('logout/', auth_views.LogoutView.as_view(), name='logout'),

# 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'),

    # Adresy URL przeznaczone do obsługi procedury odzyskiwania 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('password-reset/<uidb64>/<token>/',
         auth_views.PasswordResetConfirmView.as_view(),
         name='password_reset_confirm'),
    path('password-reset/complete/',
         auth_views.PasswordResetCompleteView.as_view(),
         name='password_reset_complete'),

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

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

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

{% block title %}Odzyskiwanie hasła{% endblock %}

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

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

```
Otrzymaliśmy żądanie odzyskania 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 odzyskania hasła.

Wiadomość ta zawiera wygenerowany przez widok token, który jest potrzebny do ustawienia nowego hasła.

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

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

{% block title %}Odzyskiwanie hasła{% endblock %}

{% block content %}
  <h1>Odzyskiwania 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
  ↪wpisałeś adres e-mail podany podczas zakładania konta użytkownika.</p>
{% endblock %}
```

Utwórz kolejny plik szablonu w tym samym katalogu i nadaj mu nazwę *password_reset_confirm.html*. Następnie w nowym pliku umieść poniższy fragment kodu.

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

{% block title %}Odzyskiwanie hasła{% endblock %}

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

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

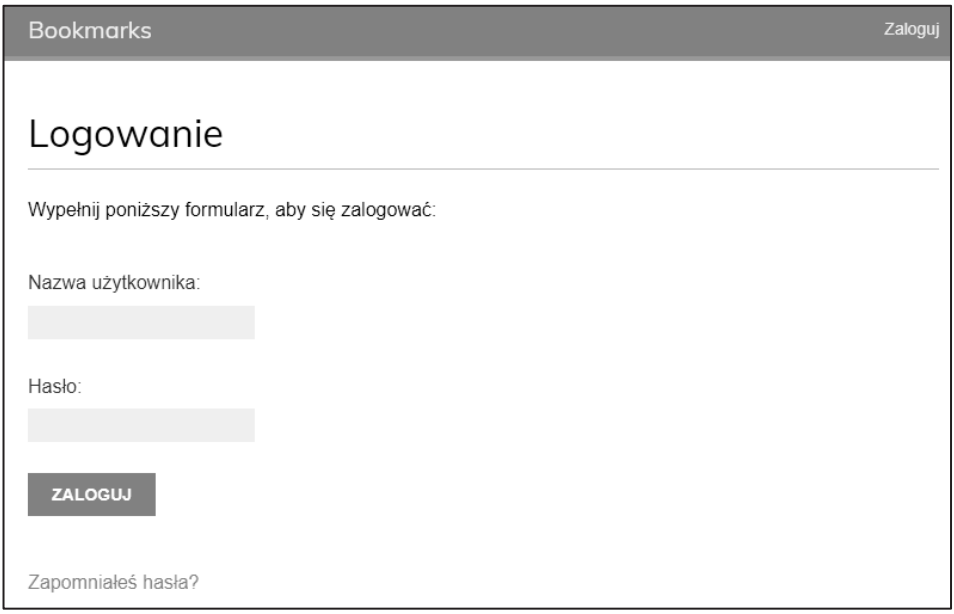
Utwórz kolejny plik szablonu i nadaj mu nazwę *password_reset_complete.html*. Następnie umieść w nim poniższy fragment kodu.

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

{% block title %}Odzyskiwanie hasła{% endblock %}

{% block content %}
  <h1>Ustawianie 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 poniższe wiersze wyróżnione pogrubioną czcionką.

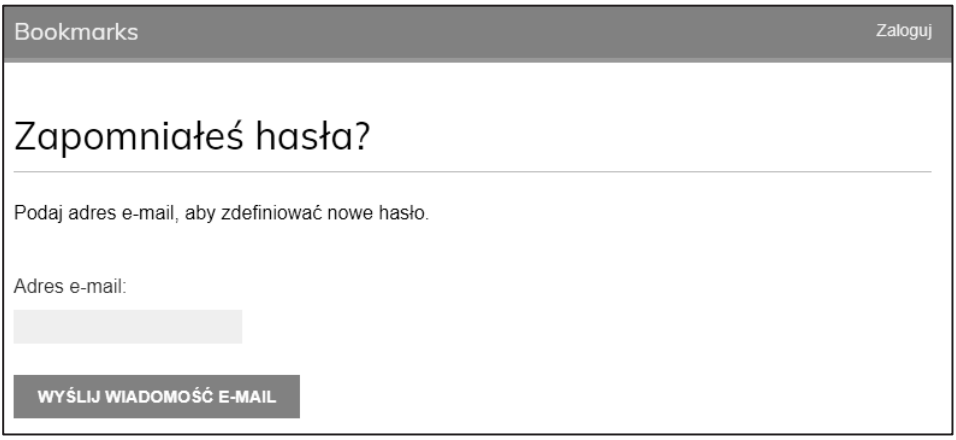
```
{% extends "base.html" %}

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

{% block content %}
  <h1>Logowanie</h1>
  {% if form.errors %}
    <p>
      Nazwa użytkownika lub hasło są niepoprawne.
      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>
    <p>
      <a href="{% url "password_reset" %}">
        Zapomniałeś hasła?
      </a>
    </p>
  </div>
{% endblock %}
```

Teraz w przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/*. Strona logowania powinna teraz zawierać łącze do strony odzyskiwania hasła (rysunek 4.11).



Rysunek 4.11. Strona logowania zawierająca łącze do strony odzyskiwania hasła

Kliknij łącze *Zapomniałeś hasła?*. Powinieneś zobaczyć stronę podobną do pokazanej na rysunku 4.12.



