BEN SHAW | SAURABH BADHWAR | ANDREW BIRD
BHARATH CHANDRA K S | CHRIS GUEST

DJANGO

TWORZENIE NOWOCZESNYCH APLIKACJI INTERNETOWYCH W PYTHONIE

Tytuł oryginału: Web Development with Django: Learn to build modern web applications
with a Python-based framework

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-283-8364-7

Copyright © Packt Publishing 2021. First published in the English language under the title 'Web Development with Django – (9781839212505)'.

Polish edition copyright © 2022 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 13

Rozdział 1. Wprowadzenie do Django 27

Wstęp 28
Tworzenie szkieletu projektu i aplikacji Django 29
Ćwiczenie 1.1. Tworzenie projektu, aplikacji oraz serwera roboczego 30
Paradygmat Model View Template 33
Modele 33
Widoki 34
Szablony 34
Wstęp do HTTP 36
Przetwarzanie żądania 40
Projekt Django 41
Aplikacje Django 44
Konfiguracja programu PyCharm 45
Ćwiczenie 1.2. Konfiguracja projektu w programie PyCharm 46
Szczegółowe informacje o widokach 52
Mapowanie adresów URL 53
Ćwiczenie 1.3. Pisanie widoku i odwzorowania URL 54
GET, POST i obiekty QueryDict 57
Ćwiczenie 1.4. Sprawdzanie wartości GET i korzystanie z obiektu QueryDict 59
Analiza ustawień Django 61
Znajdowanie szablonów HTML w katalogach aplikacji 64
Ćwiczenie 1.5. Tworzenie katalogu templates oraz szablonu bazowego 64
Renderowanie szablonu za pomocą funkcji render 67
Ćwiczenie 1.6. Renderowanie szablonu w widoku 67
Renderowanie zmiennych w szablonach 69
Ćwiczenie 1.7. Użycie zmiennych w szablonach 70
Debugowanie i obsługa błędów 71
Wyjątki 72
Ćwiczenie 1.8. Generowanie i wyświetlanie wyjątków 73

Kup ksi k Pole ksi k

Debugowanie 75
Ćwiczenie 1.9. Debugowanie kodu 75
Zadanie 1.1. Tworzenie ekranu powitalnego witryny 79
Zadanie 1.2. Szkielet wyszukiwarki w witrynie Book 80
Podsumowanie **81**

Rozdział 2. Modele i migracje **83**

Wstęp **83**
Bazy danych **84**
Relacyjne bazy danych 85
Nierelacyjne bazy danych 85
Operacje bazodanowe z użyciem języka SQL 85
Typy danych w relacyjnych bazach danych 86
Ćwiczenie 2.1. Tworzenie bazy danych książek 86
Operacje CRUD w języku SQL **89**
Operacje tworzenia w języku SQL 90
Operacje odczytu w języku SQL 90
Operacje aktualizacji w języku SQL 91
Operacje usuwania w języku SQL 91
ORM w platformie Django 92
Konfiguracja bazy danych i tworzenie aplikacji Django 93
Aplikacje Django 94
Migracje Django 94
Tworzenie modeli i migracji Django 96
Typy pól 97
Opcje pól 98
Klucze główne 100
Relacje **101**
Wiele do jednego 102
Wiele do wielu 103
Relacje jeden do jednego 105
Dodawanie modelu Review 105
Metody modeli 106
Migracja aplikacji reviews 108
Operacje bazodanowe CRUD w Django **110**
Ćwiczenie 2.2. Tworzenie wpisu w bazie danych Bookr 110
Ćwiczenie 2.3. Tworzenie wpisu za pomocą metody create() 111
Tworzenie obiektu z kluczem obcym 112
Ćwiczenie 2.4. Tworzenie rekordów dla relacji wiele do jednego 112
Ćwiczenie 2.5. Tworzenie rekordów z relacjami wiele do wielu 114
Ćwiczenie 2.6. Relacja wiele do wielu z wykorzystaniem metody add() 115
Użycie metod create() i set() podczas tworzenia relacji 115
Operacje odczytu 116
Ćwiczenie 2.7. Pobieranie obiektu za pomocą metody get() 116
Zwracanie obiektu za pomocą metody get() 117
Ćwiczenie 2.8. Użycie metody all() do pobrania zbioru obiektów 118
Pobieranie obiektów za pomocą filtrowania 118
Ćwiczenie 2.9. Użycie metody filter() do pobierania obiektów 118
Filtrowanie za pomocą wyszukiwania pól 119

Kup ksi k Pole ksi k

Dopasowywanie wzorców w operacjach filtrowania 120
Pobieranie obiektów poprzez wykluczanie 120
Pobieranie obiektów za pomocą metody order_by() 121
Przeszukiwanie relacji 123
Wyszukiwanie na podstawie kluczy obcych 123
Przeszukiwanie na podstawie nazwy modelu 123
Przeszukiwanie relacji z kluczem obcym za pomocą instancji obiektu 123
Ćwiczenie 2.10. Znajdowanie obiektów na podstawie relacji wiele do wielu za pomocą wyszukiwania pola 124
Ćwiczenie 2.11. Przeszukiwanie relacji wiele do wielu za pomocą obiektów 124
Ćwiczenie 2.12. Przeszukiwanie relacji wiele do wielu za pomocą metody set() 125
Ćwiczenie 2.13. Użycie metody update() 125
Ćwiczenie 2.14. Użycie metody delete() 126
Zadanie 2.1. Tworzenie modeli dla aplikacji do zarządzania projektami 127
Wypełnianie danymi bazy danych projektu Bookr 127
Podsumowanie **128**

Rozdział 3. Mapowanie URL, widoki i szablony **129**

Wstęp **129**
Widoki oparte na funkcjach **130**
Widoki oparte na klasach **130**
Konfiguracja URL **131**
Ćwiczenie 3.1. Implementowanie prostego widoku opartego na funkcji 133
Szablony **135**
Ćwiczenie 3.2. Użycie szablonów do wyświetlenia komunikatu powitalnego 136
Język szablonów Django **138**
Ćwiczenie 3.3. Wyświetlanie listy książek i recenzji 140
Dziedziczenie szablonów 141
Stylowanie szablonów za pomocą biblioteki Bootstrap 143
Ćwiczenie 3.4. Dodawanie dziedziczenia szablonów i paska nawigacyjnego Bootstrapa 145
Zadanie 3.1. Implementacja widoku szczegółów książki 146
Podsumowanie **148**

Rozdział 4. Wstęp do witryny administracyjnej Django **149**

Wstęp **149**
Tworzenie konta superużytkownika **150**
Ćwiczenie 4.1. Tworzenie konta superużytkownika 151
Operacje CRUD za pomocą aplikacji administracyjnej Django **153**
Tworzenie 153
Pobieranie danych 154
Aktualizowanie 156
Usuwanie 158
Użytkownicy i grupy 159
Ćwiczenie 4.2. Dodawanie i modyfikowanie użytkowników i grup w aplikacji administracyjnej 159
Rejestrowanie modelu Reviews **164**
Listy obiektów do edycji 165
Strona edycji modelu Publisher 166

Kup ksi k Pole ksi k

Strona służąca do edycji książki 169
Ćwiczenie 4.3. Klucze obce i usuwanie z poziomu aplikacji administracyjnej 172
Dostosowywanie interfejsu administracyjnego 173
Poprawki dotyczące całej witryny administracyjnej Django 174
Analiza obiektu AdminSite w powłoce Pythona 174
Zadanie 4.1. Dostosowywanie obiektu SiteAdmin 178
Dostosowywanie klas ModelAdmin 180
Ćwiczenie 4.4. Dodawanie filtra na podstawie daty oraz hierarchii dat 188
Pasek wyszukiwania 191
Wykluczanie i grupowanie pól 193
Zadanie 4.2. Dostosowywanie aplikacji administracyjnych dla modeli 196
Podsumowanie 197

Rozdział 5. Zwracanie plików statycznych 199

Wstęp 199
Zwracanie plików statycznych 201
Wprowadzenie do wyszukiwarek plików statycznych 202
Wyszukiwarki plików statycznych — użycie podczas obsługi żądania 203
AppDirectoriesFinder 203
Przestrzenie nazw plików statycznych 204
Ćwiczenie 5.1. Zwracanie pliku z katalogu aplikacji 206
Generowanie statycznych adresów URL za pomocą znacznika szablonów static 210
Ćwiczenie 5.2. Użycie znacznika szablonów static 214
FileSystemFinder 217
Ćwiczenie 5.3. Zwracanie plików z katalogu static projektu 218
Wyszukiwarki plików statycznych — użycie polecenia collectstatic 220
Ćwiczenie 5.4. Kopiowanie plików statycznych dla środowiska produkcyjnego 222
Tryb STATICFILES_DIRS z przedrostkiem 223
Polecenie findstatic 225
Ćwiczenie 5.5. Znajdowanie plików poleceniem findstatic 226
Zwracanie ostatnich plików (w celu unieważnienia pamięci podręcznej) 229
Ćwiczenie 5.6. Eksploracja silnika przechowywania ManifestFilesStorage 231
Niestandardowe silniki magazynowania 234
Zadanie 5.1. Dodawanie logo do aplikacji reviews 236
Zadanie 5.2. Ulepszenia w stylach CSS 237
Zadanie 5.3. Dodawanie globalnego logo 239
Podsumowanie 241

Rozdział 6. Formularze 243

Wstęp 243
Czym jest formularz? 244
Element <form> 246
Rodzaje pól wejściowych 247
Ćwiczenie 6.1. Tworzenie formularza HTML 247
Bezpieczeństwo formularza dzięki ochronie przeciwko Cross-Site Request Forgery 255
Dostęp do danych w widoku 258
Ćwiczenie 6.2. Pobieranie danych POST w widoku 258
Wybór między żądaniami GET i POST 263
Dlaczego trzeba używać metody GET, jeśli można umieścić parametry w URL? 265

Kup ksi k Pole ksi k

Biblioteka Forms w Django 266
Definiowanie formularza 266
Renderowanie formularza w szablonie 274
Ćwiczenie 6.3. Tworzenie i renderowanie formularza Django 278
Walidacja formularzy i pobieranie wartości Pythona 282
Ćwiczenie 6.4. Walidacja formularza w widoku 286
Wbudowana walidacja pól 288
Ćwiczenie 6.5. Dodatkowa walidacja pól 289
Zadanie 6.1. Wyszukiwanie książek 291
Podsumowanie 294

Rozdział 7. Zaawansowana walidacja formularzy i formularzy modeli 295

Wstęp 295
Niestandardowa walidacja i czyszczenie pól 296
Niestandardowe walidatory 297
Metody oczyszczania 298
Walidacja na podstawie wielu pól 299
Ćwiczenie 7.1. Niestandardowe metody oczyszczania i walidacji 303
Wartości zastępcze i początkowe 310
Ćwiczenie 7.2. Wartości zastępcze i początkowe 312
Tworzenie i edytowanie modeli Django 314
Klasa ModelForm 315
Ćwiczenie 7.3. Tworzenie i edytowanie modelu Publisher 318
Zadanie 7.1. Stylowanie i integracja formularza modelu Publisher 324
Zadanie 7.2. Interfejs tworzenia instancji modelu Review 327
Podsumowanie 332

Rozdział 8. Zwracanie multimediów i przesyłanie plików 333

Wstęp 333
Ustawienia związane z przesyłaniem i zwracaniem plików multimedialnych 334
Zwracanie plików multimedialnych w środowisku roboczym 335
Ćwiczenie 8.1. Konfiguracja magazynu plików multimedialnych i ich zwracanie 335
Procesory kontekstu i użycie opcji MEDIA_URL w szablonach 338
Ćwiczenie 8.2. Ustawienia szablonu i użycie opcji MEDIA_URL w szablonach 340
Przesyłanie plików za pomocą formularzy HTML 343
Obsługa przesłanych plików w widoku 344
Ćwiczenie 8.3. Przesyłanie i pobieranie plików 347
Przesyłanie plików za pomocą formularzy Django 350
Ćwiczenie 8.4. Przesyłanie plików za pomocą formularza Django 352
Przesyłanie obrazów za pomocą formularzy Django 355
Zmiana rozmiaru obrazów za pomocą biblioteki Pillow 357
Ćwiczenie 8.5. Przesyłanie obrazów za pomocą formularzy Django 358
Zwracanie przesłanych (i innych) plików za pomocą Django 361
Magazynowanie plików w instancjach modeli 362
Zapisywanie obrazów w instancjach modeli 365
Korzystanie z klasy FieldFile 366
Odwoływanie się do plików multimedialnych w szablonach 371
Ćwiczenie 8.6. FileField i ImageField w modelach 371
Klasa ModelForm i przesyłanie plików 376

Kup ksi k Pole ksi k

Ćwiczenie 8.7. Przesyłanie plików i obrazów za pomocą instancji klasy ModelForm 377
Zadanie 8.1. Przesyłanie obrazu i plików PDF dotyczących książek 380
Zadanie 8.2. Wyświetlanie okładki i łącza do fragmentu książki 384
Podsumowanie 386

Rozdział 9. Sesje i uwierzytelnianie 387

Wstęp 387
Middleware 388
Moduły middleware 389
Implementacja widoków i szablonów do uwierzytelniania 390
Ćwiczenie 9.1. Zmiana przeznaczenia szablonu logowania aplikacji administracyjnej 394
Przechowywanie haseł w Django 397
Strona profilu i obiekt request.user 397
Ćwiczenie 9.2. Dodawanie strony profilu 397
Dekoratory uwierzytelniania i przekierowania 399
Ćwiczenie 9.3. Dodawanie dekoratorów uwierzytelniania do widoków 401
Dodawanie danych uwierzytelniania do szablonów 403
Ćwiczenie 9.4. Przełączanie łączy logowania i wylogowania w bazowym szablonie 404
Zadanie 9.1. Udostępnianie treści na podstawie stanu uwierzytelnienia za pomocą bloków warunkowych w szablonach 405
Sesje 407
Moduł pickle lub magazyn w formacie JSON 409
Ćwiczenie 9.5. Analiza klucza sesji 410
Przechowywanie danych w sesji 413
Ćwiczenie 9.6. Zapisywanie w sesji ostatnio wyświetlanych książek 413
Zadanie 9.2. Wykorzystanie magazynu sesji na stronie wyszukiwania książek 417
Podsumowanie 419

Rozdział 10. Zaawansowane aspekty aplikacji administracyjnej Django i jej dostosowywanie 420

Wstęp 421
Dostosowywanie witryny administracyjnej 421
Wykrywanie plików administracyjnych w Django 422
Klasa AdminSite w Django 423
Ćwiczenie 10.1. Tworzenie niestandardowej witryny administracyjnej w projekcie Bookr 424
Nadpisywanie domyślnej właściwości admin.site 426
Ćwiczenie 10.2. Nadpisywanie domyślnej witryny administracyjnej 427
Dostosowanie tekstu w witrynie administracyjnej za pomocą atrybutów AdminSite 428
Dostosowywanie szablonów witryny administracyjnej 429
Ćwiczenie 10.3. Dostosowanie szablonu wylogowania dla witryny administracyjnej Bookr 430
Dodawanie widoków do witryny administracyjnej 432
Tworzenie nowej funkcji widoku 433
Dostęp do wspólnych zmiennych szablonu 433

Kup książkę Poleć książkę

Mapowanie adresów URL na niestandardowy widok 434
Ograniczanie niestandardowych widoków do witryny administracyjnej 435
Ćwiczenie 10.4. Dodawanie niestandardowych widoków do witryny administracyjnej 435
Przekazywanie dodatkowych kluczy do szablonów za pomocą zmiennych szablonów 438
Zadanie 10.1. Tworzenie niestandardowego interfejsu administracyjnego z wbudowaną wyszukiwarką 439
Podsumowanie **440**

Rozdział 11. Zaawansowane aspekty szablonów i widoki oparte na klasach **441**

Wstęp **441**
Filtry szablonów **442**
Niestandardowe filtry szablonów **443**
Filtry szablonów 444
Konfiguracja katalogu służącego do zapisywania filtrów szablonów 444
Konfiguracja biblioteki szablonów 444
Implementowanie niestandardowej funkcji filtra 445
Użycie niestandardowych filtrów w szablonach 445
Ćwiczenie 11.1. Tworzenie niestandardowego filtra szablonów 446
Filtry tekstowe 448
Znaczniki szablonów **449**
Typy znaczników szablonów 449
Proste znaczniki 449
Tworzenie prostych znaczników szablonów 450
Ćwiczenie 11.2. Tworzenie niestandardowego prostego znacznika 451
Znaczniki włączające 454
Ćwiczenie 11.3. Budowanie niestandardowego znacznika włączającego 455
Widoki Django **458**
Widoki oparte na klasach **458**
Ćwiczenie 11.4. Tworzenie katalogu książek w widoku opartym na klasach 460
Operacje CRUD za pomocą widoków opartych na klasach 464
Zadanie 11.1. Renderowanie szczegółów na stronie profilu użytkownika za pomocą znaczników włączających 469
Podsumowanie **470**

Rozdział 12. Tworzenie API REST-owego **471**

Wstęp **471**
API REST-owe **472**
Django REST Framework 473
Instalacja i konfiguracja 473
Widoki API oparte na funkcjach 473
Ćwiczenie 12.1. Tworzenie prostego API REST-owego 474
Serializery **475**
Ćwiczenie 12.2. Tworzenie widoku API w celu wyświetlenia listy książek 476
Widoki API oparte na klasach i widoki generyczne 479
Serializery modeli 479

Kup ksi k Pole ksi k

Ćwiczenie 12.3. Tworzenie widoków API opartych na klasach i serializerów modeli 480
Zadanie 12.1. Tworzenie punktu końcowego API dla strony poświęconej najaktywniejszym współautorom 481
Obiekty typu ViewSet **484**
Routery **484**
Ćwiczenie 12.4. Używanie zbiorów widoków i routerów 485
Uwierzytelnianie **487**
Uwierzytelnianie oparte na tokenach 489
Ćwiczenie 12.5. Implementowanie uwierzytelniania opartego na tokenach w API aplikacji Bookr 489
Podsumowanie **494**

Rozdział 13. Generowanie plików CSV, PDF i innych plików binarnych **495**

Wstęp **495**
Obsługa plików CSV w Pythonie 496
Korzystanie z modułu csv Pythona **496**
Odczytywanie danych z pliku CSV 497
Ćwiczenie 13.1. Odczyt pliku CSV w Pythonie 497
Zapisywanie danych do plików CSV za pomocą Pythona 499
Ćwiczenie 13.2. Generowanie pliku CSV za pomocą modułu Pythona csv 501
Lepszy sposób odczytu i zapisu plików CSV 503
Przetwarzanie plików Excela w Pythonie **505**
Eksportowanie danych do plików binarnych 506
Obsługa plików XLSX za pomocą pakietu XlsxWriter 506
Ćwiczenie 13.3. Tworzenie plików XLSX w Pythonie 508
Obsługa plików PDF w Pythonie **511**
Przekształcanie stron WWW do formatu PDF 511
Ćwiczenie 13.4. Generowanie dokumentu PDF na podstawie strony WWW w Pythonie 512
Tworzenie wykresów w Pythonie **514**
Generowanie wykresów za pomocą biblioteki plotly 514
Ćwiczenie 13.5. Generowanie wykresów w Pythonie 515
Integrowanie biblioteki plotly z Django 518
Integrowanie wizualizacji z Django **518**
Ćwiczenie 13.6. Wizualizacja historii przeczytanych książek na stronie profilowej użytkownika 518
Zadanie 13.1. Eksportowanie książek przeczytanych przez użytkownika do pliku XLSX 523
Podsumowanie **523**

Rozdział 14. Testowanie **525**

Wstęp **525**
Dlaczego testowanie jest ważne **526**
Testy automatyczne **526**
Testowanie w Django **527**
Implementowanie przypadków testowych 527
Testy jednostkowe w Django 528
Korzystanie z asercji 528

Kup ksi k   Pole ksi k

Ćwiczenie 14.1. Pisanie prostego testu jednostkowego 529
Konfiguracja przed testami i czyszczenie po wykonaniu każdego przypadku testowego 531
Testowanie modeli Django 532
Ćwiczenie 14.2. Testowanie modeli Django 532
Testowanie widoków Django 536
Ćwiczenie 14.3. Pisanie przypadków testowych dla widoków Django 536
Testowanie widoków wymagających uwierzytelniania 539
Ćwiczenie 14.4. Pisanie przypadków testowych w celu walidacji uwierzytelnionych użytkowników 539
Klasa RequestFactory Django 542
Ćwiczenie 14.5. Testowanie widoków za pomocą klasy RequestFactory 543
Testowanie widoków opartych na klasach 545
Klasy przypadków testowych w Django 545
SimpleTestCase 546
TransactionTestCase 546
LiveServerTestCase 546
Modularyzacja kodu testowego 547
Zadanie 14.1. Testowanie modeli i widoków w projekcie Bookr 547
Podsumowanie 548

Rozdział 15. Zewnętrzne biblioteki Django 549

Wstęp 549
Zmienne środowiskowe 550
django-configurations 553
Zmiany w pliku manage.py 555
Konfiguracja ze zmiennych środowiskowych 556
Ćwiczenie 15.1. Konfiguracja biblioteki django-configurations 557
dj-database-url 560
Ćwiczenie 15.2. Konfiguracja biblioteki dj-database-url 563
Django Debug Toolbar 565
Ćwiczenie 15.3. Konfiguracja narzędzia Django Debug Toolbar 580
django-crispy-forms 584
Filtr crispy 585
Znacznik szablonów crispy 587
Ćwiczenie 15.4. Użycie biblioteki Django Crispy Forms z formularzem SearchForm 589
django-allauth 592
Inicjalizacja uwierzytelniania za pomocą biblioteki django-allauth 597
Zadanie 15.1. Aktualizacja formularza z wykorzystaniem klasy FormHelper 598
Podsumowanie 600

Rozdział 16. Używanie frontendowej biblioteki JavaScriptu z Django 601

Wstęp 601
Platformy JavaScriptu 602
Wprowadzenie do JavaScriptu 604
React 609
Komponenty 610
Ćwiczenie 16.1. Konfiguracja przykładowej strony z wykorzystaniem Reacta 614

Kup książkę Poleć książkę

JSX 617
Ćwiczenie 16.2. JSX i Babel 618
Właściwości JSX 619
Ćwiczenie 16.3. Właściwości komponentu Reacta 620
Obiekty Promise w JavaScripcie 622
fetch 623
Ćwiczenie 16.4. Pobieranie i wyświetlanie książek 626
Znacznik szablonów verbatim 630
Zadanie 16.1. Podgląd recenzji 630
Podsumowanie 635

Dodatek A 637

Skorowidz 710

Kup ksi k Pole ksi k

1

ROZDZIAŁ

Wprowadzenie do Django

PRZEGLĄD ROZDZIAŁU

W tym rozdziale zapoznasz się z Django oraz dowiesz się, jaką rolę pełni w tworzeniu aplikacji internetowych. Najpierw dowiesz się, na czym polega paradygmat *MVT* (*Model View Template*, czyli *Model Widok Szablon*) oraz jak Django przetwarza *żądania* i *odpowiedzi* HTTP. Po zapoznaniu się z podstawowymi koncepcjami utworzysz pierwszy projekt Django o nazwie *Bookr*. Będzie to aplikacja służąca do dodawania i przeglądania recenzji książek oraz do zarządzania nimi, którą będziesz rozwijał w trakcie czytania tej książki. Następnie poznasz polecenie `manage.py` (służące do zarządzania działaniami Django). Za jego pomocą uruchomisz serwer roboczy Django i sprawdzisz, czy napisany kod działa poprawnie. Nauczysz się też korzystać z programu *PyCharm*, popularnego środowiska IDE wykorzystywanego do programowania w Pythonie. W trakcie pracy z książką będziesz w nim pisać kod zwracający *odpowiedź* do przeglądarki internetowej. Na koniec nauczysz się korzystać z debugera programu PyCharm, ułatwiającego rozwiązywanie problemów występujących w kodzie. Na końcu tego rozdziału będziesz wiedział, jak zacząć tworzyć projekty w Django.

Kup ksi k   Pole ksi k

Wstęp

„Platforma internetowa dla perfekcjonistów, których gonią terminy". To hasło trafnie opisuje Django, platformę dostępną już od ponad 10 lat. Została przetestowana w praktyce, jest szeroko rozpowszechniona i wykorzystywana przez coraz większą liczbę osób. Być może uważasz, że Django jest już stare i bez znaczenia. Wręcz przeciwnie, długa obecność w branży dowiodła, że jego interfejs API (Application Programming Interface) jest niezawodny i spójny, a nawet programiści, którzy korzystali z Django w wersji 1.0 w 2007 roku, mogą pisać bardzo podobny kod w nowoczesnym Django 3. Django jest stale rozwijane, a co miesiąc wydawane są poprawki błędów i zabezpieczeń.

Podobnie jak język Python, w którym napisano Django, platforma jest łatwa do nauki, a zarazem ma ogromne możliwości i jest wystarczająco elastyczna, aby sprostać rosnącym wymaganiom programisty. Jest to platforma, do której „dołączono baterie", co oznacza, że nie trzeba szukać i instalować dodatkowych bibliotek lub komponentów, aby napisać i uruchomić aplikację. Inne platformy, takie jak *Flask* lub *Pylons*, wymagają samodzielnej instalacji zewnętrznych platform służących do tworzenia połączeń z bazą danych lub renderowania szablonów. Natomiast Django zawiera wbudowaną obsługę zapytań baz danych, mapowania adresów URL i renderowania szablonów (niebawem opiszemy szczegóły tych mechanizmów). Łatwość użycia Django nie oznacza jednak, że jego możliwości są ograniczone. Django jest używane w wielu wielkich witrynach, np. Disqus (*https://disqus.com/*), Instagram (*https://www.instagram.com/*), Mozilla (*https://www.mozilla.org/*), Pinterest (*https://www.pinterest.com/*), Open Stack (*https://www.openstack.org/*) i National Geographic (*http://www.nationalgeographic.com/*).

Jaką rolę Django pełni w ekosystemie internetowym? Wśród platform WWW można wyróżnić platformy frontendowe napisane w języku JavaScript, np. ReactJS, Angular lub Vue. Te platformy ulepszają i zwiększają możliwości interakcji z wygenerowanymi stronami WWW. Django działa w warstwie znajdującej się za tymi narzędziami i jest odpowiedzialne za trasowanie adresów URL, pobieranie danych z baz danych, renderowanie szablonów i obsługę danych wejściowych od użytkowników. Nie oznacza to jednak konieczności dokonania wyboru między nimi; platformy JavaScript mogą wzbogacić wynik działania Django lub korzystać z API REST-owego generowanego przez Django.

W tej książce utworzysz projekt Django, korzystając z metod wykorzystywanych na co dzień przez programistów Django. Aplikacja ta nosi nazwę *Bookr* i umożliwia przeglądanie i dodawanie książek oraz recenzji. Ta książka jest podzielona na cztery części. W pierwszej z nich poznasz podstawy tworzenia szkieletu aplikacji Django, szybko utworzysz kilka stron i zwrócisz je za pomocą serwera roboczego Django. Będziesz mógł dodać dane do bazy danych za pomocą witryny administracyjnej Django.

W następnej części ulepszysz aplikację Bookr. Zwrócisz statyczne pliki i dodasz style oraz grafikę do witryny. Korzystając z biblioteki `form` Django, zadbasz o interakcje, a dzięki funkcji przesyłania plików dodasz możliwość przesyłania obrazów okładek książek i innych plików. Następnie zaimplementujesz logowanie użytkownika i dowiesz się, jak przechowywać w sesji informacje o bieżącym użytkowniku.

Kup ksi k

Pole ksi k

W części trzeciej skorzystasz z dotychczasowych umiejętności i przejdziesz do następnego poziomu programowania. Dostosujesz witrynę administracyjną Django, a następnie poznasz tajniki zaawansowanych szablonów. Później dowiesz się, jak tworzyć *API REST-owe* i generować dane w formacie innym niż HTML (np. CSV i PDF). Tę część zakończysz, ucząc się technik testowania Django.

Istnieje wiele zewnętrznych bibliotek zwiększających możliwości Django, ułatwiających tworzenie aplikacji, a tym samym pozwalających zaoszczędzić czas programistom. W ostatniej części poznasz kilka przydatnych bibliotek i dowiesz się, jak je zintegrować z aplikacją. Korzystając z tych umiejętności, dodasz bibliotekę JavaScriptu służącą do komunikacji z platformą REST-ową utworzoną w poprzedniej części. Na koniec dowiesz się, jak wdrożyć aplikację Django na serwerze wirtualnym.

Gdy skończysz czytać tę książkę, będziesz mieć wystarczająco duże doświadczenie, aby zaprojektować i utworzyć własny projekt Django od początku do końca.

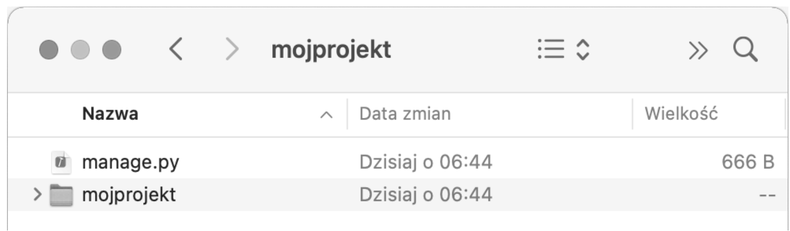
Tworzenie szkieletu projektu i aplikacji Django

Zanim zagłębimy się w teorię stojącą u podstaw paradygmatu Django i żądań HTTP, pokażemy, jak łatwo można utworzyć i uruchomić projekt Django. Po zakończeniu tej części i ćwiczenia będziesz dysponować projektem Django, wyślesz do niego żądanie w przeglądarce i zobaczysz odpowiedź.

Projekt Django jest katalogiem zawierającym wszystkie dane potrzebne w projekcie: kod, ustawienia, szablony i zasoby. Tworzy się go za pomocą programu wiersza poleceń `django-admin.py` z argumentem `startproject` i nazwą projektu. Aby utworzyć projekt Django o nazwie `mojprojekt`, należy wykonać następujące polecenie:

```
django-admin.py startproject mojprojekt
```

To polecenie utworzy katalog *mojprojekt*, w którym Django utworzy pliki potrzebne do uruchomienia projektu. W katalogu *mojprojekt* znajdują się dwa pliki (zobacz rysunek 1.1).



Rysunek 1.1. Katalog projektu mojprojekt

manage.py jest skryptem napisanym w Pythonie, który jest wykonywany w wierszu poleceń i służy do zarządzania projektem. Za jego pomocą można uruchomić *serwer roboczy Django*, czyli serwer WWW służący do obsługi projektu na komputerze lokalnym. Podobnie jak w przypadku

Kup ksi k
Pole ksi k

skryptu *django-admin.py*, polecenia przekazuje się w wierszu poleceń. W przeciwieństwie do skryptu *django-admin.py* ten skrypt nie jest zmapowany na ścieżki systemowe, a zatem trzeba go uruchamiać za pomocą Pythona. Oto przykładowe polecenie, które trzeba wykonać w katalogu projektu:

```
python3 manage.py runserver
```

W tym przykładzie przekazujemy polecenie `runserver` do skryptu *manage.py*, aby uruchomić serwer roboczy Django. W punkcie „Projekt Django" opisujemy więcej poleceń obsługiwanych przez skrypt *manage.py*. Za pomocą tego skryptu można wykonywać polecenia administracyjne, a zatem można uznać, że w tym przykładzie „wykonujemy polecenie administracyjne `runserver`".

Polecenie `startproject` utworzyło również katalog o nazwie projektu, czyli *mojprojekt* (rysunek 1.1). Jest to pakiet Pythona zawierający ustawienia i pewne pliki konfiguracyjne, które projekt musi uruchomić. Zawartość tego pakietu omawiamy w punkcie „Projekt Django".

Po uruchomieniu projektu Django trzeba uruchomić aplikację Django. Projekt Django warto podzielić na kilka aplikacji grupujących powiązane ze sobą funkcje. Np. w projekcie Bookr utworzysz aplikację `reviews`. Będzie ona zawierać cały kod, HTML, zasoby i klasy bazy danych potrzebne do obsługi recenzji książek. Jeśli zdecydujesz się rozszerzyć projekt Bookr i wykorzystać go do sprzedaży książek, możesz utworzyć aplikację `store` zawierającą pliki potrzebne do obsługi księgarni. Aplikacje tworzy się poleceniem administracyjnym `startapp`, do którego trzeba przekazać nazwę aplikacji, jak w następującym przykładzie:

```
python3 manage.py startapp mojaplikacja
```

W ten sposób w katalogu projektu zostanie utworzony katalog aplikacji (*mojaplikacja*). Django automatycznie umieści w nim pliki aplikacji, w których należy napisać potrzebny kod. Te pliki omawiamy w punkcie „Aplikacje Django", w którym także opisujemy cechy, jakie powinna posiadać dobra aplikacja.

Po zapoznaniu się z podstawowymi poleceniami potrzebnymi do utworzenia szkieletu projektu i aplikacji Django możesz z nich skorzystać, aby rozpocząć pracę nad projektem Bookr.

Ćwiczenie 1.1. Tworzenie projektu, aplikacji oraz serwera roboczego

Podczas pracy z tą książką utworzysz witrynę Bookr, poświęconą recenzjom książek. Będzie w niej można dodać pola dotyczące wydawców, współtwórców, książek i recenzji. Wydawca będzie mógł wydać co najmniej jedną książkę, a każda książka będzie miała co najmniej jednego współtwórcę (autora, redaktora, współautora, itd.). Tylko administratorzy będą mogli modyfikować te pola. Użytkownicy, którzy zarejestrują się w witrynie, będą mogli dodawać recenzje książek.

W tym ćwiczeniu utworzysz szkielet projektu Django `bookr`, uruchomisz serwer roboczy, aby sprawdzić, czy Django działa, a następnie utworzysz aplikację Django `reviews`.

Kup ksi k Pole ksi k

Powinieneś już dysponować środowiskiem wirtualnym z zainstalowaną platformą Django. We „Wstępie” możesz sprawdzić, jak to zrobić. Gdy będziesz gotowy, zacznij od utworzenia projektu Bookr.

1. Otwórz terminal i wykonaj następujące polecenie, aby utworzyć katalog projektu `bookr` wraz z domyślnymi podkatalogami:

   ```
   django-admin startproject bookr
   ```

   To polecenie nie zwróci żadnych wyników, ale utworzy folder o nazwie *bookr* w katalogu, w którym je wykonasz. Możesz zajrzeć do tego katalogu i sprawdzić, czy zawiera elementy opisane w przykładzie dotyczącym projektu `mojprojekt`: katalog pakietu *bookr* i plik *manage.py*.

2. Możesz już sprawdzić, czy projekt i Django są poprawnie skonfigurowane. W tym celu uruchom serwer roboczy za pomocą skryptu *manage.py*.

   W terminalu (lub w wierszu poleceń) przejdź do katalogu projektu *bookr* (poleceniem `cd`), a następnie wykonaj polecenie `manage.py runserver`.

   ```
   python3 manage.py runserver
   ```

> W systemie Windows we wszystkich poleceniach zastąp słowo **`python3`** (pogrubione) słowem `python`.

   To polecenie uruchomi serwer roboczy Django. Powinieneś uzyskać wynik podobny do następującego:

   ```
   Watching for file changes with StatReloader
   Performing system checks...

   System check identified no issues (0 silenced).

   You have 17 unapplied migration(s). Your project may not work properly until
   you apply the migrations for app(s): admin, auth, contenttypes, sessions.
   Run 'python manage.py migrate' to apply them.

   September 14, 2019 - 09:40:45
   Django version 3.0a1, using settings 'bookr.settings'
   Starting development server at http://127.0.0.1:8000/
   Quit the server with CONTROL-C.
   ```

   Prawdopodobnie zobaczysz ostrzeżenia o pominiętych migracjach, ale nie musisz się tym na razie przejmować.

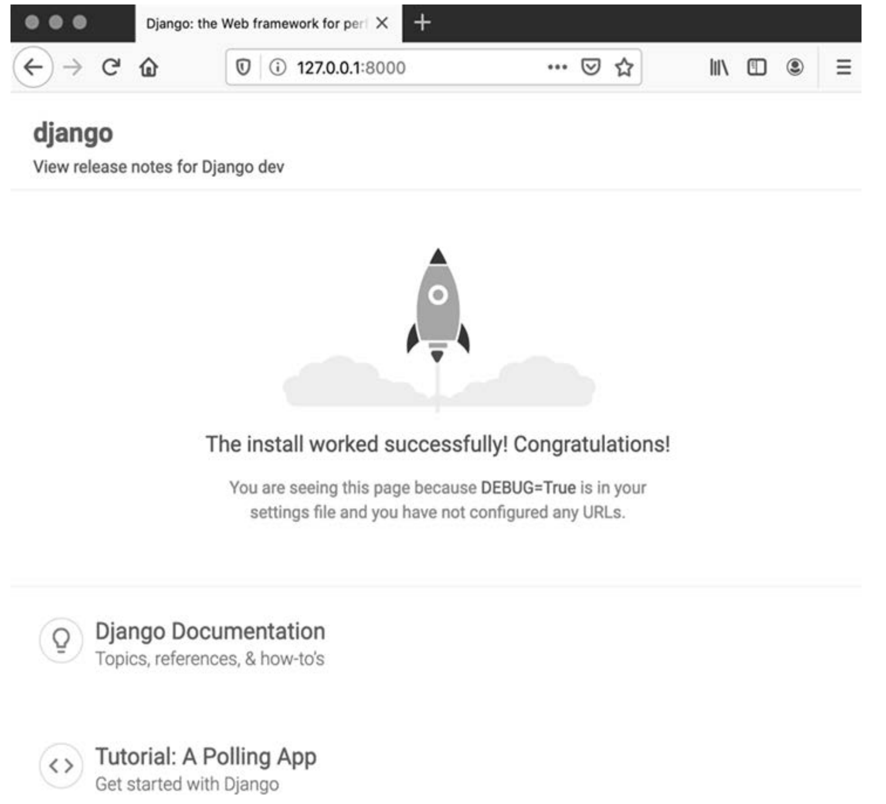
3. Otwórz przeglądarkę i stronę *http://127.0.0.1:8000/*. Powinieneś zobaczyć ekran powitalny Django (zobacz rysunek 1.2). Jeśli ekran zostanie wyświetlony, będzie wiadomo, że projekt Django został poprawnie utworzony i uruchomiony.

4. Powróć do terminala i zatrzymaj serwer roboczy, naciskając klawisze *Ctrl+C*.

5. Teraz można utworzyć aplikację *reviews* dla projektu *bookr*. W terminalu upewnij się, że znajdujesz się w katalogu projektu *bookr*, a następnie wykonaj następujące polecenie, aby utworzyć aplikację *reviews*:

   ```
   python3 manage.py startapp reviews
   ```


Kup ksi k Pole ksi k



Rysunek 1.2. Ekran powitalny Django

> Po utworzeniu aplikacji *reviews* katalog projektu *bookr* powinien zawierać pliki znajdujące się w katalogu *Chapter01/Exercise1.01* w pakiecie dołączonym do książki.