Rysunek 4.12. Formularz przywracania 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

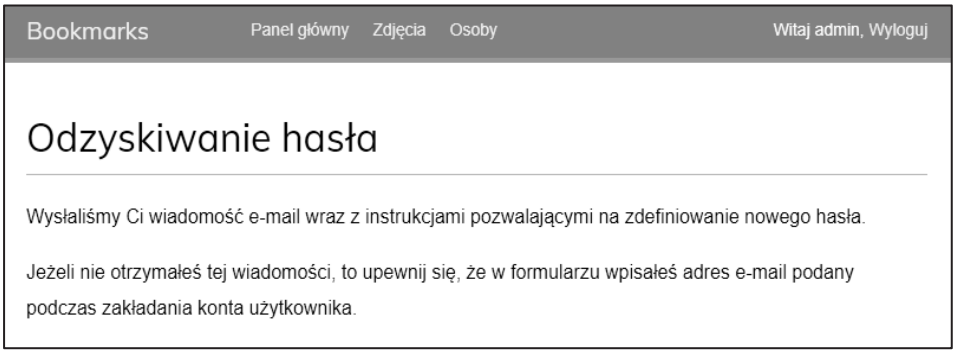
pomocą funkcji zaawansowanych". Jednak podczas pracy nad aplikacją można skonfigurować Django do przekazywania wiadomości e-mail na standardowe wyjście zamiast ich faktycznego wysyłania za pomocą serwera SMTP. Framework Django oferuje mechanizm backend do wyświetlania wiadomości e-mail w konsoli.

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

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

Ustawienie `EMAIL_BACKEND` wskazuje klasę, która będzie używana do wysyłania wiadomości e-mail.

Wróć do przeglądarki internetowej, podaj adres e-mail istniejącego użytkownika i kliknij przycisk *Wyślij e-mail*. Powinieneś zobaczyć stronę podobną do pokazanej na rysunku 4.13.



Rysunek 4.13. Strona z informacją o wysłaniu wiadomości e-mail dotyczącej odzyskiwania hasła

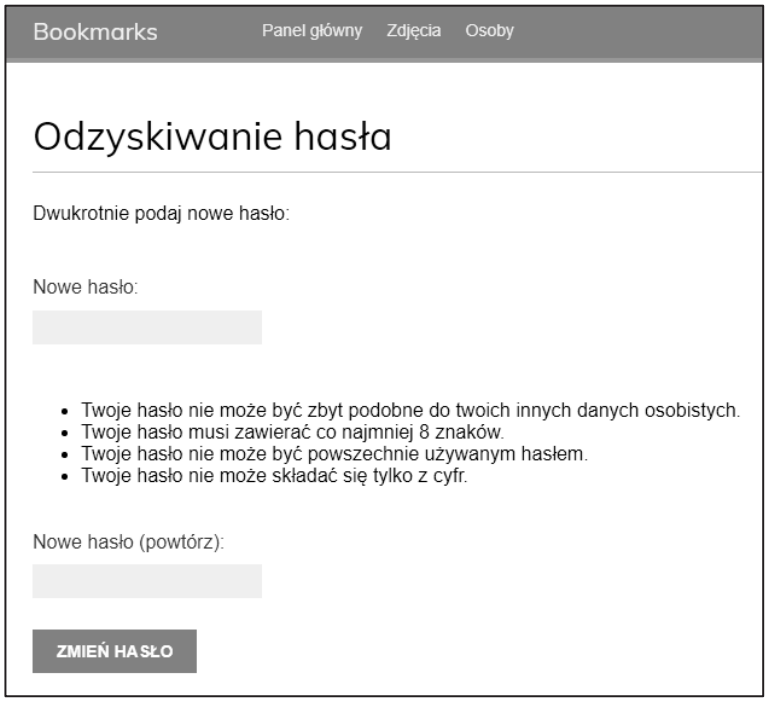
Spójrz na konsolę, na której został uruchomiony serwer programistyczny. Powinieneś zobaczyć wygenerowaną wiadomość e-mail.

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
Subject:
 =?utf-8?q?Reset_has=C5=82a_dla_konta_na_stronie_127=2E0=2E0=2E1=3A8000?=
From: webmaster@localhost
To: test@gmail.com
Date: Sat, 25 Feb 2023 12:57:29 -0000
Message-ID: <167732984913.11996.12138340043145121556@DESKTOP-DCOPFRB>

Otrzymaliśmy żądanie odzyskania hasła dla użytkownika używającego adresu
➥e-mail kowalski@op.pl. Kliknij poniższe łącze:
```