Jeśli polecenie zakończy się powodzeniem, na ekranie nie zostaną wyświetlone żadne dane, natomiast zostanie utworzony katalog aplikacji *reviews*. Możesz zajrzeć do tego katalogu, aby sprawdzić wygenerowaną zawartość. Powinieneś zobaczyć katalog *migrations*, pliki *admin.py*, *models.py* i inne. Omówimy je szczegółowo w punkcie „Aplikacje Django".

W tym ćwiczeniu utworzyłeś projekt *bookr*, sprawdziłeś, czy działa, uruchamiając serwer roboczy Django, a następnie utworzyłeś w projekcie aplikację *reviews*. Po wstępnym zapoznaniu się z projektem Django warto poznać teorię stojącą u podstaw działania Django oraz żądań i odpowiedzi HTTP.

Kup ksi k

Pole ksi k

Paradygmat Model View Template

Aplikacje często tworzy się na podstawie wzorca projektowego *Model View Controller* (*MVC*), zgodnie z którym model aplikacji (jej dane) jest wyświetlany w jednym lub kilku widokach, a kontroler zarządza interakcjami między modelem a widokiem. W Django wykorzystuje się podobny paradygmat zwany *Model View Template* (*MVT*).

Zgodnie z paradygmatem MVT, podobnie jak w MVC, dane przechowuje się za pomocą modeli. Jednak w przypadku MVT widok sprawdza model i wyświetla go w szablonie. Zwykle w przypadku platform opartych na wzorcu MVC wszystkie trzy komponenty muszą być napisane w tym samym języku. W przypadku wzorca MVT szablon może być napisany w innym języku. W platformie Django modele i widoki są napisane w Pythonie, a szablon w języku HTML. Oznacza to, że programista Pythona może pracować nad modelami i widokami, a programista HTML-a może pracować nad kodem w tym języku. Najpierw omówimy szczegóły dotyczące modeli, widoków i szablonów, a następnie przeanalizujemy przykładowe scenariusze, w których wykorzystywane są te komponenty.

Modele

Modele Django definiują dane dla aplikacji i tworzą warstwę abstrakcji dla obsługi dostępu do bazy danych SQL za pomocą *mapowania obiektowo-relacyjnego* (*Object Relational Mapper — ORM*). ORM umożliwia definiowanie schematów danych (klas, pól i relacji między nimi) w kodzie Pythona bez konieczności znajomości działania wykorzystywanej bazy danych. Oznacza to, że w kodzie Pythona można zdefiniować swoją warstwę obsługi baz danych, a Django wygeneruje automatycznie zapytania SQL. Mechanizmy ORM są omówione szczegółowo w rozdziale 2., „Modele i migracje”.

> Skrót *SQL* oznacza *Structured Query Language*. Jest to sposób opisu typu bazy danych, która zapisuje dane w tabelach złożonych z wielu wierszy. Każdą tabelę można porównać do arkusza kalkulacyjnego. Jednak w przeciwieństwie do arkuszy kalkulacyjnych między danymi z każdej tabeli można tworzyć relacje. Danymi zarządza się za pomocą zapytań SQL (zwykle podczas rozważań dotyczących baz danych używa się po prostu określenia „zapytanie”). Za pomocą zapytań można pobierać dane (`SELECT`), dodawać i zmieniać dane (odpowiednio `INSERT` i `UPDATE`), a także usuwać dane (`DELETE`). Istnieje wiele serwerów baz danych SQL, np. SQLite, PostgreSQL, MySQL lub Microsoft SQL Server. Składnia SQL w różnych bazach danych jest podobna, chociaż zdarzają się różnice w poszczególnych dialektach. ORM platformy Django obsługuje te różnice. Najpierw będziemy zapisywać dane na dysku w bazie danych SQLite, ale później, podczas wdrożenia na serwerze zmienimy bazę danych na PostgreSQL. Nie będzie to jednak wymagać żadnych zmian w kodzie.

Zwykle podczas tworzenia zapytań do bazy danych wynik ma postać prymitywnych obiektów Pythona (np. list ciągów tekstowych, liczb całkowitych, zmiennoprzecinkowych lub bajtów).

Kup ksi k   Pole ksi k

W przypadku ORM wyniki są automatycznie przekształcane w instancje zdefiniowanych klas modeli. Dzięki użyciu ORM otrzymujemy automatycznie ochronę przed luką w zabezpieczeniach zwaną wstrzykiwaniem SQL.

Jeśli masz doświadczenie z bazami danych i SQL, możesz też pisać własne zapytania.

Widoki

W widoku Django definiuje się większość logiki aplikacji. Gdy użytkownik odwiedza witrynę, jego przeglądarka internetowa wysyła żądanie, aby pobrać dane z witryny (w następnym punkcie omawiamy szczegółowo żądania HTTP oraz zawarte w nich informacje). Widok jest funkcją, która pobiera żądanie w postaci obiektu Pythona (konkretnie obiektu `HttpRequest` platformy Django). Widok decyduje, jak odpowiedzieć na żądanie i jakie dane odesłać użytkownikowi. Widok musi zwracać obiekt `HttpResponse`, który zawiera wszystkie informacje dla klienta: treść, status HTTP i inne nagłówki.

Widok może też opcjonalnie pobierać informacje z adresu URL żądania, np. identyfikator. Zgodnie z typowym wzorcem projektowym widok wykonuje zapytanie do bazy danych za pośrednictwem mechanizmu ORM Django, korzystając z identyfikatora przekazanego do widoku. Następnie widok może wyrenderować szablon (więcej na ten temat znajduje się nieco dalej), uzupełniając go danymi z modelu pobranego z bazy danych. Wyrenderowany szablon zostanie umieszczony jako treść w obiekcie `HttpResponse` i zwrócony przez funkcję widoku. Django automatycznie prześle dane do przeglądarki.

Szablony

Szablon jest plikiem w formacie *HyperText Markup Language* (*HTML*) (szablonem może być właściwie dowolny plik tekstowy), zawierającym specjalne elementy tymczasowe, które są zastępowane przez zmienne dostarczane przez aplikację. Np. aplikacja może renderować listę elementów w postaci galerii lub tabeli. Widok pobierze te same modele dla obydwu przypadków, ale będzie mógł wyrenderować inne pliki HTML, aby w różny sposób przedstawić dane. Django kładzie nacisk na bezpieczeństwo, dlatego automatycznie stosuje sekwencje ucieczki w zmiennych. Np. w HTML symbole < i > (między innymi) są znakami specjalnymi. Jeśli użyjesz ich w zmiennej, Django automatycznie je zakoduje, aby zostały poprawnie wyświetlone w przeglądarce.

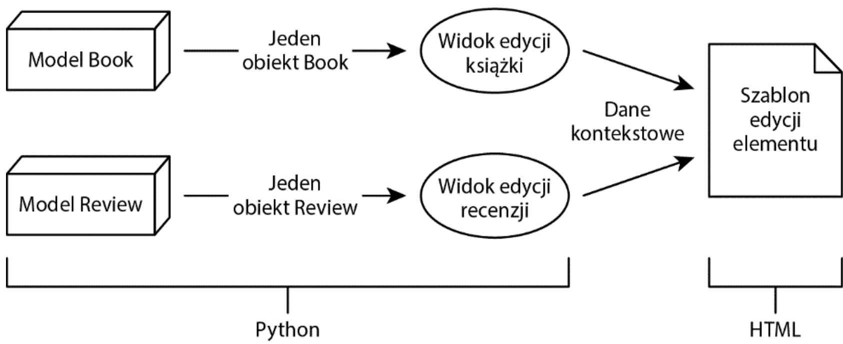
MVT w praktyce

Omówimy teraz kilka przykładów prezentujących działanie wzorca MVT. W tych przykładach korzystamy z modelu `Book`, który zawiera informacje o różnych książkach, oraz z modelu `Review`, który zawiera informacje o różnych recenzjach książek.

W pierwszym przykładzie chcemy uzyskać możliwość edycji danych o książce lub recenzji. Przeanalizujmy scenariusz edycji danych o książce. Potrzebny jest widok służący do pobierania danych z tabeli `Book` z bazy danych i zwrócenia modelu `Book`. Następnie trzeba przekazać kontekst zawierający obiekt `Book` (i inne dane) do szablonu zawierającego formularz służący do podania

Kup ksi k Pole ksi k

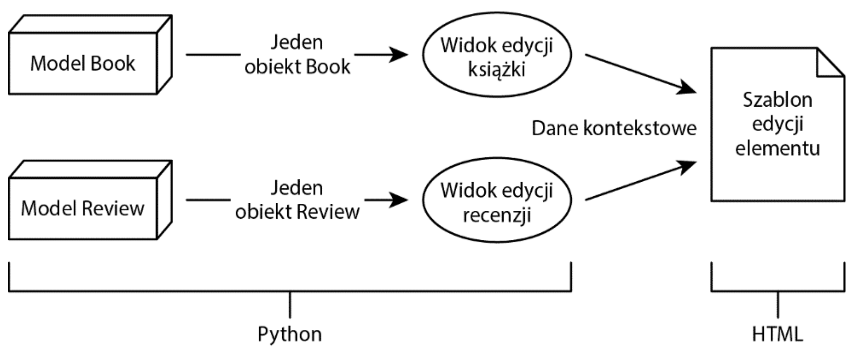
nowych informacji. Drugi scenariusz (edycja recenzji) jest podobny: najpierw trzeba pobrać model `Review` z bazy danych, a następnie przekazać obiekt `Review` i inne dane do szablonu wyświetlającego formularz edycji. Te scenariusze mogą być tak podobne, że w obydwu można ponownie wykorzystać ten sam szablon. Zobacz rysunek 1.3.



Rysunek 1.3. Edycja książki lub recenzji

Jak widać, korzystamy z dwóch modeli, dwóch widoków i jednego szablonu. Każdy widok pobiera jedną instancję powiązanego z nim modelu, ale obydwa mogą korzystać z tego samego szablonu. Jest to generyczna strona HTML wyświetlająca formularz. Widoki mogą zwracać dodatkowe dane kontekstowe. Diagram pokazuje również, które części kodu są napisane w Pythonie, a które w języku HTML.

W drugim przykładzie chcemy pokazać użytkownikowi listę książek lub recenzji zapisanych w aplikacji. Ponadto chcemy umożliwić użytkownikowi wyszukiwanie książek w celu uzyskania listy tytułów spełniających kryteria wyszukiwania. Skorzystamy z tych samych modeli jak w poprzednim przykładzie (`Book` i `Review`), ale utworzymy nowe widoki i szablony. Ponieważ w tym przykładzie można wyróżnić trzy scenariusze, tym razem skorzystamy z trzech widoków: pierwszy służy do pobierania wszystkich widoków, drugi do pobierania wszystkich recenzji, a ostatni do wyszukiwania książek na podstawie pewnych kryteriów. Również w tym przykładzie można skorzystać z jednego szablonu HTML, o ile zostanie on poprawnie zdefiniowany. Zobacz rysunek 1.4.

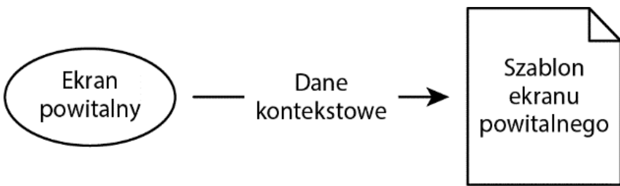


Rysunek 1.4. Wyświetlanie wielu książek lub recenzji

Kup ksi k
Pole ksi k

Modele `Book` i `Review` są takie same jak w poprzednim przykładzie. Trzy widoki pobiorą wiele (zero lub więcej) książek lub recenzji. Następnie każdy widok skorzysta z tego samego szablonu, czyli generycznego pliku HTML, który iteruje podaną listę obiektów i je wyświetla. Również w tym przypadku widoki mogą przesłać w kontekście dodatkowe dane, aby zmienić zachowanie szablonów, ale większa część szablonu będzie dość ogólna.

W platformie Django nie zawsze trzeba użyć modelu do renderowania szablonu HTML. Widok może samodzielnie generować dane kontekstowe i renderować je w szablonie bez użycia modelu danych. Na rysunku 1.5 przedstawiono schemat widoku przesyłającego dane bezpośrednio do szablonu.



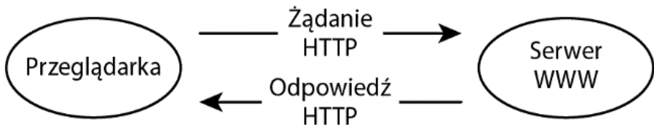
Rysunek 1.5. Przesłanie danych z widoku do szablonu z pominięciem modelu

W tym przykładzie widoczny jest widok powitalny dla użytkowników witryny. Nie zawiera żadnych informacji z bazy danych, dlatego może samodzielnie wygenerować dane kontekstowe. Dane te zależą od typu informacji, jakie należy wyświetlić; można np. przekazać dane o użytkowniku, aby w powitaniu wyświetlić imię zalogowanego użytkownika. Można też wyrenderować szablon bez żadnych danych kontekstowych. Przykładem jest wyświetlany w witrynie plik HTML z danymi statycznymi.

Wstęp do HTTP

Po zapoznaniu się ze wzorcem MVT w Django można przeanalizować, jak Django przetwarza żądanie HTTP i generuje odpowiedź HTTP. Najpierw omówimy szczegółowo, czym są żądania i odpowiedzi HTTP, a także jakie informacje zawierają.

Załóżmy, że ktoś chce odwiedzić Twoją stronę internetową. Wpisuje URL lub klika link do witryny znajdujący się na bieżącej stronie. Przeglądarka internetowa tworzy wtedy żądanie HTTP, które wysyła na serwer hostujący Twoją witrynę. Gdy serwer otrzyma żądanie HTTP z przeglądarki, może je zinterpretować i odesłać odpowiedź (zobacz rysunek 1.6). Odpowiedź odesłana przez serwer może być prosta i zawierać tylko plik HTML lub obraz graficzny pobrany z dysku. Może być też bardziej złożona i powstać w wyniku działania oprogramowania na serwerze (np. Django), które dynamicznie wygeneruje jej zawartość przed odesłaniem.



Rysunek 1.6. Żądanie i odpowiedź HTTP

Kup ksi k Pole ksi k

Żądanie składa się z czterech głównych części, czyli z metody, ścieżki, nagłówków i ciała. Niektóre typy żądań nie zawierają ciała. Jeśli tylko odwiedzasz stronę, przeglądarka nie wyśle ciała. Natomiast jeśli wysyłasz formularz (np. logując się w witrynie lub korzystając z wyszukiwarki), żądanie będzie zawierać ciało z wysłanymi danymi. Przeanalizujmy dwa przykładowe żądania.

Pierwsze żądanie dotyczy przykładowej strony dostępnej pod adresem URL *https://www.example.com/page*. Aby otworzyć tę stronę, przeglądarka najpierw wysyła następujące informacje:

```
GET /page HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:15.0) Firefox/15.0.1
Cookie: sessid=abc123def456
```

Pierwszy wiersz zawiera metodę (`GET`) oraz ścieżkę (`/page`). Widoczna jest też wersja HTTP, w tym przypadku `1.1`, chociaż nie musisz się nią przejmować. Korzystać można z wielu różnych metod HTTP, w zależności od sposobu interakcji ze stroną zdalną. Popularnymi metodami są `GET` (pobieranie zdalnej strony), `POST` (wysyłanie danych do strony zdalnej), `PUT` (tworzenie strony zdalnej) i `DELETE` (usuwanie strony zdalnej). Zauważ, że opisy tych akcji są dość uproszczone — serwer może zdecydować, jak zareaguje na różne metody, a nawet doświadczeni programiści mogą się spierać, którą metodę zaimplementować w przypadku określonej akcji. Warto również zauważyć, że nawet jeśli serwer obsługuje określoną metodę, do jej wykonania często potrzebne są odpowiednie uprawnienia — nie można po prostu użyć metody `DELETE` w przypadku strony internetowej, która się nam nie podoba.

Pisząc aplikację internetową, najczęściej będziesz korzystać z żądań `GET`. Gdy zaczniesz pisać kod obsługujący formularze, musisz skorzystać również z żądań `POST`. Tylko w przypadku zaawansowanych funkcji, takich jak tworzenie API typu REST, będziesz musiał obsłużyć również metody `PUT`, `DELETE` i inne.

Począwszy od drugiego wiersza, w przykładowym żądaniu znajdują się nagłówki. Zawierają one dodatkowe metadane dotyczące żądania. Każdy nagłówek znajduje się w osobnym wierszu, a jego nazwa i wartość są rozdzielone przecinkiem. Większość nagłówków jest opcjonalna (z wyjątkiem nagłówka `Host`, który omówimy nieco dalej). Wielkość liter w nazwach nagłówków ma znaczenie. W tym przykładzie pokazujemy tylko trzy najpopularniejsze nagłówki. Oto ich opis w kolejności występowania:

- `Host`. Jak już wspomniano, jest to jedyny wymagany nagłówek (w przypadku protokołu HTTP w wersji 1.1 lub późniejszych). Na jego podstawie serwer WWW wie, która witryna lub aplikacja powinna odpowiedzieć na żądanie, jeśli jeden serwer hostuje kilka witryn.
- `User-Agent`. Przeglądarka zwykle wysyła na serwer ciąg tekstowy identyfikujący jej wersję oraz system operacyjny. Aplikacja serwera może na tej podstawie wysyłać różne strony na różne urządzenia (np. wersję mobilną strony na smartfony).
- `Cookie`. Prawdopodobnie podczas przeglądania stron internetowych zauważyłeś komunikaty informujące o zapisywaniu cookie w przeglądarce. Cookie to małe elementy danych, które witryna może zapisać w przeglądarce i na ich podstawie identyfikować użytkownika lub zapisywać ustawienia odczytywane podczas kolejnych odwiedzin. Za pomocą tego nagłówka przeglądarka odsyła cookie na serwer.

Kup ksi k Pole ksi k

Istnieje wiele innych standardowych nagłówków, ale ich lista zajęłaby zbyt wiele miejsca. Za ich pomocą można się uwierzytelnić na serwerze (`Authorization`), poinformować serwer o rodzaju akceptowanych danych (`Accept`), a nawet ustawić preferowany język, w jakim powinna być wyświetlona strona (`Accept-Language`, chociaż ten mechanizm zadziała, tylko jeśli twórca strony udostępnia jej treści w żądanym języku). Można nawet zdefiniować własne nagłówki, które będą rozpoznawane tylko przez własną aplikację.