```
http://127.0.0.1:8000/account/password-reset/Mg/
↪bk6k3t-a46f68e5113ebe4762af23497afa7343/
Twoja nazwa użytkownika: test
```

Ta wiadomość e-mail została wygenerowana za pomocą utworzonego wcześniej szablonu *password_reset_email.html*. Adres URL pozwalający na przejście do strony odzyskiwania hasła zawiera token dynamicznie wygenerowany przez Django.

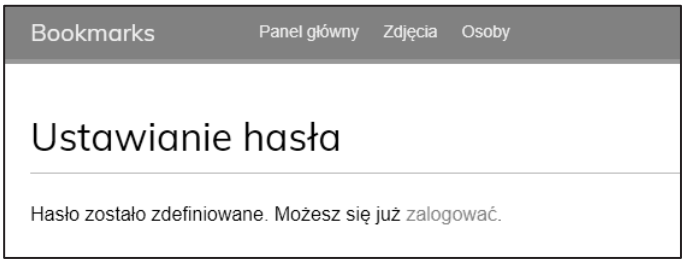
Skopiuj z wiadomości e-mail adres URL, który powinien wyglądać podobnie do następującego *http://127.0.0.1:8000/account/password-reset/MQ/ardx0u-b4973cfa2c70d652a190e79054bc479a/*, i otwórz go w przeglądarce. Powinieneś zobaczyć stronę podobną do pokazanej na rysunku 4.14.



Rysunek 4.14. Formularz odzyskiwania hasła

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 nowy skrót hasła i zapisze go w bazie danych. Jeżeli operacja przebiegnie bez problemów, zostanie wyświetlona pokazana poniżej strona wraz z komunikatem informującym o sukcesie (rysunek 4.15).

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



Rysunek 4.15. Strona z komunikatem o pomyślnym odzyskaniu hasła

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.

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. Teraz zastąpimy wzorce adresów URL uwierzytelniania wzorcami dostarczonymi przez 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. Nowy kod został wyróżniony pogrubioną czcionką.

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

urlpatterns = [
    # Poprzedni widok logowania
    # 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')

    # 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'),
```

```
    # Adresy URL przeznaczone do obsługi procedury odzyskiwania 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('password-reset/<uidb64>/<token>/',
    #     auth_views.PasswordResetConfirmView.as_view(),
    #     name='password_reset_confirm'),
    # path('password-reset/complete/',
    #    auth_views.PasswordResetCompleteView.as_view(),
    #    name='password_reset_complete'),

    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
]
```

Więcej informacji na temat wzorców adresów URL uwierzytelniania można znaleźć na stronie *https://github.com/django/django/blob/stable/4.0.x/django/contrib/auth/urls.py*

Dodaliśmy teraz do naszego projektu wszystkie niezbędne widoki uwierzytelniania. Następnie zaimplementujemy mechanizmy rejestracji użytkowników.

Rejestracja użytkownika i profile użytkownika

Istniejący użytkownicy mogą się zalogować, wylogować, zmienić hasło lub je odzyskać, jeśli go zapomną. 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 import forms
from django.contrib.auth.models import User
```

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


class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                               widget=forms.PasswordInput)

class Meta:
      model = User
      fields = ['username', 'first_name', 'email']
```

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 na przykład użytkownik wybierze już istniejącą nazwę użytkownika, otrzyma komunikat o błędzie w trakcie walidacji formularza, ponieważ pole `username` jest zdefiniowane z ustawieniem `unique=True`.

Dodaliśmy dwa nowe pola `password` i `password2` przeznaczone do zdefiniowania hasła i jego potwierdzenia. Dodajmy kod walidacji pola, aby sprawdzić, czy oba hasła są takie same.

Zmodyfikuj plik *forms.py* w aplikacji `account` i dodaj do klasy `UserRegistrationForm` zamieszczoną poniżej metodę `clean_password2()`. Nowy kod został wyróżniony pogrubioną czcionką.

```
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']
```

Zdefiniowaliśmy metodę `clean_password2()` w celu porównywania drugiego hasła z pierwszym i zgłoszenia błędu sprawdzania poprawności, jeśli hasła nie pasują. Metoda ta jest wywoływana podczas walidacji formularza za pomocą jego metody `is_valid()`.

Istnieje możliwość dostarczenia metody `clean_()` dla dowolnego pola formularza w celu wyczyszczenia jego wartości lub zgłoszenia błędu walidacji formularza dla określonego pola. Formularze zawierają także ogólną metodę `clean()` przeznaczoną do sprawdzenia całego formularza, co okazuje się użyteczne podczas walidacji pól zależnych wzajemnie od siebie. W tym przypadku zamiast zastępowania metody `clean()` formularza użyliśmy walidacji specyficznej dla pola `clean_password2()`. Pozwala to uniknąć zastępowania innych, specyficznych dla pól mechanizmów walidacji, które klasa `ModelForm` pobiera na podstawie mechanizmów kontroli (ang. *constraints*) ustawionych w modelu (na przykład sprawdzanie, czy nazwa użytkownika jest unikatowa).

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

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

```
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from django.contrib.auth.decorators import login_required
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`. Metoda ta obsługuje haszowanie hasła przed zapisaniem go w bazie danych.

Django nie przechowuje haseł w postaci zwykłego tekstu; zamiast tego przechowuje skróty (ang. *hashes*) haseł. Haszowanie to proces przekształcania podanego klucza w inną wartość (tzw. skrót). Funkcja skrótu służy do generowania wartości o stałej długości zgodnie z algorytmem matematycznym. Dzięki haszowaniu haseł za pomocą bezpiecznych algorytmów Django znacząco utrudnia złamanie haseł przechowywanych w bazie danych.

Domyślnie do przechowywania wszystkich haseł użytkowników Django używa algorytmu haszowania PBKDF2 ze skrótem SHA256. Framework obsługuje nie tylko sprawdzanie istniejących skrótów haseł obliczonych za pomocą algorytmu PBKDF2, ale także sprawdzanie przechowywanych skrótów haseł obliczonych za pomocą innych algorytmów, takich jak `PBKDF2SHA1`, `argon2`, `bcrypt` i `scrypt`.

Skróty haseł obsługiwane przez projekt Django są zdefiniowane za pomocą ustawienia `PASSWORD_HASHERS`. Oto domyślna lista `PASSWORD_HASHERS`.

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.ScryptPasswordHasher',
]
```

Do haszowania wszystkich haseł Django używa pierwszego wpisu na liście, w tym przypadku `PBKDF2PasswordHasher`. Reszty hasherów z listy Django może używać do sprawdzania istniejących haseł.