Przeanalizujmy teraz nieco bardziej zaawansowane żądanie, które wysyła pewne informacje na serwer, a zatem (w przeciwieństwie do poprzedniego) zawiera ciało. W tym przykładzie użytkownik loguje się na stronie WWW, wysyłając nazwę użytkownika i hasło. Np. otwiera stronę *https://www.example.com/login* z formularzem, w którym trzeba wpisać nazwę użytkownika i hasło. Po kliknięciu przycisku *Login* na serwer zostanie wysłane następujące żądanie:

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

username=user1&password=password1
```

Jak widać, jest ono podobne do żądania z poprzedniego przykładu, ale są pewne różnice. Używa metody `POST` i zawiera dwa nowe nagłówki (można założyć, że przeglądarka wyśle również inne nagłówki z poprzedniego przykładu):

- `Content-Type`. Ten nagłówek informuje serwer o typie danych zawartych w ciele. W przypadku wartości `application/x-www-form-urlencoded` ciało składa się z par klucz – wartość. Klient HTTP może w tym nagłówku poinformować o wysłaniu danych innego typu, np. w formacie JSON lub XML.
- `Content-Length`. Aby poinformować serwer, ile danych trzeba odczytać, klient musi podać rozmiar wysyłanych danych. Nagłówek `Content-Length` informuje, ile danych zawiera ciało. W tym przykładzie ciało zawiera 32 znaki.

Nagłówki są zawsze oddzielone od ciała pustym wierszem. W tym przykładzie widać sposób zapisania danych w ciele: parametr `username` ma wartość `user1`, a parametr `password` ma wartość `password1`.

Te żądania są dość proste, podobnie jak większość innych. Mogą one zawierać różne metody i nagłówki, ale zwykle mają ten sam format. Po zapoznaniu się z żądaniami przyjrzymy się odpowiedziom HTTP, które są zwracane przez serwery.

Odpowiedź HTTP jest podobna do żądania i zawiera trzy główne części: status, nagłówki i ciało. Podobnie jak w przypadku żądania, w zależności od typu odpowiedzi ciało może być nieobecne. W pierwszym przykładzie widoczna jest prosta odpowiedź zakończona powodzeniem:

```
HTTP/1.1 200 OK
Server: nginx
Content-Length: 18132
Content-Type: text/html
Set-Cookie: sessid=abc123def46

<!DOCTYPE html><html><head>...
```


Kup ksi k Pole ksi k

W pierwszym wierszu znajduje się wersja HTTP, liczbowy kod stanu (200) oraz opis tekstowy kodu (OK — żądanie zakończyło się powodzeniem). Po omówieniu następnego przykładu pokażemy więcej kodów stanu. Wiersze od 2. do 5. zawierają nagłówki, podobnie jak żądanie. Niektóre nagłówki już znasz; poniżej opisujemy je w nowym kontekście:

- `Server`. Jest to przeciwieństwo nagłówka `User-Agent`. W tym przypadku serwer informuje klienta o oprogramowaniu serwera.
- `Content-Length`. Na podstawie tej wartości klient wie, ile danych odczytać z serwera, aby pobrać ciało.
- `Content-Type`. Za pomocą tego nagłówka serwer informuje klienta o typie wysyłanych danych. Klient może następnie ustalić sposób wyświetlenia danych — np. obraz musi być wyświetlony inaczej niż kod HTML.
- `Set-Cookie`. W pierwszym przykładowym żądaniu pokazaliśmy, jak klient wysyła cookie na serwer. Ten nagłówek jest odpowiednikiem nagłówka wysyłanego przez klienta. Serwer wysyła go, aby zainstalować ten obiekt cookie w przeglądarce.

Po nagłówkach znajduje się pusty wiersz, a następnie ciało odpowiedzi. Nie pokazujemy jego całej treści, tylko kilka pierwszych znaków z 18 132 znaków kodu HTML wysłanego przez serwer.

Pokażemy teraz przykład odpowiedzi, która zostanie zwrócona w przypadku, gdy żądana strona nie istnieje:

```
HTTP/1.1 404 Not Found
Server: nginx
Content-Length: 55
Content-Type: text/html

<!DOCTYPE html><html><body>Page Not Found</body></html>
```

Jest on podobny do poprzedniego przykładu, ale status ma teraz postać `404 Not Found`. Jeśli podczas przeglądania internetu zetknąłeś się z błędem 404, wiesz już, że przeglądarka otrzymała odpowiedź tego typu. Różne kody stanu są pogrupowane według typu powodzenia lub niepowodzenia.

- 100 – 199. Serwer wysyła kody z tego zakresu, aby poinformować o zmianach w protokole lub o konieczności przesłania większej ilości danych. Nie musisz się nimi przejmować.
- 200 – 299. Kod stanu z tego zakresu oznacza pomyślną obsługę żądania. Najczęściej występuje kod stanu `200 OK`.
- 300 – 399. Kod stanu z tego zakresu oznacza, że żądana strona została przeniesiona pod inny adres. Takie kody zwracają np. usługi skracania adresów URL, które przekierowują z krótkich adresów URL na pełne. Typowe odpowiedzi to `301 Moved Permanently` lub `302 Found`. Podczas wysyłania przekierowania serwer dodaje także nagłówek `Location` zawierający adres URL, na który należy przekierować żądanie.
- 400 – 499. Kod stanu z tego zakresu oznacza, że nie można obsłużyć żądania, ponieważ wystąpił problem z danymi przesłanymi przez klienta. Ten zakres różni się od błędów wynikających z problemów po stronie serwera (które omawiamy w następnym punkcie). Pokazaliśmy już odpowiedź `404 Not Found`; jest ona zwracana ze względu

Kup ksi k   Pole ksi k

na błędne żądanie, ponieważ klient żąda nieistniejącego dokumentu. Niektóre inne popularne odpowiedzi to `401 Unauthorized` (klient powinien się zalogować) i `403 Forbidden` (klient nie ma uprawnień dostępu do określonego zasobu). Obydwu problemów można uniknąć po zalogowaniu się klienta, a zatem są one uznawane za problemy zaistniałe po stronie klienta (żądanie).

- `500 – 599`. Kod stanu z tego zakresu oznacza błąd po stronie serwera. Klient nie rozwiąże tego problemu, poprawiając żądanie. W przypadku Django najczęściej zwracany jest błąd `500 Internal Server Error`. Zostanie on wygenerowany, jeśli w kodzie wystąpi wyjątek. Innym typowym błędem jest `504 Gateway Timeout`, który może wystąpić, gdy wykonywanie kodu trwa zbyt długo. Inne typowe błędy to `502 Bad Gateway` i `503 Service Unavailable`, które zwykle oznaczają problem z hostingiem aplikacji.

Są to jedynie najpopularniejsze kody stanu HTTP. Pełna lista jest dostępna pod adresem *https://developer.mozilla.org/en-US/docs/Web/HTTP/Status*. Jednak podobnie jak w przypadku nagłówków HTTP kody stanu mogą być przypisywane dość dowolnie, a zatem aplikacja może zwracać niestandardowe wartości. To serwer i klienci decydują, co oznaczają te niestandardowe kody stanu.

Jeśli masz do czynienia z protokołem HTTP po raz pierwszy, zapewne przyznasz, że to dość sporo nowych informacji. Na szczęście Django znacznie ułatwia pracę programistom i umieszcza wszystkie dane przychodzące w obiekcie `HttpRequest`. Najczęściej nie musisz znać większości przychodzących danych, ale zawsze możesz je sprawdzić. To samo dotyczy odpowiedzi, które Django umieszcza w obiekcie `HttpResponse`. Zwykle wystarczy podać dane, które należy zwrócić, ale można też samodzielnie ustawić kody stanu HTTP i nagłówki. W dalszej części rozdziału pokazujemy, jak zarządzać danymi w obiektach `HttpRequest` i `HttpResponse`.

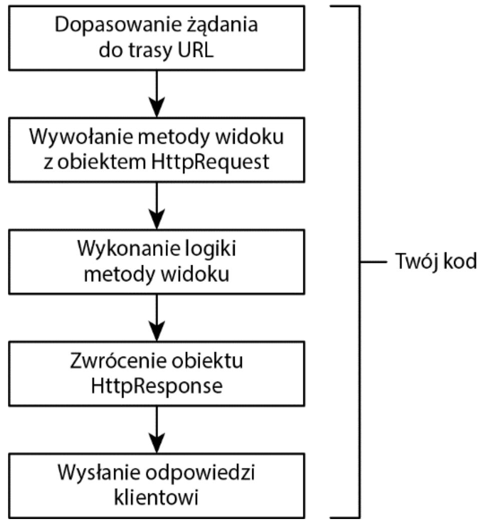
Przetwarzanie żądania

Omówimy teraz podstawowy przepływ żądań i odpowiedzi, aby pokazać poszczególne kroki wykonywane w kodzie na każdym etapie. Podczas tworzenia kodu najpierw należy napisać widok. Widok będzie wykonywał pewne działania, np. wysyłał zapytania do bazy danych. Następnie będzie przesyłał uzyskane dane do innej funkcji renderującej szablon, a na koniec zwróci obiekt `HttpResponse` zawierający dane, które trzeba odesłać klientowi.

Następnie Django musi wiedzieć, jak odwzorować określone adresy URL na widoki, aby wczytać poprawny widok dla adresu URL otrzymanego żądania. Mapowanie to tworzy się w pliku Pythona zawierającym konfigurację adresów URL.

Gdy Django otrzyma żądanie, przetworzy plik konfiguracyjny dotyczący adresów URL, aby znaleźć odpowiedni widok. Wywołuje ten widok, przekazując obiekt `HttpRequest` reprezentujący otrzymane żądanie. Widok zwróci obiekt `HttpResponse`. Django go przetworzy i odeśle zawarte w nim dane na serwer WWW hosta, a następnie do klienta, który wysłał żądanie.

Kup ksi k Pole ksi k

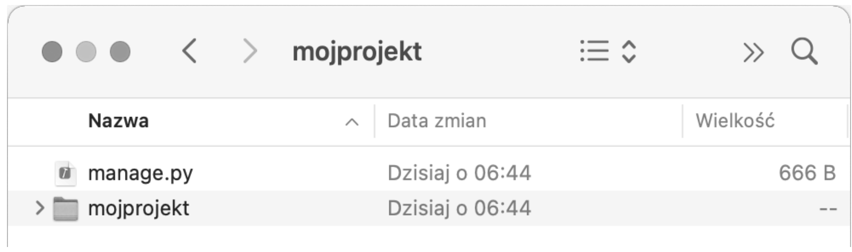


Rysunek 1.7. Przepływ żądania i odpowiedzi

Przepływ danych między żądaniem a odpowiedzią jest przedstawiony na rysunku 1.7; elementy oznaczone etykietą *Twój kod* reprezentują kod pisany przez programistę — pierwszy i ostatni krok obsługuje Django. Django dopasowuje URL, wywołuje kod widoku i przekazuje odpowiedź do klienta.

Projekt Django

W poprzednim punkcie wspomnieliśmy już o projektach Django. Przypomnij sobie, co się dzieje po uruchomieniu polecenia `startproject` (w projekcie *mojprojekt*): polecenie utworzy katalog *mojprojekt* zawierający plik *manage.py* oraz katalog *mojprojekt* (zgodnie z nazwą projektu; w ćwiczeniu 1.1, „Tworzenie projektu, aplikacji oraz serwera roboczego", był to folder *bookr*, o nazwie odpowiadającej nazwie projektu). Hierarchia katalogów jest widoczna na rysunku 1.8. Omówimy teraz szczegółowo plik *manage.py* oraz zawartość pakietu *mojprojekt*.



Rysunek 1.8. Hierarchia katalogów w projekcie mojprojekt

Kup ksi k

Pole ksi k

manage.py

Jak sugeruje nazwa skryptu, służy on do zarządzania projektem Django. Większość poleceń służących do interakcji z projektem przekazuje się w postaci argumentów tego skryptu w wierszu poleceń. Aby np. wykonać polecenie `manage.py runserver`, należy uruchomić skrypt *manage.py* w następujący sposób:

```
python3 manage.py runserver
```

Skrypt *manage.py* umożliwia wykonanie wielu przydatnych poleceń. Są one omówione szczegółowo w tej książce. Oto najpopularniejsze z nich:

- `runserver`. Uruchamia serwer roboczy HTTP platformy Django, obsługujący aplikację Django na komputerze lokalnym.
- `startapp`. Tworzy nową aplikację Django w projekcie. Aplikacje są omówione szczegółowo nieco dalej.
- `shell`. Uruchamia interpreter Pythona z wczytanymi ustawieniami Django. Polecenie to przydaje się podczas interakcji z aplikacją bez konieczności ręcznego wczytywania ustawień Django.
- `dbshell`. Uruchamia interaktywną powłokę połączoną z bazą danych z użyciem domyślnych parametrów z ustawień Django. W ten sposób można samodzielnie wykonywać zapytania SQL.
- `makemigrations`. Generuje instrukcje zmiany w bazie danych na podstawie definicji modelu. Więcej informacji na ten temat znajduje się w rozdziale 2., „Modele i migracje".
- `migrate`. Stosuje migracje wygenerowane poleceniem `makemigrations`. Również z tego polecenia skorzystasz w rozdziale 2., „Modele i migracje".
- `test`. Uruchamia napisane testy automatyczne. Z tego polecenia skorzystasz w rozdziale 14., „Testowanie".

Pełna lista poleceń jest dostępna pod adresem *https://docs.djangoproject.com/en/3.0/ref/django-admin/*.

Katalog mojprojekt

Oprócz pliku *manage.py* polecenie `startproject` utworzyło również katalog *mojprojekt* (zobacz rysunek 1.9). Jest to pakiet Pythona dla tego projektu. Zawiera ustawienia projektu, pliki konfiguracyjne dla serwera WWW i globalne mapowanie adresów URL. W katalogu mojprojekt znajduje się pięć plików:

- *__init__.py*
- *asgi.py*
- *settings.py*
- *urls.py*
- *wsgi.py*

Kup książkę Poleć książkę

mojprojekt

Nazwa	Data zmian	Wielkość
__init__.py	Dzisiaj o 06:44	0 bajtów
asgi.py	Dzisiaj o 06:44	397 B
settings.py	Dzisiaj o 06:44	3 KB
urls.py	Dzisiaj o 06:44	752 B
wsgi.py	Dzisiaj o 06:44	397 B

Rysunek 1.9. Pakiet mojprojekt (znajdujący się w katalogu projektu mojprojekt)

__init__.py

Pusty plik, dzięki któremu wiadomo, że katalog *mojprojekt* jest modułem Pythona. Jeśli wcześniej programowałeś w Pythonie, znasz już tego typu pliki.

settings.py

Ten plik zawiera ustawienia aplikacji Django. Nieco dalej opisujemy jego zawartość.

urls.py

Ten plik zawiera globalne mapowanie adresów URL, na podstawie którego Django będzie początkowo znajdować widoki lub inne potomne mapowania adresów URL. Niebawem umieścisz w tym pliku mapowanie adresów URL.

asgi.py* i *wsgi.py

Za pomocą tych plików serwery WWW typu ASGI lub WSGI komunikują się z aplikacją Django po wdrożeniu na serwer produkcyjny. Zwykle nie ma potrzeby ich edytowania, ponadto nie używa się ich w trakcie codziennej pracy programistycznej. Ich użycie opisujemy w dodatkowym rozdziale 17., „Deployment of a Django Application”.

Serwer roboczy Django

W ćwiczeniu 1.1, „Tworzenie projektu, aplikacji oraz serwera roboczego”, uruchomiłeś już serwer roboczy Django. Jak wcześniej wspomniano, jest to serwer WWW przeznaczony do uruchamiania na komputerze programisty podczas pracy. Nie jest przeznaczony do użycia w środowisku produkcyjnym.

Domyślnie serwer ten nasłuchuje na porcie 8000 hosta `localhost` (127.0.0.1), ale można to zmienić, dodając numer portu lub adres i numer portu po argumencie `runserver`:

```
python3 manage.py runserver 8001
```

Po wykonaniu tego polecenia serwer będzie nasłuchiwał na porcie 8001 hosta `localhost` (127.0.0.1).

Kup ksi k
Pole ksi k

Jeśli komputer hostuje różne adresy, można też skonfigurować nasłuchiwanie na jednym z nich lub na wszystkich za pomocą adresu `0.0.0.0`:

```
python3 manage.py runserver 0.0.0.0:8000
```

W tym przykładzie serwer będzie nasłuchiwał wszystkich adresów komputera na porcie `8000`. Ta technika sprawdzi się, jeśli chcesz przetestować aplikację na innym komputerze lub smartfonie.

Serwer roboczy obserwuje katalog projektu Django i restartuje się automatycznie po każdym zapisaniu dowolnego pliku. Dzięki temu wszystkie zmiany w kodzie zostaną automatycznie wczytane na serwerze. Nadal jednak trzeba samodzielnie odświeżyć przeglądarkę, aby zobaczyć zmiany.

Aby zakończyć działanie polecenia `runserver`, trzeba skorzystać ze standardowej metody kończenia procesów w terminalu, czyli nacisnąć kombinację klawiszy *Ctrl+C*.

Aplikacje Django

Po omówieniu podstaw teoretycznych aplikacji można przejść do szczegółów dotyczących ich przeznaczenia. Katalog aplikacji zawiera wszystkie modele, widoki i szablony (oraz inne elementy), które są potrzebne do działania aplikacji. Projekt Django zawiera co najmniej jedną aplikację (o ile nie zostanie poważnie zmodyfikowany i nie będzie wymagał wielu funkcji Django). Jeśli aplikacja jest dobrze zaprojektowana, można ją usunąć z projektu i przenieść do innego bez modyfikacji. Zwykle aplikacja zawiera modele dotyczące jednej domeny projektowej. Na tej podstawie można ustalić, czy aplikację należy podzielić na wiele aplikacji.

Aplikacja może mieć dowolną nazwę, która musi jednak spełniać warunki nazewnictwa modułów Pythona (czyli składać się tylko z liter, cyfr i podkreślników) i musi być różna od nazw innych plików znajdujących się w katalogu projektu. W dotychczasowym przykładzie w katalogu projektu znajduje się już katalog *mojprojekt* (zawierający plik *settings.py*), a zatem aplikacja nie może mieć nazwy mojprojekt. Jak pokazaliśmy w ćwiczeniu 1.1, „Tworzenie projektu, aplikacji oraz serwera roboczego”, do tworzenia aplikacji służy polecenie `manage.py startapp nazwaaplikacji`. Oto przykład:

```
python3 manage.py startapp mojaaplikacja
```

Polecenie `startapp` tworzy katalog w projekcie o podanej nazwie aplikacji. Tworzy też początkowe pliki aplikacji. W katalogu aplikacji znajduje się kilka plików i folder, co widać na rysunku 1.10.

- ***__init.py__***. Pusty plik oznaczający, że ten katalog jest modułem Pythona.
- ***admin.py***. Django udostępnia wbudowaną witrynę administracyjną, służącą do przeglądania i edycji danych w graficznym interfejsie użytkownika (*Graphical User Interface — GUI*). W tym pliku definiuje się udostępnianie modeli aplikacji w witrynie administracyjnej Django. Więcej informacji na ten temat znajduje się w rozdziale 4., „Wstęp do witryny administracyjnej Django”.
- ***apps.py***. Ten plik zawiera konfigurację metadanych aplikacji. Nie ma potrzeby jego edycji.

Kup ksi k Pole ksi k

mojaaplikacja

Nazwa	Data zmian	Wielkość
__init__.py	Dzisiaj o 19:37	0 bajtów
admin.py	Dzisiaj o 19:37	63 B
apps.py	Dzisiaj o 19:37	158 B
migrations	Dzisiaj o 19:37	--
models.py	Dzisiaj o 19:37	57 B
tests.py	Dzisiaj o 19:37	60 B
views.py	Dzisiaj o 19:37	63 B

Rysunek 1.10. Zawartość katalogu aplikacji myapp

- ***models.py***. W tym pliku definiuje się modele dla aplikacji. Więcej informacji na ten temat znajduje się w rozdziale 2., „Modele i migracje".
- ***migrations***. Django używa plików migracji do automatycznej rejestracji zmian w bazie danych podczas zmian w modelach. Są one generowane przez Django po uruchomieniu polecenia `manage.py makemigrations` i zapisywane w tym katalogu. Nie zostaną wykonane w bazie danych, dopóki nie wykonasz polecenia `manage.py migrate`. Również te pliki są omówione w rozdziale 2., „Modele i migracje".
- ***tests.py***. Aby umożliwić testowanie poprawności działania kodu, Django umożliwia pisanie testów (jednostkowych, funkcjonalnych lub integracyjnych), których szuka w tym pliku. W tej książce napiszemy nieco testów, a metody testowania są omówione w rozdziale 14., „Testowanie".
- ***views.py***. W tym pliku umieszcza się widoki Django (kod odpowiadający żądaniom HTTP). Niebawem utworzysz podstawowy widok, natomiast szczegółowe informacje o widokach znajdują się w rozdziale 3., „Mapowanie URL, widoki i szablony".

Zawartość tych plików przeanalizujemy nieco później, a teraz uruchomimy Django, wykonując drugie ćwiczenie.

Konfiguracja programu PyCharm

W ćwiczeniu 1.1, „Tworzenie projektu, aplikacji oraz serwera roboczego", przekonałeś się, że projekt Bookr został poprawnie skonfigurowany (ponieważ serwer roboczy został z powodzeniem uruchomiony). Możesz zatem zacząć korzystać z programu *PyCharm* do uruchamiania i edycji projektu. PyCharm jest środowiskiem służącym do pisania programów w Pythonie i zawiera takie funkcje jak uzupełnianie kodu, automatyczne formatowanie stylów i wbudowany debuger. Skorzystasz zatem z tego programu do napisania odwzorowań adresów URL, widoków i szablonów. Za jego pomocą będziesz także uruchamiać i zatrzymywać serwer roboczy, dzięki czemu będziesz mógł debugować kod, korzystając z punktów przerwania.

Kup ksi k

Pole ksi k

Ćwiczenie 1.2. Konfiguracja projektu w programie PyCharm

W tym ćwiczeniu otworzysz projekt Bookr w programie PyCharm i skonfigurujesz interpreter projektu, aby PyCharm mógł uruchamiać i debugować projekt.

1. Otwórz PyCharm. Gdy otworzysz program PyCharm po raz pierwszy, zobaczysz ekran *Welcome to PyCharm* (zobacz rysunek 1.11), na którym będziesz mógł wybrać, co chcesz zrobić.

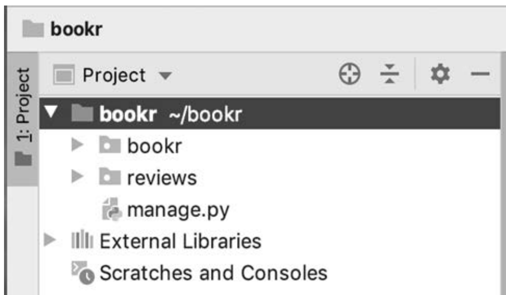
Rysunek 1.11. Ekran powitalny programu PyCharm

2. Kliknij *Open*, a następnie przejdź do utworzonego projektu *bookr* i go otwórz. Upewnij się, że znajdujesz się w katalogu projektu *bookr*, a nie w znajdującym się w nim katalogu pakietu *bookr*.

   Jeśli nie używałeś jeszcze programu PyCharm, zobaczysz pytania o ustawienia i motywy, z których chcesz skorzystać, a gdy na nie odpowiesz, w panelu *Project* z lewej strony okna zobaczysz strukturę projektu *bookr*.

   Panel *Project* powinien wyglądać jak na rysunku 1.12 i zawierać katalogi *bookr* i *reviews* oraz plik *manage.py*. Jeśli zamiast nich widoczne są pliki *asgi.py*, *settings.py*, *urls.py* i *wsgi.py*, oznacza to, że otworzyłeś katalog pakietu *bookr*. W tym przypadku wybierz opcję *File/Open*, a następnie otwórz katalog projektu *bookr*.

   Aby program PyCharm wiedział, jak uruchomić serwer programistyczny Django dla projektu, trzeba skonfigurować interpreter i przypisać do niego plik binarny Pythona w środowisku wirtualnym. W tym celu trzeba najpierw dodać interpreter do globalnych ustawień interpretera.



Rysunek 1.12. Panel Project w programie PyCharm

3. Otwórz *Preferences* (macOS) lub *Settings* (Windows i Linux) w programie PyCharm.

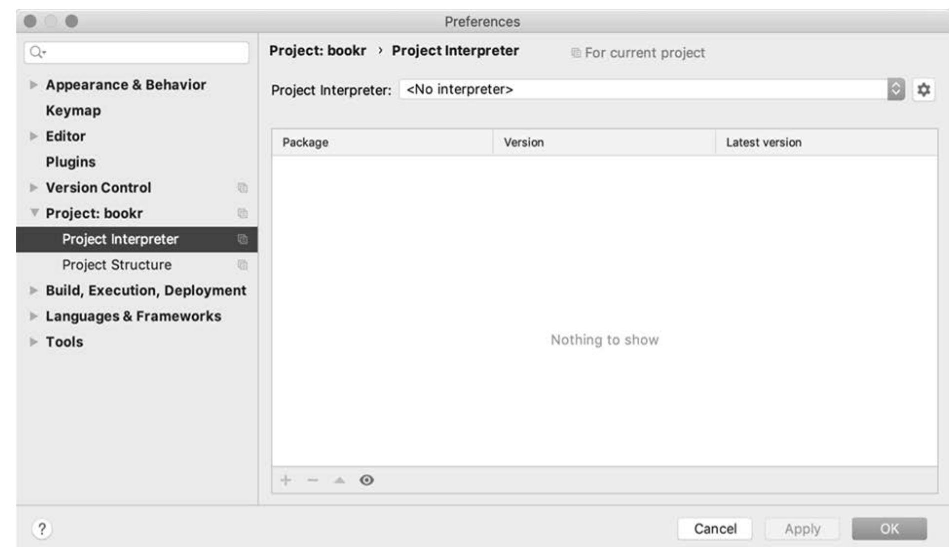
   macOS:

   Menu *PyCharm/Preferences*

   Windows i Linux:

   File/Settings

4. W panelu z listą ustawień z lewej strony wybierz element *Project: bookr*, a następnie kliknij *Project Interpreter* (zobacz rysunek 1.13).



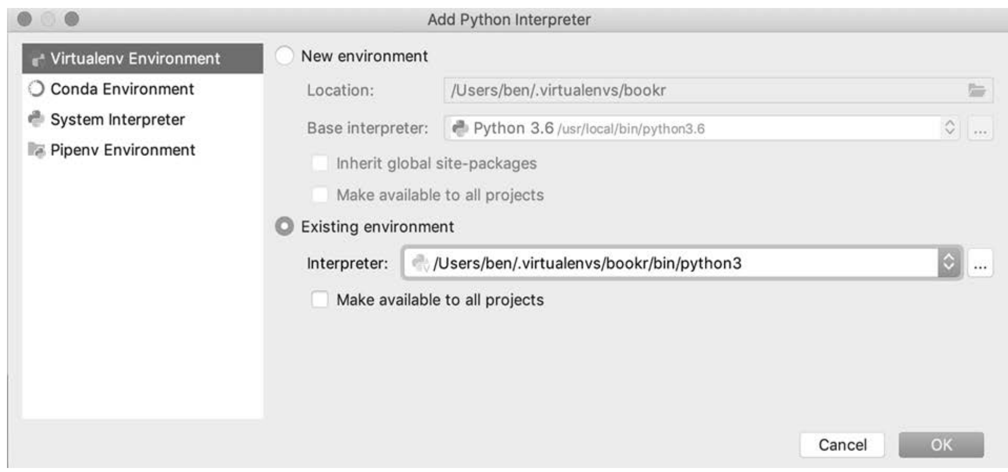
Rysunek 1.13. Ustawienia interpretera projektu

5. Czasem PyCharm może automatycznie znaleźć środowiska wirtualne. W tym przypadku pole *Project Interpreter* może zawierać poprawny interpreter. Jeśli tak i jeśli na liście pakietów znajduje się Django, możesz kliknąć *OK*, aby zamknąć okno i zakończyć to ćwiczenie.

   Jednak w większości przypadków trzeba ręcznie ustawić interpreter Pythona. W tym celu kliknij ikonę zębatki obok listy *Project Interpreter*, a następnie kliknij *Add...*.

Kup ksi k Pole ksi k

6. Na ekranie zostanie wyświetlone okno *Add Python Interpreter*. Zaznacz opcję *Existing environment*, a następnie kliknij wielokropek (...) obok listy *Interpreter*. Znajdź i wybierz interpreter Pythona ze swojego środowiska wirtualnego (zobacz rysunek 1.14).



Rysunek 1.14. Okno Add Python Interpreter

7. W systemie macOS (zakładając, że nazwałeś środowisko wirtualne *bookr*) interpreter zwykle znajduje się w katalogu */Users/<nazwaużytkownika>/.*
   ↪virtualenvs/bookr/bin/python3. Natomiast w systemie Linux interpreter powinien się znajdować w katalogu
   /home/<nazwaużytkownika>/.virtualenvs/bookr/bin/python3.

   Jeśli nie wiesz, gdzie znajduje się interpreter, wykonaj w terminalu polecenie `which python3` w tym samym katalogu, w którym wcześniej wykonałeś polecenie `python manage.py`. W ten sposób powinieneś uzyskać ścieżkę do interpretera Pythona:

   ```
   which python3
   /Users/ben/.virtualenvs/bookr/bin/python3
   ```

   W systemie Windows interpreter znajduje się w katalogu, w którym utworzyłeś środowisko wirtualne poleceniem `virtualenv`.

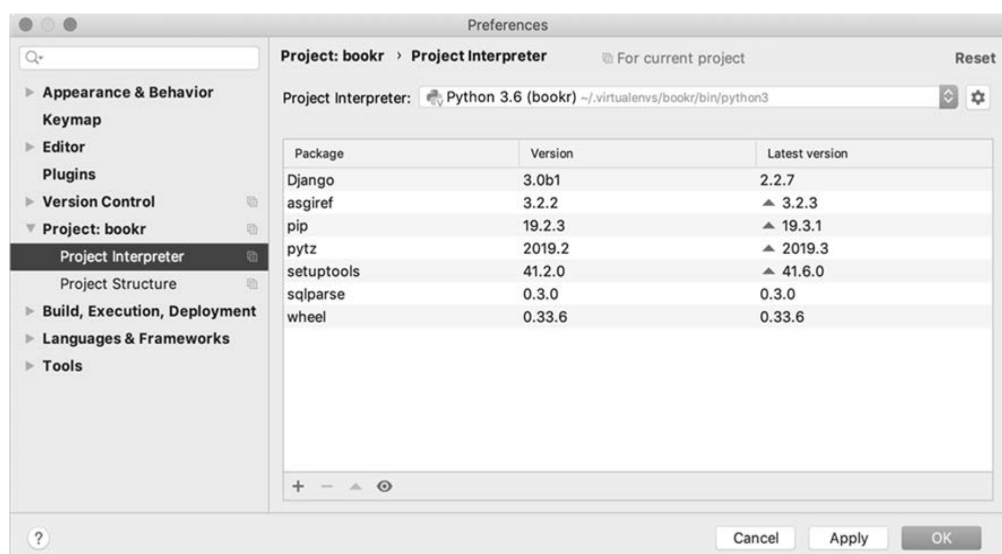
   Po wybraniu interpretera okno *Add Python Interpreter* powinno wyglądać jak na rysunku 1.14.

8. Kliknij *OK*, aby zamknąć okno *Add Python interpreter*.

9. Na ekranie powinno być widoczne główne okno z ustawieniami (zobacz rysunek 1.15), zawierające listę z elementem *Django* (i innymi pakietami ze środowiska wirtualnego).

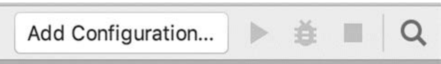
10. Kliknij *OK* w oknie *Preferences*, aby je zamknąć. PyCharm potrzebuje teraz kilku sekund, aby zindeksować zawartość środowiska i zainstalowane w nim biblioteki. Postęp tego procesu można obserwować na pasku stanu w prawym dolnym rogu. Poczekaj na zakończenie procesu i zniknięcie paska postępu.

Kup ksi k Pole ksi k



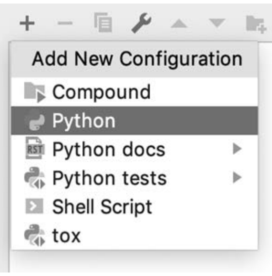
Rysunek 1.15. Widoczne są pakiety ze środowiska wirtualnego

11. Aby uruchomić serwer roboczy Django, trzeba odpowiednio skonfigurować Pythona. Kliknij *Add Configuration…* w prawym górnym rogu okna projektu w programie PyCharm, aby otworzyć okno *Run/Debug Configuration* (zobacz rysunek 1.16).



Rysunek 1.16. Przycisk Add Configuration… w prawym górnym rogu okna PyCharm

12. Kliknij przycisk + w lewym górnym rogu tego okna i wybierz z listy opcję *Python* (zobacz rysunek 1.17).



Rysunek 1.17. Dodawanie nowej konfiguracji Pythona w oknie Run/Debug Configuration

13. Z prawej strony okna zostanie wyświetlony nowy panel zawierający pola służące do konfiguracji sposobu uruchamiania projektu. Wypełnij te pola zgodnie z następującym opisem.

Kup ksi k

Pole ksi k

W polu *Name* można wpisać dowolną nazwę, która jednak powinna być zrozumiała. Wpisz **`Django Dev Server`**.

Script Path jest ścieżką do pliku *manage.py*. Kliknij ikonę foldera w tym polu, po czym znajdź i zaznacz plik *manage.py* znajdujący się w katalogu projektu *bookr*.

Parameters to argumenty, które należy podać za nazwą skryptu *manage.py*. Są to te same argumenty, które podaje się w wierszu poleceń. Użyj argumentu służącego do uruchomienia serwera, czyli **`runserver`**.

> Jak wcześniej wspomniano, polecenie `runserver` może przyjmować argument określający port lub adres, którego powinien nasłuchiwać serwer. Jeśli chcesz, możesz dodać ten argument za argumentem `runserver` w polu *Parameters*.

Pole *Python interpreter* powinno być ustawione automatycznie, zgodnie z opisem w krokach od 5. do 8. Jeśli nie jest ustawione, kliknij strzałkę z prawej strony pola i wybierz interpreter.

W polu *Working directory* powinien się znajdować katalog projektu *bookr*. Prawdopodobnie zostało ono już poprawnie skonfigurowane.

Pola *Add content roots to PYTHONPATH* i *Add source roots to PYTHONPATH* powinny być zaznaczone. Dzięki temu PyCharm doda katalog projektu *bookr* do zmiennej *PYTHONPATH* (listy ścieżek, które interpreter Pythona przeszukuje podczas wczytywania modułu). Jeśli pola te nie zostaną zaznaczone, importowanie komponentów z projektu nie będzie działać.

Upewnij się, że okno *Run/Debug configurations* wygląda podobnie jak na rysunku 1.18, a następnie kliknij *OK*, aby zapisać konfigurację.

14. Teraz zamiast uruchamiać serwer roboczy Django w terminalu, możesz go uruchomić, klikając ikonę strzałki w górnym prawym rogu okna *Project* (zobacz rysunek 1.19).

15. Kliknij ikonę strzałki, aby uruchomić serwer roboczy Django.

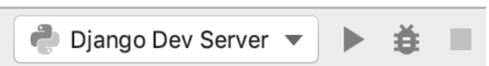
> Upewnij się, że nie są uruchomione żadne inne instancje serwera roboczego Django (np. w terminalu). W przeciwnym razie uruchamiany serwer nie zdoła się przypisać do portu `8000` i jego uruchomienie się nie powiedzie.

16. W dolnej części okna PyCharm pojawi się konsola wyświetlająca wynik informujący o uruchomieniu serwera roboczego (zobacz rysunek 1.20).

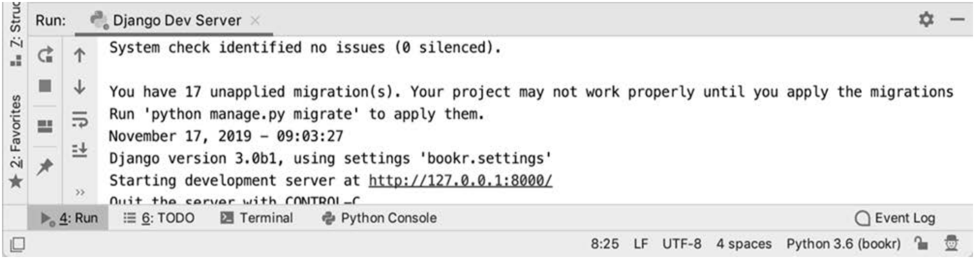
17. Otwórz przeglądarkę i przejdź pod adres *http://127.0.0.1:8000*. Powinieneś zobaczyć ten sam ekran przykładowy Django, który znasz z ćwiczenia 1.1, „Tworzenie projektu, aplikacji oraz serwera roboczego” (rysunek 1.2). Oznacza to, że wszystko jest poprawnie skonfigurowane.

Kup ksi k Pole ksi k

Rysunek 1.18. Ustawienia konfiguracyjne



Rysunek 1.19. Konfiguracja serwera roboczego Django z przyciskami uruchamiania, debugowania i zatrzymywania serwera



Rysunek 1.20. Konsola z uruchomionym serwerem roboczym Django

W tym ćwiczeniu otworzyłeś projekt Bookr w programie PyCharm i skonfigurowałeś interpreter Pythona dla projektu. Następnie dodałeś konfigurację uruchamiania w programie PyCharm, która umożliwia uruchamianie i zatrzymywanie serwera roboczego Django z poziomu tego programu. Później będziemy też debugować projekt w debugerze programu PyCharm.

Szczegółowe informacje o widokach

Możesz już zacząć pisanie swoich widoków Django oraz skonfigurować prowadzące do nich adresy URL. Wiesz już, że widok jest funkcją, która przyjmuje instancję klasy `HttpRequest` (tworzoną przez Django) i (opcjonalnie) pewne parametry z adresu URL. Następnie wykonuje pewne operacje, takie jak pobieranie danych z bazy danych. Na koniec zwraca instancję klasy `HttpResponse`.

W aplikacji Bookr można np. utworzyć widok przyjmujący żądanie dotyczące określonej książki. W tym celu widok powinien wysłać do bazy danych zapytanie o tę książkę, a następnie zwrócić odpowiedź zawierającą stronę HTML przedstawiającą informacje o tej książce. Inny widok może przyjmować żądanie wyświetlenia listy wszystkich książek i zwracać odpowiedź z inną stroną HTML zawierającą żądaną listę. Widoki mogą również tworzyć lub modyfikować dane: inny widok może przyjmować żądanie utworzenia nowej książki; następnie może dodawać książkę do bazy danych i zwracać odpowiedź HTML wyświetlającą dane nowej książki.

W tym rozdziale utworzysz tylko widoki w postaci funkcji, ale Django umożliwia również tworzenie widoków opartych na klasach, dzięki czemu można skorzystać z paradygmatów programowania zorientowanego obiektowo (np. dziedziczenia). Dzięki temu można uprościć kod wykorzystywany w wielu widokach, które mają tę samą logikę biznesową. Można np. pokazać wszystkie książki lub tylko książki z jednego wydawnictwa. Obydwa widoki muszą pobrać listę książek z bazy, a następnie wyrenderować ją w szablonie wyświetlającym listę książek. Jedna klasa widoku może dziedziczyć po innej klasie i zaimplementować tylko różnice w pobieraniu danych. Natomiast pozostałe działanie (np. renderowanie) się nie zmieni. Widoki oparte na klasach mogą mieć więcej możliwości, ale również być trudniejsze w nauce. Zapoznasz się z nimi w rozdziale 11., „Zaawansowane aspekty szablonów i widoki oparte na klasach", gdy będziesz już mieć większe doświadczenie z Django.

Instancja `HttpRequest` przekazywana do widoku zawiera wszystkie dane związane z żądaniem i ma następujące atrybuty:

- `method`. Ciąg reprezentujący metodę HTTP, której użyła przeglądarka w żądaniu o przesłanie strony; zwykle jest to metoda `GET`, ale jeśli użytkownik prześle formularz, będzie to metoda `POST`. Na podstawie metody można zmienić logikę widoku. Np. w przypadku metody `GET` można pokazać pusty formularz, a w przypadku metody `POST` dokonać walidacji danych formularza i je przetworzyć.
- `GET`. Instancja `QueryDict` zawierająca parametry znajdujące się ciągu zapytania URL. Jest to fragment adresu URL znajdujący się za znakiem `?`. Klasę `QueryDict` omawiamy szczegółowo nieco dalej. Zauważ, że ten atrybut jest zawsze dostępny, nawet jeśli żądanie zostało wysłane inną metodą niż `GET`.

Kup ksi k Pole ksi k

- `POST`. Inna instancja `QueryDict` zawierająca parametry przesłane do widoku w żądaniu `POST`, np. podczas przesyłania formularza. Zwykle używa się jej z formularzami Django, które są omówione w rozdziale 6., „Formularze".
- `headers`. Jest to słownik z kluczami, w których nie uwzględnia się wielkości liter, zawierający nagłówki HTTP z żądania. Przykładowo: w zależności od nagłówka `User-Agent` ustawionego przez różne przeglądarki można zwracać odpowiedzi z różną zawartością. We wcześniejszej części tego rozdziału omówiliśmy niektóre nagłówki HTTP, wysyłane przez klienta.
- `path`. To jest ścieżka używana w żądaniu. Zwykle nie musisz jej sprawdzać, ponieważ Django automatycznie ją przetworzy i przekaże do funkcji widoku w postaci parametru, ale w niektórych przypadkach pole to jest przydatne.

Nie będziemy jeszcze używać tych wszystkich atrybutów, a w dalszej części książki omówimy inne, ale warto już teraz wiedzieć, jaką rolę odgrywa argument `HttpRequest` w widoku.

Mapowanie adresów URL

W punkcie „Przetwarzanie żądania" wspomnieliśmy już o mapowaniu adresów URL. Django nie wie automatycznie, którą funkcję widoku należy wykonać po otrzymaniu żądania dla określonego adresu URL. Za pomocą mapowania URL można powiązać ze sobą URL i widok. Np. w projekcie Bookr można odwzorować URL `/books/` na utworzony przez nas widok `books_list`.

Mapowanie ścieżek URL na widoki definiuje się w pliku *urls.py*, który został utworzony automatycznie przez Django w katalogu pakietu *bookr* (chociaż można użyć innego pliku, który należy ustawić w pliku *settings.py*; więcej na ten temat już niebawem).

W tym pliku zdefiniowana jest zmienna `urlpatterns` zawierająca listę ścieżek, które Django sprawdza po kolei, aż znajdzie dopasowanie do żądanego ciągu URL. Na tej podstawie zwraca funkcję widoku lub inny plik *urls.py* również zawierający zmienną `urlpatterns`, która będzie przetwarzana tak samo. Pliki zawierające mapowania URL można łączyć ze sobą w dowolnie długie łańcuchy. Można w ten sposób podzielić mapowania URL na wiele plików (np. jeden lub kilka na aplikację), dzięki czemu można uniknąć tworzenia zbyt dużych plików. Po znalezieniu widoku Django wywołuje instancję `HttpRequest` z wszystkimi parametrami pobranymi z adresu URL.

Reguły ustawia się, wywołując funkcję `path`, która przyjmuje w pierwszym argumencie ścieżkę ciągu URL. Ścieżka może zawierać nazwane parametry, które zostaną przekazane do widoku jako parametry funkcji. Drugim argumentem jest widok lub inny plik zawierający zmienną `urlpatterns`.

Istnieje również funkcja `re_path`, która jest podobna do funkcji `path`, ale w pierwszym argumencie przyjmuje wyrażenie regularne, które umożliwia wykonanie bardziej zaawansowanej konfiguracji. Mechanizm mapowania URL jest bardziej skomplikowany i jest opisany szczegółowo w rozdziale 3., „Mapowanie URL, widoki i szablony".

Kup ksi k

Pole ksi k

Rysunek 1.21 pokazuje domyślny plik *urls.py* generowany przez Django. Widać zmienną `urlpatterns`, która zawiera listę wszystkich skonfigurowanych ciągów URL. W tym przykładzie skonfigurowana jest tylko jedna reguła, która odwzorowuje każdą ścieżkę zaczynającą się od ciągu `admin/` na mapowanie URL w module administracyjnym (o nazwie `admin.site.urls`). Nie jest to mapowanie na widok; jest to przykład połączenia plików z mapowaniem ciągów URL — moduł `admin.site.urls` definiuje pozostałe części ścieżki (znajdujące się za elementem `admin/`), które są odwzorowane na widoki administracyjne. Witrynę administracyjną Django omawiamy w rozdziale 4., „Wstęp do witryny administracyjnej Django".

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

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

Rysunek 1.21. Domyślny plik urls.py

Teraz napiszesz widok i skonfigurujesz mapowanie URL, aby przećwiczyć opisane koncepcje.

Ćwiczenie 1.3. Pisanie widoku i odwzorowania URL

Pierwszy widok będzie bardzo prosty i będzie zawierał tylko tekst statyczny. W tym ćwiczeniu dowiesz się, jak napisać widok i skonfigurować mapę URL, aby odwzorować URL na widok.

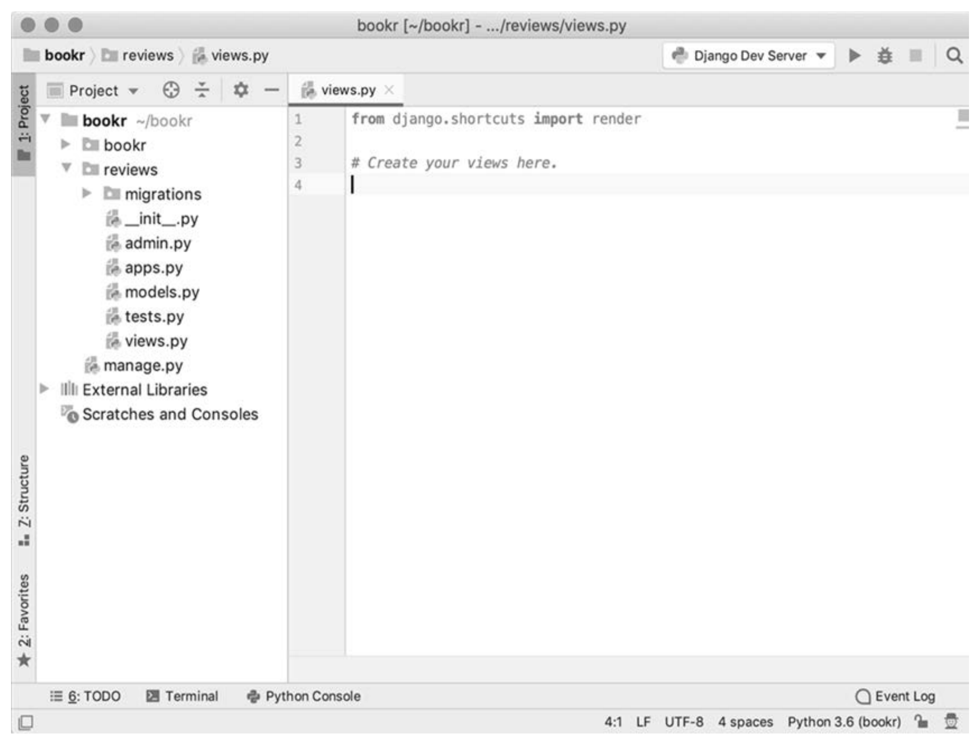
> Podczas wprowadzania zmian w plikach projektu i zapisywania ich zauważysz, że serwer roboczy Django będzie się automatycznie restartował w terminalu lub w konsoli, w której jest uruchomiony. Jest to normalne zachowanie; serwer automatycznie się restartuje, aby wczytać wszystkie zmiany w kodzie. Zauważ również, że zmiany w modelach lub migracjach nie spowodują automatycznego uaktualnienia bazy danych — więcej informacji na ten temat znajduje się w rozdziale 2., „Modele i migracje".

1. W programie PyCharm rozwiń folder *reviews* w podglądzie projektu z lewej strony, a następnie kliknij dwukrotnie znajdujący się w nim plik *views.py*, aby go otworzyć. W panelu z prawej strony (edytor) programu PyCharm powinieneś zauważyć tymczasowy tekst wygenerowany automatycznie przez Django:

   ```
   from django.shortcuts import render
   # Create your views here.
   ```

   Edytor powinien wyglądać jak na rysunku 1.22.

Kup ksi k Pole ksi k



Rysunek 1.22. Domyślna zawartość pliku views.py

2. Usuń ten tekst tymczasowy z pliku *views.py* i wpisz następujący kod:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Witaj, świecie!")
```