Ostrzeżenie

Hasher `scrypt` wprowadzono w Django 4.0. Jest bezpieczniejszy i bardziej zalecany niż `PBKDF2`. `PBKDF2` nadal jest jednak domyślnym hasherem, ponieważ `scrypt` wymaga zainstalowania OpenSSL 1.1+ i więcej pamięci.

Więcej informacji o sposobach przechowywania haseł w Django i o hasherach możesz znaleźć na stronie *https://docs.djangoproject.com/en/4.1/topics/auth/passwords/*.

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

```
urlpatterns = [

    # ...
```

```
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
]
```

Na koniec utwórz nowy szablon w katalogu *templates/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 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 w nowym pliku umieść poniższy fragment kodu.

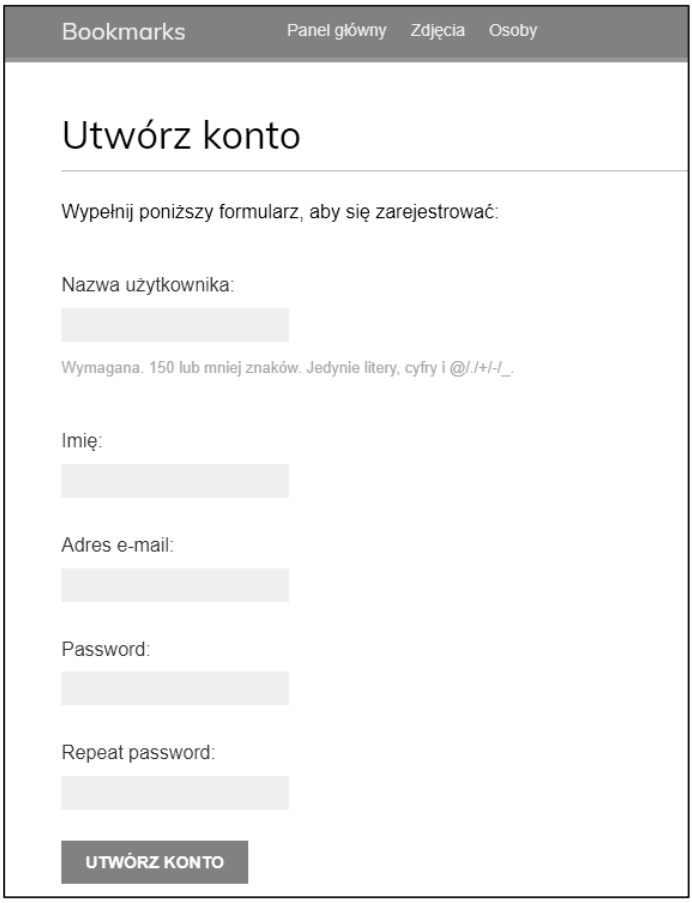
```
{% extends "base.html" %}

{% 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 %}
```

W przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/register/*. Powinieneś zobaczyć wyświetloną stronę rejestracji nowego użytkownika (rysunek 4.16).

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 4.17).

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



Rysunek 4.16. Formularz tworzenia konta

Rysunek 4.17. Strona z komunikatem o pomyślnym utworzeniu 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 przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/account/login/*. Wynik powinien być podobny do pokazanego na rysunku 4.18.

Rysunek 4.18. Strona logowania zawierająca łącze do rejestracji

Stworzyliśmy stronę rejestracji dostępną 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 standardowy model `User` zawiera ograniczony zestaw pól. Dlatego też prawdopodobnie będzie trzeba niekiedy rozbudować model o możliwość przechowywania dodatkowych danych.

Prostym sposobem na rozszerzenie modelu `User` jest utworzenie modelu profilu, który zawiera relację „jeden do jednego” z modelem `User` frameworka Django oraz wszelkie dodatkowe pola. Relacja „jeden do jednego” jest podobna do stosowania pola `ForeignKey`

z parametrem `unique = True`. Druga strona relacji to niejawna relacja „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 oznaczony pogrubioną czcionką.

```
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 f'Profil użytkownika {self.user.username}'
```

Ostrzeżenie

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/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model*.

Nasz profil użytkownika będzie zawierał datę urodzenia użytkownika oraz jego zdjęcie.

Do powiązania profili z użytkownikami służy pole relacji „jeden do jednego" `user`. Użyliśmy klauzuli `CASCADE` dla parametru `on_delete`, aby po usunięciu użytkownika został również usunięty powiązany z nim profil.

Pole `date_of_birth` jest typu `DateField`. Ustawiliśmy to pole jako opcjonalne za pomocą opcji `blank=True`, a dzięki opcji `null=True` dopuściliśmy wprowadzanie wartości `null`.

Zdjęcie użytkownika jest przechowywane w kolumnie `photo` typu `ImageField`. Ustawiliśmy to pole jako opcjonalne za pomocą opcji `blank=True`. Pole `ImageField` służy do przechowywania plików ze zdjęciami. Django sprawdza, czy podany plik jest poprawnym zdjęciem, zapisuje plik ze zdjęciem w katalogu wskazanym przez parametr `upload_to` i zapisuje względną ścieżkę do pliku w odpowiednim polu bazy danych. Pole `ImageField` w bazie danych jest domyślnie tłumaczone na kolumnę `VARHAR(100)`. Jeśli wartość pola jest pusta, w bazie danych zostanie zapisany pusty ciąg.

Instalowanie modułu Pillow i udostępnianie plików multimedialnych

W celu zarządzania obrazami konieczne będzie zainstalowanie pakietu Pythona `Pillow`. Moduł `Pillow` jest standardem de facto przetwarzania zdjęć w Pythonie. Obsługuje wiele formatów obrazów i zapewnia zaawansowane funkcje przetwarzania obrazu. Moduł `Pillow` jest wymagany przez Django do obsługi zdjęć zapisanych w polach typu `ImageField`.