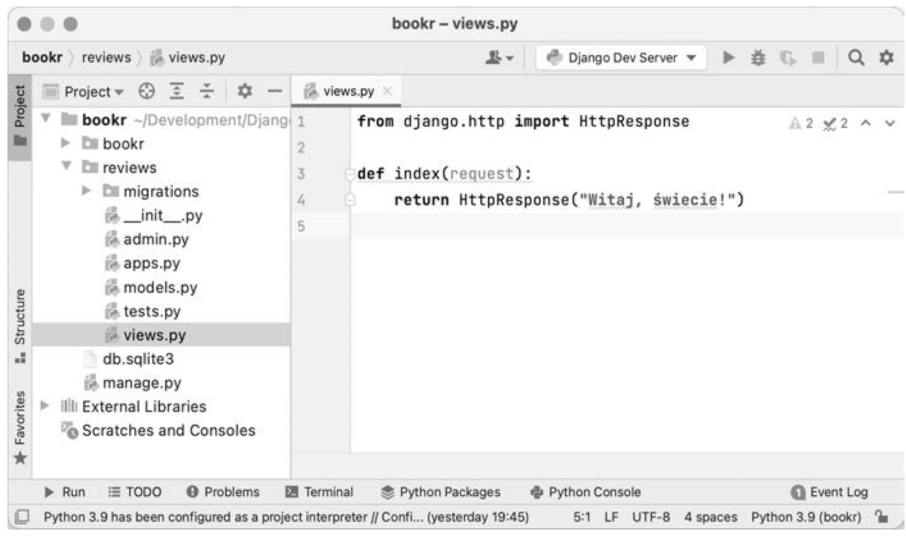
Najpierw trzeba zaimportować klasę `HttpResponse` z pakietu `django.http`. Ta klasa służy do utworzenia odpowiedzi, którą trzeba zwrócić do przeglądarki. Można za jej pomocą kontrolować także nagłówki HTTP i kody stanu. W tym przykładzie używamy domyślnych nagłówków i kodu stanu `200 Success`. Pierwszym argumentem jest treść tekstowa, która zostanie wysłana w ciele odpowiedzi.

Następnie funkcja widoku zwraca instancję `HttpResponse` ze zdefiniowaną przez nas treścią (`Witaj, świecie!`, zobacz rysunek 1.23).

1. Teraz można skonfigurować odwzorowanie URL na widok `index`. Jest to bardzo proste i nie wymaga podawania żadnych parametrów. Rozwiń katalog *bookr* w panelu *Project*, a następnie otwórz plik *urls.py*, który został wygenerowany automatycznie przez Django.

   W tym przykładzie dodamy tylko prosty URL, aby zastąpić domyślne odwzorowanie utworzone przez Django.

Kup ksi k   Pole ksi k



Rysunek 1.23. Zawartość pliku views.py po edycji

2. Zaimportuj swoje widoki do pliku *urls.py*, dodając następujący wiersz za istniejącymi instrukcjami importu.

   ```
   import reviews.views
   ```

3. Dodaj mapę do widoku `index` do listy `urlpatterns`. W tym celu wywołaj funkcję `path` z pustym ciągiem i odwołaniem do funkcji `index`:

   ```
   urlpatterns = [path('admin/', admin.site.urls),\
                  path('', reviews.views.index)]
   ```

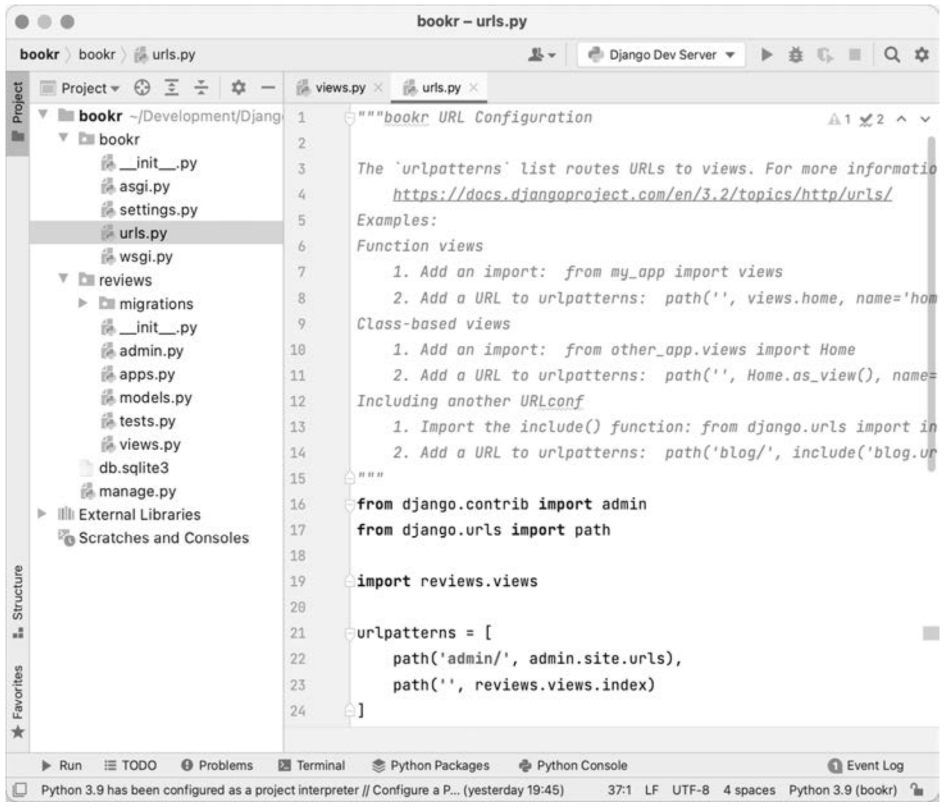
> W ostatnim fragmencie kodu używamy lewego ukośnika (\ ), aby podzielić logikę na kilka wierszy. Podczas wykonywania kodu Python zignoruje ten ukośnik i potraktuje kod w następnym wierszu jako kontynuację bieżącego wiersza.

   Uważaj, aby nie dodawać nawiasów po funkcji `index` (powinna być zapisana w postaci `reviews.views.index`, a nie `reviews.views.index()`), ponieważ trzeba przekazać odwołanie do funkcji, a nie ją wywołać. Gdy skończysz, plik *urls.py* powinien wyglądać jak na rysunku 1.24.

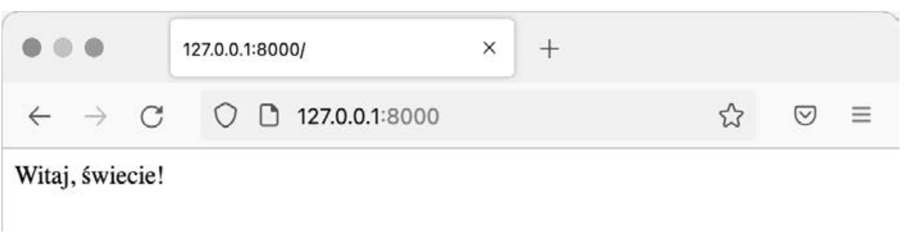
4. Wróć do przeglądarki internetowej i odśwież ją. Domyślny ekran powitalny Django powinien zostać zastąpiony tekstem `Witaj, świecie!`, zdefiniowanym w widoku (zobacz rysunek 1.25).

Dowiedziałeś się, jak napisać funkcję widoku i zmapować na nią URL. Następnie przetestowałeś widok, wczytując go w przeglądarce.

Kup ksi k Pole ksi k



Rysunek 1.24. Plik urls.py po edycji



Rysunek 1.25. Przeglądarka internetowa powinna wyświetlać tekst Witaj, świecie!

GET, POST i obiekty QueryDict

Dane można przesyłać w żądaniu HTTP w postaci parametrów ciągu URL lub w ciele żądania `POST`. Być może przeglądając internet, zauważyłeś już parametry w adresach URL — mają one postać tekstu znajdującego się za znakiem ? — na przykład *http://www.example.com/?parametr1=wartość1 ↪¶metr2=wartość2*. Wcześniej w tym rozdziale widziałeś przykład danych z formularza znajdujących się w żądaniu POST, przesyłanych w celu zalogowania użytkownika (ciało żądania zawierało tekst *username=user1&password=password1*).

Django automatycznie przekształca te parametry w obiekty QueryDict, które będą dostępne w obiekcie HttpRequest przekazywanym do widoku — w atrybutach HttpRequest.GET i HttpRequest.POST, odpowiednio dla parametrów z adresu URL i z ciała. Obiekty QueryDict zachowują się podobnie jak słowniki, ale do jednego klucza można przypisać kilka wartości.

Aby zaprezentować różne sposoby dostępu do elementów, użyjemy przykładowej prostej instancji obiektu QueryDict o nazwie qd i zawierającej tylko jeden klucz (k). Element k zawiera listę z trzema wartościami. Są to litery a, b i c. Przedstawione dalej fragmenty kodu pokazują wyniki uzyskane w interpreterze Pythona.

Najpierw utwórz obiekt QueryDict o nazwie qd, przekazując do konstruktora parametr w postaci ciągu:

```
>>> qd = QueryDict("k=a&k=b&k=c")
```

Jeśli pobierzesz elementy, korzystając z nawiasów lub metody get, uzyskasz ostatnią wartość z listy przypisanej do klucza:

```
>>> qd["k"]
'c'
>>> qd.get("k")
'c'
```

Aby pobrać wszystkie wartości klucza, użyj metody getlist:

```
>>> qd.getlist("k")
['a', 'b', 'c']
```

getlist zawsze zwraca listę; jeśli klucz nie istnieje, lista jest pusta:

```
>>> qd.getlist("błędny klucz")
[]
```

Chociaż metoda getlist nie wygeneruje wyjątku, jeśli klucz nie istnieje, to próba pobrania wartości nieistniejącego klucza za pomocą nawiasów kwadratowych wywoła błąd KeyError, jak w przypadku zwykłego słownika. Unikniesz tego błędu, jeśli użyjesz metody get.

Obiekty QueryDict dla metod GET i POST są niemutowalne (nie można ich zmienić), a zatem skorzystaj z metody copy, aby uzyskać mutowalną kopię umożliwiającą zmianę ich wartości:

```
>>> qd["k"] = "d"
AttributeError: This QueryDict instance is immutable
>>> qd2 = qd.copy()
>>> qd2
<QueryDict: {'k': ['a', 'b', 'c']}>
>>>  qd2["k"] = "d"
>>>  qd2["k"]
"d"
```

Aby zrozumieć, jak powstaje obiekt QueryDict na podstawie adresu URL, przeanalizuj przykładowy URL: *http://127.0.0.1:8000?val1=a&val2=b&val2=c&val3*.

Kup ksi k Pole ksi k

Django za kulisami przekazuje zapytanie z ciągu URL (część znajdującą się za znakiem ?) do konstruktora obiektu `QueryDict` i dołącza go do instancji `request`, którą przekazuje do funkcji widoku. Oto przykład:

```
request.GET = QueryDict("val1=a&val2=b&val2=c&val3")
```

Pamiętaj, że to przypisanie odbywa się, zanim instancja `request` zostanie przekazana do funkcji widoku; nie musisz tego robić samodzielnie.