Instalacja pakietu `Pillow` zostanie przeprowadzona po wydaniu w powłoce poniższego polecenia.

```
pip install Pillow==9.2.0
```

Przeprowadź edycję pliku *settings.py* projektu i dodaj w nim poniższe wiersze kodu.

```
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Wprowadzenie tych ustawień umożliwi frameworkowi Django zarządzanie przesyłaniem plików i udostępnianiem plików multimedialnych. 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. Ścieżki i adresy URL plików są budowane dynamicznie przez dodawanie do nich ścieżki projektu lub adresu URL multimediów, co ułatwia przenośność plików.

Teraz przeprowadź edycję pliku głównego *urls.py* projektu `bookmarks` i w następujący sposób zmodyfikuj znajdujący się w nim kod. Nowe wiersze zostały wyróżnione pogrubioną czcionką.

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

Do serwowania plików multimedialnych przez serwer programistyczny Django podczas programowania (to znaczy gdy opcja `DEBUG` jest ustawiona na `True`) dodaliśmy funkcję pomocniczą `static()`.

Ostrzeżenie

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 17., „Wdrożenie".

Tworzenie migracji dla modelu profilu

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':
  account/migrations/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']
    raw_id_fields = ['user']
```

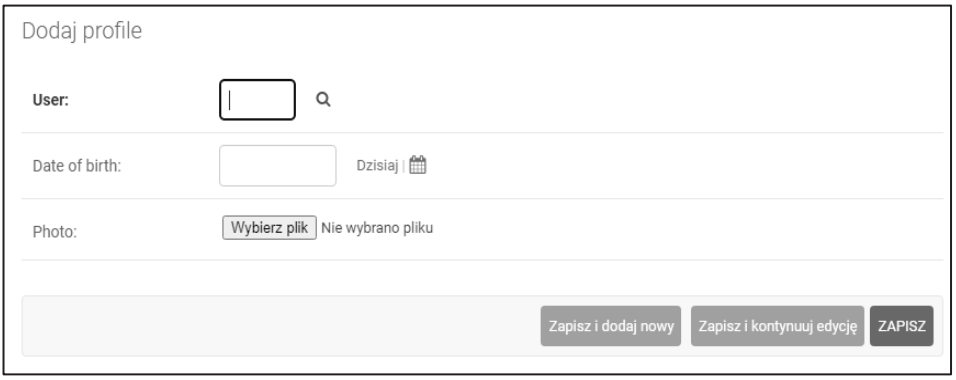
Uruchom serwer programistyczny za pomocą następującego polecenia:

```
python manage.py runserver
```

W przeglądarce internetowej przejdź pod adres *http://127.0.0.1:8000/admin*. Teraz będziesz mógł zobaczyć model Profile w witrynie administracyjnej projektu, co pokazałem na rysunku 4.19.

W wierszu *Profiles* kliknij łącze *Dodaj*. Zobaczysz formularz służący do dodawania nowego profilu, pokazany na rysunku 4.20.

Rysunek 4.19. Blok Account głównej strony witryny administracyjnej



Rysunek 4.20. Formularz dodawania profilu

Utwórz ręcznie obiekt `Profile` dla każdego istniejącego użytkownika w bazie danych.

Teraz zajmiemy się umożliwieniem użytkownikom edycji profilu w witrynie internetowej.

Przeprowadź edycję pliku *forms.py* w aplikacji `account` i umieść w nim poniższe wiersze oznaczone pogrubioną czcionką.

```
# ...
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* w aplikacji `account` i umieść w nim poniższe wiersze oznaczone pogrubioną czcionką.

```
# ...
from .models import Profile

# ...

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

Gdy użytkownicy rejestrują się w witrynie, tworzony jest obiekt profilu, który jest powiązany z utworzonym obiektem `User`.

Teraz umożliwimy użytkownikowi edycję profilu.

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

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

```
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm, \
                   UserEditForm, ProfileEditForm
from .models import Profile

# ...

@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})
```

Dodaliśmy nowy widok edycji, aby umożliwić użytkownikom modyfikowanie swoich danych osobowych. Dodaliśmy do widoku dekorator `login_required`, ponieważ tylko uwierzytelnieni użytkownicy będą mogli edytować swoje profile. W tym widoku używamy dwóch formularzy opartych na modelu `UserEditForm`, przeznaczonym do przechowywania danych wbudowanego modelu `User` i `ProfileEditForm`, przeznaczonym do przechowywania dodatkowych danych profilu. Aby zweryfikować poprawność danych wysłanych w formularzu, sprawdzamy, czy wartością zwrotną metody `is_valid()` w obu wymienionych formularzach jest `True`. Jeżeli oba formularze zawierają prawidłowe dane, zawartość obu formularzy zapisujemy przez wywołanie metody `save()` w celu uaktualnienia odpowiedniego obiektu w bazie danych.