Parametry przykładowego adresu URL można odczytać w funkcji widoku w następujący sposób:

```
request.GET["val1"]
```

Korzystając ze standardowego dostępu do słownika, mogłeś pobrać wartość `a`.

```
request.GET["val2"]
```

Klucz `val2` zawiera dwie wartości, a zatem w ten sposób pobrałeś ostatnią wartość, czyli `c`.

```
request.GET.getlist("val2")
```

W ten sposób pobrałeś wszystkie wartości zmiennej `val2`: `["b", "c"]`:

```
request.GET["val3"]
```

Ten klucz znajduje się w ciągu zapytania, ale nie ma ustawionej wartości, a zatem poprzednia instrukcja zwróci pusty ciąg.

```
request.GET["val4"]
```

Ten klucz nie jest ustawiony, a zatem zostanie wygenerowany błąd `KeyError`. Dlatego użyj instrukcji `request.GET.get("val4")`, która zwróci wartość `None`.

```
request.GET.getlist("val4")
```

Ponieważ ten klucz nie jest ustawiony, zostanie zwrócona pusta lista (`[]`).

Przyjrzyjmy się teraz użyciu obiektu `QueryDict` z parametrami `GET`. Parametry `POST` przeanalizujesz szczegółowo w rozdziale 6., „Formularze”.

Ćwiczenie 1.4. Sprawdzanie wartości GET i korzystanie z obiektu QueryDict

Wprowadzisz teraz pewne zmiany w widoku `index` z poprzedniego ćwiczenia, aby odczytać wartości z atrybutu `GET` ciągu URL. Następnie przekażesz inne parametry i sprawdzisz wyniki.

1. Otwórz plik *views.py* w programie PyCharm. Dodaj nową zmienną o nazwie `name`, do której wczytasz nazwę użytkownika z parametrów `GET`. Dodaj ten wiersz za definicją funkcji `index`.

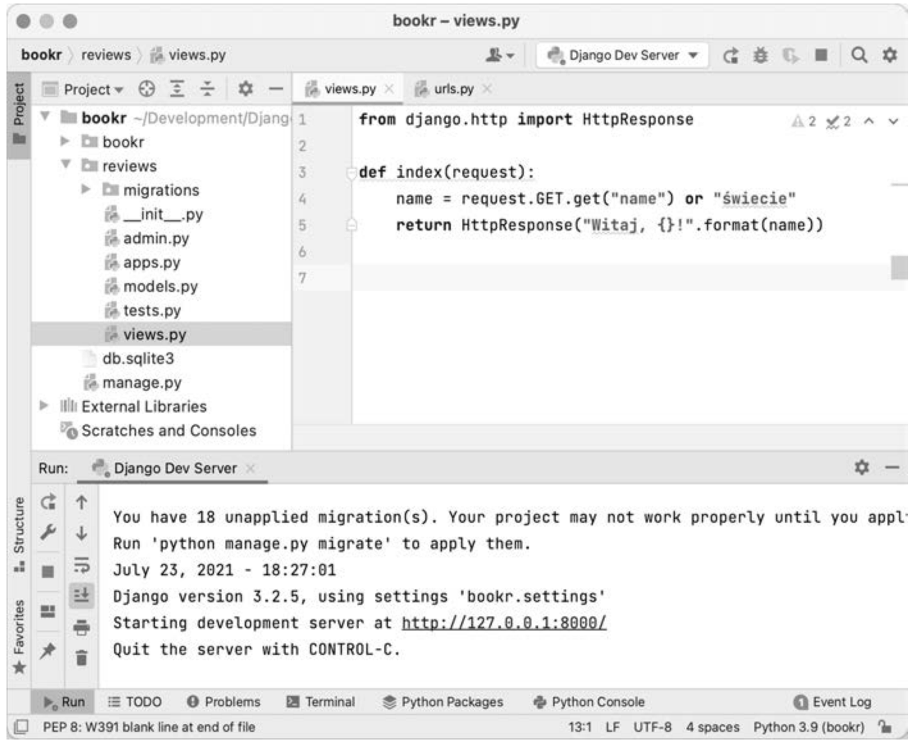
   ```
   name = request.GET.get("name") or "świecie"
   ```


Kup ksi k   Pole ksi k

2. Użyj zmiennej `name` w zwracanej treści:

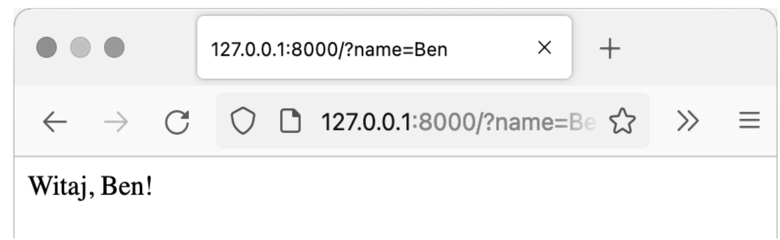
```
return HttpResponse("Witaj, {}!".format(name))
```

Zmieniony kod będzie wyglądał w programie PyCharm tak jak na rysunku 1.26.



Rysunek 1.26. Uaktualniony plik views.py

3. Otwórz stronę *http://127.0.0.1:8000* w przeglądarce. Zauważysz, że nadal zawiera napis *Witaj, świecie!*, ponieważ nie podałeś jeszcze wartości parametru `name` (zobacz rysunek 1.27). Wpisz swoje imię w adresie URL, np. **`http://127.0.0.1:8000?name=Ben`**.

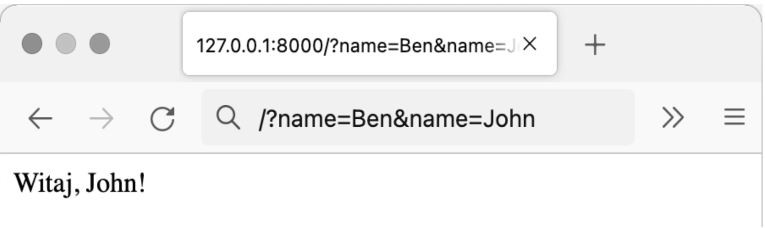


Rysunek 1.27. Ustawianie parametru name w ciągu URL

Kup ksi k

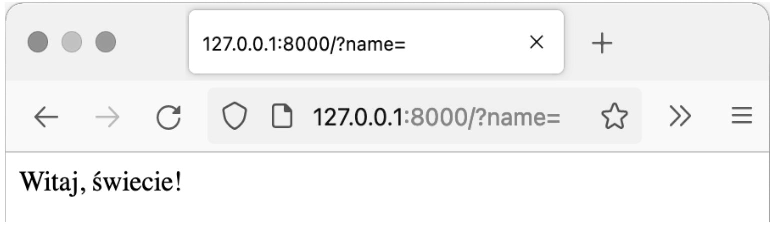
Pole ksi k

4. Spróbuj dodać dwa parametry `name`, np. **`http://127.0.0.1:8000?name=Ben&name=John`**. Jak już wspomniano, funkcja `get` pobiera wartość ostatniego parametru, a zatem na ekranie powinien się pojawić napis *Witaj, John!* (zobacz rysunek 1.28).



Rysunek 1.28. Ustawianie wielu parametrów name w ciągu URL

5. Spróbuj pominąć wartość parametru `name`: **`http://127.0.0.1:8000?name=`**. Na stronie powinien się ponownie pojawić napis *Witaj, świecie!* (zobacz rysunek 1.29).



Rysunek 1.29. Brak wartości parametru name w ciągu URL

> Być może zastanawiasz się, dlaczego ustawiamy domyślną wartość zmiennej `name` na słowo `świecie`, korzystając ze składni `or`, a nie przekazujemy wartości `'świecie'` jako domyślnej wartości do funkcji `get`. Przypomnij sobie, co się stało w kroku 5., w którym przekazałeś pustą wartość do parametru `name`. Gdybyś przekazał do metody `get` domyślną wartość `'świecie'`, funkcja ta zwróciłaby pusty ciąg, ponieważ parametr `name` ma ustawioną pustą wartość. Pamiętaj o tym, tworząc widoki, ponieważ istnieje różnica między brakiem wartości a ustawioną pustą wartością. W niektórych przypadkach można się zdecydować na przekazywanie domyślnej wartości do metody `get`.

W tym ćwiczeniu pobrałeś wartości z ciągu URL w widoku, korzystając z atrybutu `GET` przychodzącego żądania. Dowiedziałeś się, jak ustawić wartości domyślne i która wartość zostanie pobrana, jeśli do tego samego parametru przypisano wiele wartości.

Analiza ustawień Django

Nie opisaliśmy jeszcze, jak Django przechowuje ustawienia. Ponieważ poznałeś już różne elementy Django, możesz przeanalizować plik *settings.py*. Zawiera on wiele ustawień, za pomocą których można dostosować Django do swoich potrzeb. Domyślny plik *settings.py* został utworzony wraz z nowym projektem Bookr.

Kup ksi k   Pole ksi k

Opiszemy teraz kilka najważniejszych ustawień dostępnych w tym pliku oraz kilka innych, które mogą się przydać po zdobyciu pewnego doświadczenia z platformą Django. Otwórz plik *settings.py* w programie PyCharm i przeanalizuj go, czytając dalszą część podrozdziału, aby się dowiedzieć, jakie wartości ustawić dla swojego projektu oraz gdzie się one znajdują.

Wszystkie ustawienia zdefiniowane w tym pliku są zmiennymi globalnymi dostępnymi w całym pliku. Kolejność, w jakiej je omawiamy, jest zgodna z kolejnością, w jakiej są zdefiniowane w pliku, chociaż pomijamy niektóre z nich — np. między ustawieniami `DEBUG` i `INSTALLED_APPS` znajduje się opcja `ALLOWED_HOSTS`, której nie omawiamy w tej części książki (jej opis znajduje się w dodatkowym rozdziale 17., „Deployment of a Django Application (Part 1 — Server Setup)”).

```
SECRET_KEY = '...'
```

To jest automatycznie wygenerowana wartość, której nie należy nikomu udostępniać. Służy do obliczania wartości skrótów, generowania tokenów i jest wykorzystywana w innych funkcjach kryptograficznych. Jeśli w cookie zapisana jest istniejąca sesja, a wartość tej opcji ulegnie zmianie, sesja zostanie unieważniona.

```
DEBUG = True
```

Gdy ta opcja ma wartość `True`, Django automatycznie wyświetli wyjątki w przeglądarce, aby umożliwić debugowanie wszystkich napotkanych problemów. Przed wdrożeniem aplikacji w środowisku produkcyjnym trzeba ustawić wartość opcji na `False`.

```
INSTALLED_APPS = [...]
```

Jeśli piszesz swoją aplikację Django (np. aplikację `reviews`) lub instalujesz zewnętrzne aplikacje (co opisujemy w rozdziale 15., „Zewnętrzne biblioteki Django”), musisz je umieścić na tej liście. Jak mogłeś się już przekonać, nie trzeba ich koniecznie dodawać w tym miejscu (widok `index` działał, mimo że nie dodałeś aplikacji `reviews` do tej listy). Jeśli jednak Django ma automatycznie znajdować szablony aplikacji, pliki statyczne, pliki migracji i inne pliki konfiguracyjne, aplikacje trzeba dodać do tej listy.

```
ROOT_URLCONF = 'bookr.urls'
```

To jest moduł Pythona, który platforma Django wczyta, aby znaleźć obsługiwane ciągi URL. Zauważ, że jest to plik, w którym wcześniej umieściłeś mapę adresów URL widoku `index`.

```
TEMPLATES = [...]
```

Obecnie nie musisz rozumieć tego ustawienia, ponieważ nie będziesz go zmieniał. Istotny jest natomiast następujący wiersz:

```
'APP_DIRS': True,
```

Ta opcja informuje Django, że podczas wczytywania szablonów należy ich szukać w katalogu *templates* każdej aplikacji uwzględnionej na liście `INSTALLED_ APP`. Aplikacja `reviews` nie ma jeszcze katalogu *templates*, ale dodasz go w następnym ćwiczeniu.

Django udostępnia większą liczbę ustawień, które nie znajdują się w pliku *settings.py*, a zatem w tych przypadkach skorzysta z wbudowanych ustawień domyślnych. W tym pliku można też definiować własne ustawienia potrzebne w aplikacji. Można tu również umieszczać ustawienia

Kup ksi k Pole ksi k

zewnętrznych aplikacji. W następnych rozdziałach dodasz w tym pliku ustawienia innych aplikacji. Lista wszystkich ustawień z wartościami domyślnymi jest dostępna pod adresem *https://docs.djangoproject.com/en/3.0/ref/settings/*.

Korzystanie z ustawień w kodzie

Czasem trzeba się odwołać w kodzie do ustawień z pliku *settings.py*. Dotyczy to zarówno ustawień wbudowanych w Django, jak i zdefiniowanych przez programistę. Niektórzy w tym celu będą chcieli użyć następującego kodu:

```
from bookr import settings

if settings.DEBUG: # sprawdzenie, czy aplikacja działa w trybie DEBUG
do_some_logging()
```

> Symbol # w poprzednim przykładzie oznacza komentarz. Komentarze umieszcza się w kodzie, aby opisać działanie pewnych fragmentów logiki.

Z kilku powodów jest to błędne podejście:

- Istnieje możliwość uruchomienia Django z innym plikiem ustawień. W tym przypadku przykładowy kod spowoduje błąd, ponieważ podany plik nie zostanie znaleziony. Natomiast jeśli plik istnieje, zostanie zaimportowany, ale może zawierać błędne ustawienia.
- Django ma ustawienia, które nie zawsze są uwzględnione w pliku *settings.py*. W tej sytuacji Django użyje wbudowanych wartości domyślnych. Jeśli ktoś usunie wiersz `DEBUG = True` z pliku *settings.py*, Django skorzysta z domyślnej wartości opcji `DEBUG` (czyli `False`). Natomiast jeśli spróbujesz uzyskać do niej bezpośredni dostęp za pomocą składni `settings.DEBUG`, zostanie wygenerowany błąd.
- Inne wykorzystywane biblioteki mogą zmienić definicję ustawień, w wyniku czego plik *settings.py* może zawierać zupełnie inną zawartość. Zachowanie wszystkich aplikacji wykracza poza zakres tej książki, ale warto pamiętać, że mają one wpływ na ten plik.

Zalecany sposób polega na użyciu modułu `django.conf`.

```
from django.conf import settings # zaimportuj ustawienia z tego modułu

if settings.DEBUG:
do_some_logging()
```

Podczas importowania komponentu `settings` z modułu `django.conf` Django rozwiązuje trzy wspomniane wcześniej problemy:

- Wczytuje ustawienia z tego pliku ustawień, który został skonfigurowany w platformie Django.
- Interpoluje wszystkie domyślne wartości ustawień.
- Przetwarza wszystkie ustawienia zdefiniowane przez zewnętrzne biblioteki.

Kup ksi k Pole ksi k

Nawet jeśli plik *settings.py* w ostatnim przykładzie nie zawiera ustawienia `DEBUG`, dostępna będzie wartość domyślna, ustawiona wewnętrznie w platformie Django (czyli `False`). To samo dotyczy wszystkich pozostałych ustawień zdefiniowanych w Django; jeśli jednak zdefiniujesz w tym pliku swoje ustawienia, Django nie ustawi dla nich domyślnych wartości, a zatem w kodzie trzeba uwzględnić sytuację, gdy ustawienia te nie będą dostępne — zachowanie aplikacji w tej sytuacji zależy od programisty i wykracza poza tematykę tej książki.

Znajdowanie szablonów HTML w katalogach aplikacji

Sposób znajdowania szablonów przez Django zależy od wielu opcji. Można np. ustawić wartość zmiennej `TEMPLATES` w pliku *settings.py*, ale najłatwiej (na razie) będzie utworzyć katalog *templates* w katalogu *reviews*. Django będzie szukać szablonów w tym katalogu (i w katalogach *templates* innych aplikacji), ponieważ opcja `APP_DIRS` ma wartość `True` w pliku *settings.py*, o czym przekonałeś się w poprzednim punkcie.

Ćwiczenie 1.5. Tworzenie katalogu templates oraz szablonu bazowego

W tym ćwiczeniu utworzysz katalog `templates` dla aplikacji `reviews`. Następnie dodasz plik szablonu HTML, który platforma Django będzie mogła wyrenderować w odpowiedzi HTTP.

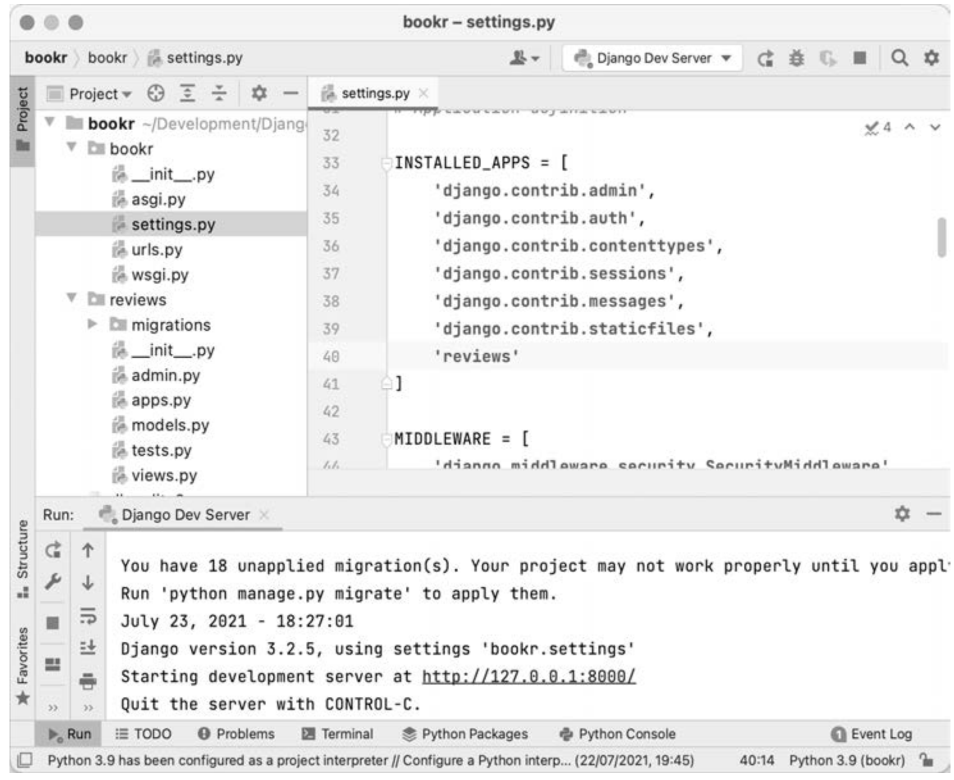
1. Plik *settings.py* oraz opcję `INSTALLED_APPS` omówiliśmy w poprzednim punkcie („Analiza ustawień Django”). Musisz dodać aplikację `reviews` do listy `INSTALLED_APPS`, aby platforma Django mogła znaleźć potrzebne szablony. Otwórz plik *settings.py* w programie PyCharm. Edytuj opcję `INSTALLED_APPS` i dodaj na końcu listy aplikację `reviews`. Opcja ta powinna mieć następującą wartość:

```
INSTALLED_APPS = ['django.contrib.admin',\
                  'django.contrib.auth',\
                  'django.contrib.contenttypes',\
                  'django.contrib.sessions',\
                  'django.contrib.messages',\
                  'django.contrib.staticfiles',\
                  'reviews']
```

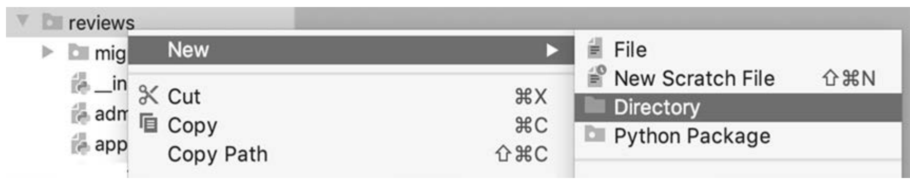
W programie PyCharm plik ten powinien wyglądać jak na rysunku 1.30.