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

```
urlpatterns = [
    #...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    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 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 elemencie HTML `<form>` musi znaleźć się opcja `enctype="multipart/form-data"`. Wykorzystujemy tylko jeden formularz HTML do wysłania obu formularzy Django, czyli `user_form` i `profile_form`.

Otwórz w przeglądarce stronę *http://127.0.0.1:8000/account/register/* i zarejestruj nowego użytkownika. Następnie zaloguj się jako nowy użytkownik i otwórz stronę *http://127.0.0.1:8000/account/edit/*. Powinieneś zobaczyć stronę podobną do pokazanej na rysunku 4.21.

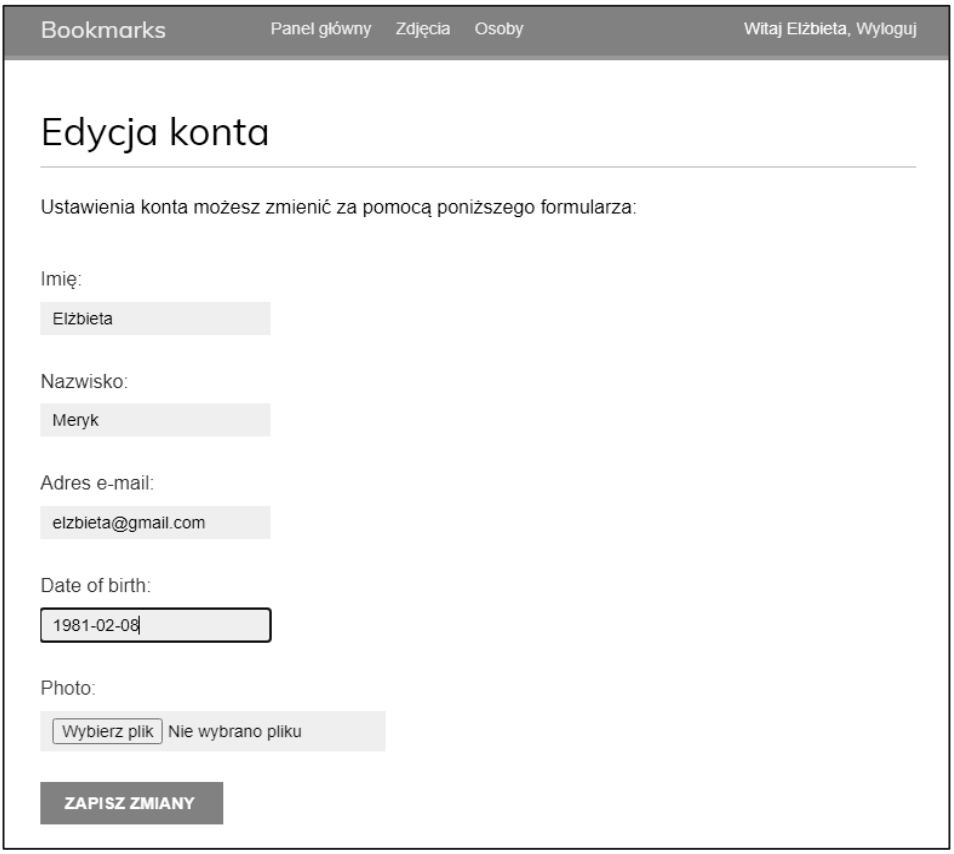
Możesz teraz dodać informacje o profilu i zapisać zmiany.

Teraz możemy zmodyfikować stronę panelu głównego i umieścić na niej łącza prowadzące do stron pozwalających na edycję profilu i zmianę hasła.

Otwórz w przeglądarce szablon *templates/account/dashboard.html* i dodaj następujące wiersze wyróżnione pogrubioną czcionką.

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

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

{% block content %}
  <h1>Panel główny</h1>
  <p>
  Witaj w panelu głównym. Możesz <a href="{% url "edit" %}">edytować
profil</a> lub <a href="{% url "password_change" %}">zmienić hasło</a>.
  </p>
{% endblock %}
```

Po wprowadzonych zmianach użytkownik będzie miał z poziomu panelu głównego dostęp do formularza umożliwiającego edycję profilu. Otwórz w przeglądarce stronę



Rysunek 4.21. Formularz edycji profilu

http://127.0.0.1:8000/account/ i przetestuj nowe łącze edycji profilu użytkownika (rysunek 4.22). Panel główny powinien teraz wyglądać następująco (rysunek 4.22).

Rysunek 4.22. 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/4.1/topics/auth/customizing/#substituting-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 messages

Podczas obsługi działań podejmowanych przez użytkowników może wystąpić konieczność informowania ich o skutkach podjętych przez nich czynności. Django oferuje wbudowany framework `messages`, 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`. Plik ustawień zawiera również oprogramowanie middleware `django.contrib.messages.middle`
`↪ware.MessageMiddleware`, umieszczone na liście `MIDDLEWARE` ustawień projektu.

Framework `messages` 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 `messages` 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()`: Działanie zakończyło się niepowodzeniem lub doszło do awarii.
- `debug()`: Komunikaty debugowania, które w środowisku produkcyjnym będą usunięte lub zignorowane.

Dodajmy do projektu obsługę komunikatów. Framework `messages` jest stosowany globalnie w projekcie. Do wyświetlania klientowi wszelkich dostępnych komunikatów użyjemy szablonu bazowego. Dzięki temu będziemy mogli informować klienta o wynikach wszelkich działań na dowolnej stronie.

Otwórz szablon *templates/base.html* aplikacji `account` i dodaj następujący kod wyróżniony pogrubioną czcionką.

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

Framework `messages` 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.

Ostrzeżenie

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 8., „Utworzenie sklepu internetowego”.

Teraz zmodyfikujmy widok edycji profilu, aby wykorzystać w nim framework `messages`.

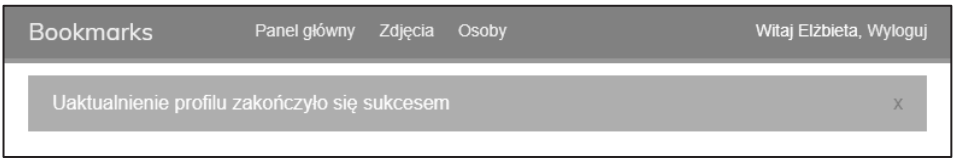
Przeprowadź edycję pliku *views.py* w aplikacji `account` i umieść w nim poniższe wiersze oznaczone pogrubioną czcionką.

```
# ...
from django.contrib import messages

# ...

@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()
            messages.success(request, 'Uaktualnienie profilu '\
                                      'zakończyło się sukcesem')
        else:
            messages.error(request, 'Wystąpił błąd podczas uaktualniania
            ↪profilu.')
    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})
```

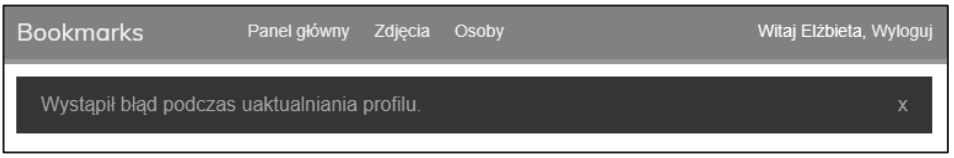
Dodaliśmy komunikat informujący o sukcesie wyświetlany, gdy operacja zmiany profilu zakończy się powodzeniem. Jeżeli którykolwiek formularz będzie wypełniony nieprawidłowo, nastąpi wyświetlenie komunikatu o błędzie.

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 4.23.

Wprowadź błędną datę w polu *Date of birth* i ponownie wyślij formularz. Powinieneś zobaczyć komunikat pokazany na rysunku 4.24.



Rysunek 4.23. Komunikat o pomyślnej edycji profilu



Rysunek 4.24. Komunikat o błędzie aktualizacji profilu

Generowanie komunikatów informujących użytkowników o wynikach ich działań jest bardzo proste. Komunikaty można dodawać również do innych widoków.

Więcej informacji na temat frameworka `messages` można znaleźć na stronie *https://docs.djangoproject.com/en/4.1/ref/contrib/messages*.

Po utworzeniu wszystkich funkcji związanych z uwierzytelnianiem użytkownika i edycją profilu zajmiemy się personalizacją mechanizmów uwierzytelniania. Dowiesz się, jak zbudować niestandardowy backend uwierzytelniania, aby użytkownicy mogli się logować do serwisu za pomocą swojego adresu e-mail.

Implementacja własnego backendu uwierzytelniania

Django pozwala na uwierzytelnianie względem wielu różnych źródeł. Opcja `AUTHENTI↪CATION_BACKENDS` to lista backendów 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 nadaje się do wykorzystania 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/4.1/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. Backend uwierzytelniania opiera się na klasie zapewniającej 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 w tym pliku umieść poniższy fragment kodu.

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

class EmailAuthBackend:
    """
    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, User.MultipleObjectsReturned):
            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 skrótu, którą następnie porównuje z wartością przechowywaną w bazie danych. Przechwytywane są dwa różne wyjątki dotyczące kolekcji `QuerySet`: `DoesNotExist` i `MultipleObjectsReturned`. Wyjątek `DoesNotExist` jest zgłaszany w przypadku, gdy nie zostanie znaleziony żaden użytkownik o podanym adresie e-mail. Wyjątek `MultipleObjectsReturned` jest zgłaszany w przypadku, gdy zostanie znalezionych wielu użytkowników z tym samym adresem e-mail. Później, aby uniemożliwić użytkownikom korzystanie z istniejącego adresu e-mail, zmodyfikujemy widoki rejestracji i edycji.
- `get_user()`:metoda pobiera użytkownika na podstawie identyfikatora podanego w parametrze `user_id`. Do pobrania obiektu `User` na czas trwania sesji użytkownika Django wykorzystuje backend, który uwierzytelnił użytkownika. `pk` to skrót od *primary key* (klucz główny), który jest unikatowym identyfikatorem dla każdego rekordu w bazie danych. Pole, które służy jako klucz główny, ma każdy model Django. Domyślnie kluczem głównym jest automatycznie wygenerowane pole `id`. Klucz podstawowy może być również określany jako `pk` wewnątrz mechanizmu ORM frameworka Django. Więcej informacji na temat automatycznych pól kluczy głównych można znaleźć pod adresem *https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields*.

Przeprowadź edycję pliku *settings.py* projektu i dodaj poniższy fragment kodu.

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

W powyższym ustawieniu zachowujemy domyślny mechanizm `ModelBackend`, który jest używany do uwierzytelniania za pomocą nazwy użytkownika i hasła, oraz dołączamy własny backend uwierzytelniania oparty na e-mailu `EmailAuthBackend`.

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 mechanizmu uwierzytelniania `EmailAuthBackend`.

Ostrzeżenie

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.

Uniemożliwianie użytkownikom korzystania z istniejącego adresu e-mail

Model `User` frameworka uwierzytelniania nie blokuje możliwości tworzenia użytkowników z tym samym adresem e-mail. Jeśli co najmniej dwa konta użytkowników mają ten sam adres e-mail, nie będziemy w stanie rozpoznać, który użytkownik się uwierzytelnia. Teraz, gdy użytkownicy mogą się logować przy użyciu swojego adresu e-mail, musimy zablokować im możliwość rejestracji przy użyciu istniejącego adresu e-mail.

Zmodyfikujemy teraz formularz rejestracji użytkownika, aby zapobiec rejestracji wielu użytkowników przy użyciu tego samego adresu e-mail.

Przeprowadź edycję pliku *forms.py* aplikacji `account` i umieść w nim poniższe wiersze oznaczone pogrubioną czcionką.

```
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']
```

```
    def clean_email(self):
        data = self.cleaned_data['email']
        if User.objects.filter(email=data).exists():
            raise forms.ValidationError('Ten adres e-mail już jest
            ↪używany.')
        return data
```

Dodaliśmy walidację pola `email`, która uniemożliwia użytkownikom rejestrację przy użyciu istniejącego adresu e-mail. Aby znaleźć istniejących użytkowników z tym samym adresem e-mail, budujemy kolekcję `QuerySet`. Za pomocą metody `exists()` sprawdzamy, czy są jakieś wyniki. Metoda `exists()` zwraca `True`, jeśli kolekcja `QuerySet` zawiera jakieś wyniki, i `False` w przeciwnym razie.

Teraz dodaj do klasy `UserEditForm` następujące wiersze wyróżnione pogrubioną czcionką.

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

    def clean_email(self):
        data = self.cleaned_data['email']
        qs = User.objects.exclude(id=self.instance.id)\
                         .filter(email=data)
        if qs.exists():
            raise forms.ValidationError(' Ten adres e-mail jest zajęty.')
        return data
```

W tym przypadku dodaliśmy walidację pola `email`, która uniemożliwia użytkownikom zmianę istniejącego adresu e-mail na istniejący adres e-mail innego użytkownika. Z kolekcji `QuerySet` wykluczamy bieżącego użytkownika. W przeciwnym razie aktualny adres e-mail użytkownika zostanie uznany za istniejący adres e-mail, a formularz nie zostanie zwalidowany.

Dodatkowe zasoby

Oto lista zasobów zawierających dodatkowe informacje związane z tematami omawianymi w tym rozdziale:

- Kod źródłowy przykładów zamieszczonych w tym rozdziale — *https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04*.
- Wbudowane widoki uwierzytelniania — *https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views*.
- Wzorce adresów URL uwierzytelniania — *https://github.com/django/django/blob/stable/3.0.x/django/contrib/auth/urls.py*.

- W jaki sposób Django zarządza hasłami i dostępnymi mechanizmami haszowania — *https://docs.djangoproject.com/en/4.1/topics/auth/passwords/*.
- Generyczny model użytkownika i metoda `get_user_model()` — *https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model*.
- Korzystanie z niestandardowego modelu użytkownika — *https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#substituting-a-custom-user-model*.
- Framework `messages` w Django — *https://docs.djangoproject.com/en/4.1/ref/contrib/messages/*.
- Niestandardowe źródła uwierzytelniania — *https://docs.djangoproject.com/en/4.1/topics/auth/ customizing/#other-authentication-sources*.
- Automatyczne pola klucza głównego — *https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields*.

Podsumowanie

W tym rozdziale dowiedziałeś się, jak zaimplementować w budowanej witrynie internetowej system uwierzytelniania. Zaimplementowałeś wszystkie widoki potrzebne do rejestracji użytkownika, logowania i wylogowania oraz edycji i odzyskiwania 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.

W następnym rozdziale dowiesz się, jak zaimplementować uwierzytelnianie społecznościowe na swojej stronie za pomocą frameworka `Social Auth` Pythona. Użytkownicy będą mogli się uwierzytelniać za pomocą swoich kont w serwisach Google, Facebook lub Twitter. Dowiesz się również, jak obsługiwać serwer programistyczny przez HTTPS przy użyciu Django Extensions. Nauczysz się ponadto personalizacji potoku uwierzytelniania w celu automatycznego tworzenia profili użytkowników.

Django: wypróbowany framework dla perfekcjonistów z napiętymi terminami!

Django służy do tworzenia aplikacji internetowych w Pythonie. Pozwala w pełni skorzystać z zalet tego języka, a przy tym jest łatwy do nauki. Praca z Django jest atrakcyjna dla programistów o różnym stopniu zaawansowania, co potwierdzają badania ankietowe serwisu Stack Overflow. Aby zapewnić swoim aplikacjom odpowiednią jakość, trzeba poznać sposób działania Django, stosować najlepsze praktyki, a także skutecznie wdrażać i testować aplikację.

Być może Django jest dla Ciebie zupełną nowością, a może posiadasz już pewną wiedzę na jego temat i chcesz wycisnąć z niego jak najwięcej — czwarte wydanie poświęconego mu podręcznika pomoże Ci opanować kluczowe umiejętności związane z obsługą tego frameworka. Pokazano tu techniki tworzenia kilku różnorodnych projektów, opisano przy tym krok po kroku wszystkie istotne etapy procesu rozwijania i wdrażania aplikacji bloga, serwisu społecznościowego, aplikacji e-commerce i platformy e-learningowej. Dowiesz się też, jak pomyślnie zastosować w swoich projektach takie technologie jak PostgreSQL, Redis, Celery, RabbitMQ i Memcached. Lektura przygotuje Cię do tego, co najistotniejsze podczas pracy z Django: budowania od podstaw poprawnie działających aplikacji. Jeśli programujesz w Pythonie i znasz przynajmniej w stopniu podstawowym HTML i JavaScript — to propozycja dla Ciebie.

W książce:

- podstawy Django, w tym modele, ORM, widoki, szablony, adresy URL, formularze, uwierzytelnianie, sygnały i oprogramowanie middleware
- integracja projektu aplikacji Django z zewnętrznym oprogramowaniem
- tworzenie aplikacji asynchronicznych (ASGI)
- konfiguracja środowiska produkcyjnego
- tworzenie złożonych aplikacji webowych i rozwiązywanie praktycznych problemów

Antonio Melé jest współzałożycielem i dyrektorem firmy Nucoro, która dostarcza platformy fintech. Pełni także funkcję CTO w Exo Investing, dostawcy cyfrowej platformy inwestycyjnej opartej na SI. Rozwija projekty Django dla klientów z różnych branż od 2006 roku. Jest uznanym mentorem start-upów na wczesnym etapie rozwoju.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-370-4
9 788383 223704
Cena: 129,00 zł
Packt