1. Zapisz i zamknij plik *settings.py*.
2. W przeglądarce projektu w programie PyCharm kliknij prawym przyciskiem myszy katalog *reviews* i wybierz opcję *New/Directory* (zobacz rysunek 1.31).
3. Wpisz nazwę **`templates`** i kliknij *OK*, aby utworzyć katalog (zobacz rysunek 1.32).
4. Kliknij prawym przyciskiem myszy nowy katalog *templates* i wybierz polecenie *New/HTML File* (zobacz rysunek 1.33).

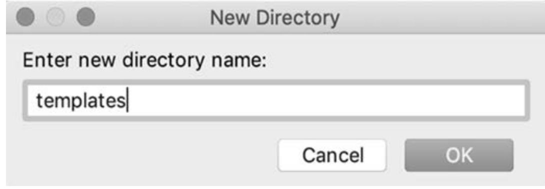
Kup ksi k Pole ksi k



Rysunek 1.30. Aplikacja reviews dodana do pliku settings.py



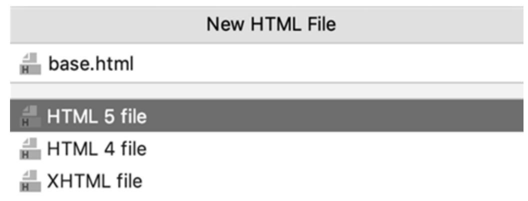
Rysunek 1.31. Tworzenie nowego katalogu w katalogu reviews



Rysunek 1.32. Nadaj katalogowi nazwę templates

Rysunek 1.33. Tworzenie nowego pliku HTML w katalogu templates

5. W oknie, które się pojawi, wpisz nazwę **`base.html`**, nie usuwaj zaznaczenia opcji *HTML 5 file* i naciśnij *Enter*, aby utworzyć plik (zobacz rysunek 1.34).



Rysunek 1.34. Okno New HTML File

6. PyCharm automatycznie otworzy nowy plik, który będzie miał następującą zawartość:

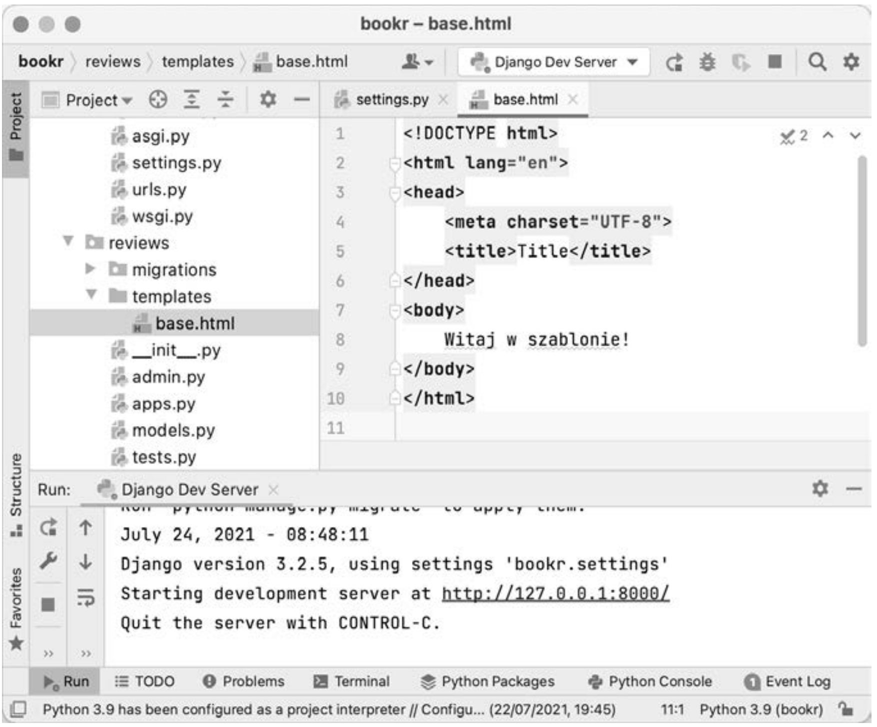
```
<!DOCTYPE html> <html lang="en">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>

</body>
</html>
```

7. Umieść krótką informację między znacznikami `<body>...</body>`, aby sprawdzić, czy szablon zostanie wyrenderowany:

```
<body>
Witaj w szablonie!
</body>
```

Wygląd szablonu w programie PyCharm pokazuje rysunek 1.35.

W tym ćwiczeniu utworzyłeś katalog *templates* dla aplikacji `reviews` i dodałeś do niego szablon HTML. Szablon ten zostanie wyrenderowany po zaimplementowaniu funkcji `render` w widoku.



Rysunek 1.35. Szablon base.html z przykładowym tekstem

Renderowanie szablonu za pomocą funkcji render

Utworzyłeś już szablon, ale musisz jeszcze uaktualnić widok `index`, aby wyrenderował szablon, a nie tekst `Witaj (name)!`, który jest obecnie wyświetlany na ekranie (jak na rysunku 1.29). W tym celu trzeba skorzystać z funkcji `render` i przekazać do niej nazwę szablonu. Funkcja `render` zwraca instancję obiektu `HttpResponse`. Szablon można renderować innymi metodami, dzięki którym można mieć większą kontrolę nad tym procesem, ale wspomniana funkcja spełnia nasze potrzeby. Funkcja `render` przyjmuje co najmniej dwa argumenty: pierwszym jest zawsze żądanie przekazane do widoku, a drugim nazwa lub ścieżka względna renderowanego szablonu. W trzecim argumencie przekażemy też kontekst renderowania, zawierający wszystkie zmienne, które będą dostępne w szablonie — więcej informacji na ten temat znajduje się w ćwiczeniu 1.7, „Użycie zmiennych w szablonach".

Ćwiczenie 1.6. Renderowanie szablonu w widoku

W tym ćwiczeniu uaktualnisz funkcję `index`, aby renderowała szablon HTML utworzony w ćwiczeniu 1.5, „Tworzenie katalogu templates oraz szablonu bazowego". Skorzystasz z funkcji `render`, która wczytuje szablon z dysku i wysyła go do przeglądarki. W ten sposób zastąpisz statyczny tekst zwracany obecnie przez funkcję widoku `index`.

Kup ksi k Pole ksi k

1. W programie PyCharm otwórz plik *views.py* znajdujący się w katalogu *reviews*.
2. Nie musisz już samodzielnie tworzyć instancji `HttpResponse`, dlatego usuń wiersz importu tej klasy:

   ```
   from django.http import HttpResponse
   ```

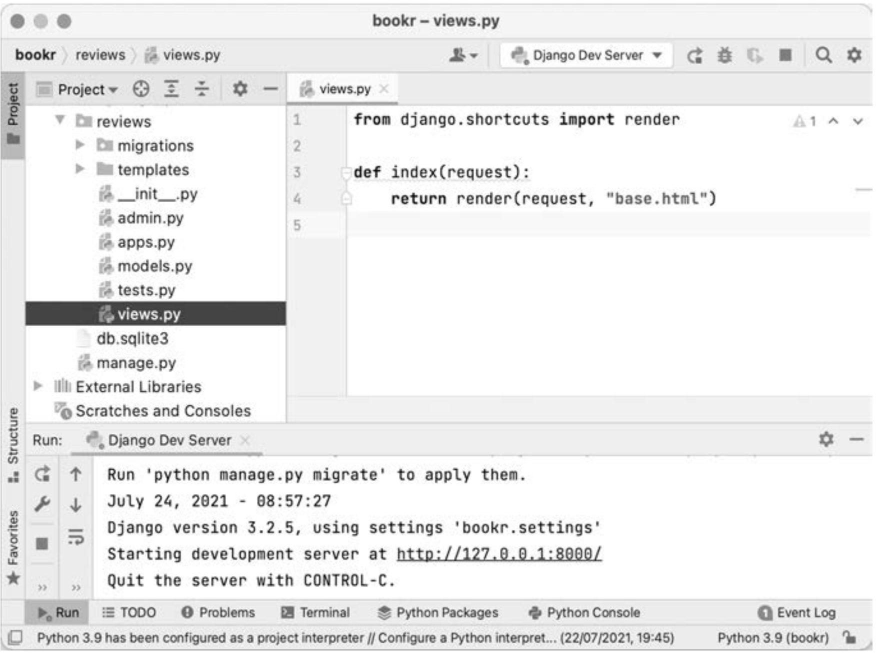
3. Zastąp go instrukcją importującą funkcję `render` z modułu `django.shortcuts`:

   ```
   from django.shortcuts import render
   ```

4. Uaktualnij funkcję `index`, aby zamiast zwracać `HttpResponse`, wywoływała funkcję `render`, przekazując do niej instancję `request` i nazwę szablonu:

   ```
   def index(request):
       return render(request, "base.html")
   ```

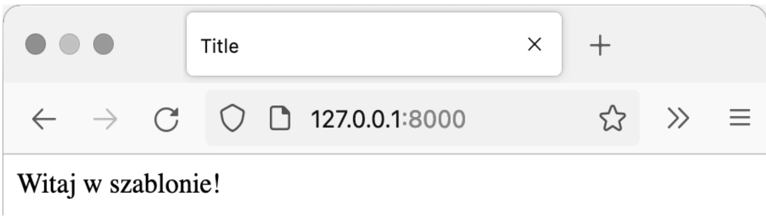
Zawartość pliku widoczna w programie PyCharm wygląda tak jak na rysunku 1.36.



Rysunek 1.36. Gotowy plik views.py

5. Jeśli to konieczne, uruchom serwer roboczy. Następnie otwórz przeglądarkę i odśwież stronę *http://127.0.0.1:8000*. Na ekranie powinien zostać wyświetlony napis *Witaj w szablonie!*, jak na rysunku 1.37.

Kup ksi k Pole ksi k



Rysunek 1.37. Pierwszy wyrenderowany szablon HTML

Renderowanie zmiennych w szablonach

Szablony nie są jedynie statycznymi plikami HTML. Zwykle zawierają zmienne, które są interpolowane podczas renderowania. Większość zmiennych przekazuje do szablonu widok za pośrednictwem kontekstu, czyli słownika (lub obiektu o charakterze słownika) zawierającego nazwy wszystkich zmiennych, z których może skorzystać szablon. Ponownie posłużymy się przykładem projektu Bookr. Bez użycia zmiennych w szablonie dla każdej książki, którą trzeba wyświetlić, potrzebny byłby osobny plik HTML. Aby tego uniknąć, można zdefiniować w szablonie zmienną `book_name`, którą widok przekaże do szablonu po przypisaniu do niej tytułu wczytanego modelu książki. Aby wyświetlić informacje o innej książce, nie trzeba zmieniać kodu HTML; widok przekaże do szablonu dane innej książki. Teraz powinieneś już rozumieć, na czym polega współpraca modelu, widoku i szablonu.

Inaczej niż w niektórych innych językach, takich jak PHP, zmienne trzeba przekazać do szablonu jawnie, a zmienne zdefiniowane w widoku nie są automatycznie dostępne w szablonie. Jest to podyktowane kwestiami bezpieczeństwa, a także koniecznością uniknięcia przypadkowego zaśmiecania przestrzeni nazw szablonu (chcemy uniknąć obecności nieoczekiwanych zmiennych w szablonie).

Wewnątrz szablonu zmienne są oznaczane podwójnymi nawiasami klamrowymi `{{ }}`. Chociaż nie jest to standardowa konwencja, ten styl jest dość popularny i wykorzystywany w innych systemach szablonów, np. Vue.js i Mustache. Symfony (platforma PHP) również wykorzystuje podwójne nawiasy klamrowe w języku szablonów Twig, a zatem możliwe, że znasz już tę składnię.

Aby wyrenderować zmienną w szablonie, umieść ją w nawiasach klamrowych: `{{ book_name }}`. Django automatycznie zastosuje sekwencje ucieczki dla wyrenderowanego kodu HTML, a zatem w zmiennej można uwzględnić znaki specjalne (np. < lub >) bez obaw o poprawny wynik. Jeśli zmienna nie zostanie przekazana do szablonu, Django nie wyrenderuje w tym miejscu niczego i nie wygeneruje błędu.

Zmienną można renderować na wiele innych sposobów, korzystając z filtrów, które są omówione w rozdziale 3., „Mapowanie URL, widoki i szablony”.

Kup ksi k Pole ksi k

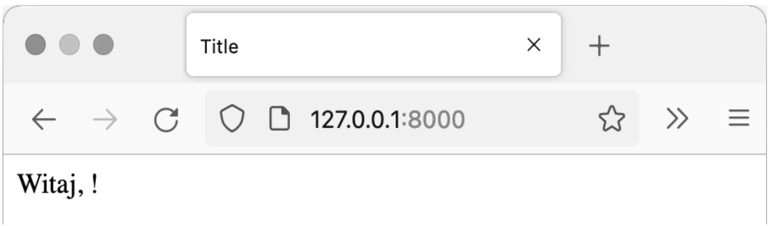
Ćwiczenie 1.7. Użycie zmiennych w szablonach

Umieścisz teraz prostą zmienną w pliku *base.html*, aby sprawdzić, jak działa interpolacja zmiennych w Django.

1. Otwórz plik *base.html* w programie PyCharm.
2. Zmodyfikuj element `<body>`, dodając do niego miejsce na wyrenderowanie zmiennej `name`:

```
<body>
Witaj, {{ name }}!
</body>
```

3. Powróć do przeglądarki i odśwież jej zawartość (upewniwszy się, że wyświetlasz stronę *http://127.0.0.1:8000*). Zauważysz, że na stronie widnieje teraz napis `Witaj, !`. Brakuje w nim imienia, ponieważ nie ustawiłeś zmiennej `name` w kontekście renderowania (zobacz rysunek 1.38).



Rysunek 1.38. Szablon nie wyrenderował żadnej wartości, ponieważ nie skonfigurowałeś kontekstu

4. Otwórz plik *views.py* i w funkcji `index` dodaj zmienną o nazwie `name`, a następnie przypisz do niej `wartość "świecie"`:

```
def index(request):
    name = "świecie"
    return render(request, "base.html")
```

5. Ponownie odśwież stronę w przeglądarce. Nic się nie powinno zmienić, ponieważ wszystko, co chcemy wyrenderować, trzeba jawnie przekazać do funkcji `render` w postaci kontekstu. Jest to słownik zmiennych, które są dostępne podczas renderowania.
6. Dodaj słownik kontekstu w trzecim argumencie funkcji `render`. Zmień wiersz z wywołaniem funkcji `render` na następujący:

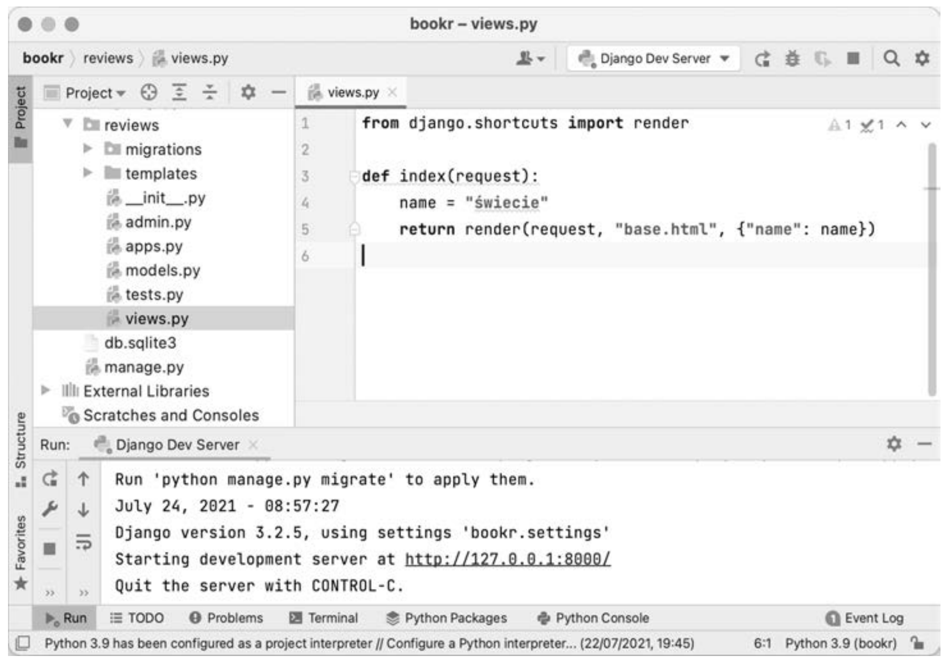
```
return render(request, "base.html", {"name": name})
```

   Podgląd kodu w programie PyCharm wygląd tak jak na rysunku 1.39.

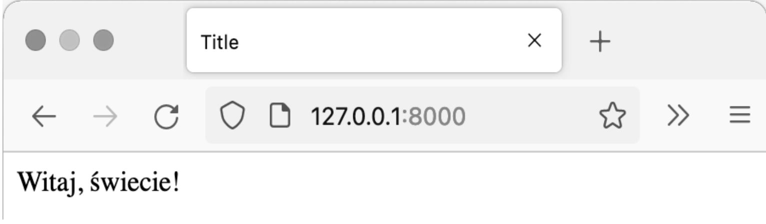
7. Ponownie odśwież stronę w przeglądarce. Tym razem powinna wyświetlać tekst *Witaj, świecie!* (zobacz rysunek 1.40).

W tym ćwiczeniu połączyłeś szablon utworzony w poprzednim ćwiczeniu z funkcją `render`, aby wyrenderować stronę HTML ze zmienną `name` przekazaną do szablonu w słowniku kontekstu.

Kup ksi k Pole ksi k



Rysunek 1.39. views.py ze zmienną name przekazaną w postaci kontekstu renderowania



Rysunek 1.40. Szablon z wyrenderowaną zmienną

Debugowanie i obsługa błędów

O ile nie jesteś doskonałym programistą, który nigdy nie popełnia błędów, prawdopodobnie kiedyś będziesz musiał obsłużyć błędy lub debugować kod. Jeśli w programie pojawi się błąd, zwykle można go wychwycić na dwa sposoby: kod wygeneruje wyjątek albo uzyskasz nieoczekiwane dane wyjściowe lub zaobserwujesz nieoczekiwany efekt podczas przeglądania strony. Prawdopodobnie częściej będą się pojawiać wyjątki w kodzie, ponieważ mogą być generowane w wielu sytuacjach. Jeśli kod generuje nieoczekiwane wyniki, ale nie generuje żadnych wyjątków, prawdopodobnie warto skorzystać z debugera programu PyCharm, aby sprawdzić przyczynę problemu.

Kup ksi k Pole ksi k

Wyjątki

Jeśli pisałeś programy w Pythonie lub w innych językach, prawdopodobnie znasz już pojęcie wyjątku. Jeśli nie, poznasz je już za chwilę. Wyjątki są generowane (lub rzucane — w terminologii innych języków) w przypadku wystąpienia błędów. Wykonywanie programu zostaje w tym miejscu kodu wstrzymane, a wyjątek jest przesyłany przez łańcuch wywoływania funkcji, aż zostanie przechwycony. Jeśli nie zostanie przechwycony, program ulegnie awarii, czasem zwracając komunikat błędu opisujący wyjątek oraz miejsce, w którym został wygenerowany. Niektóre wyjątki są generowane przez Pythona. Również kod napisany przez programistę może generować wyjątki, aby szybko zatrzymać wykonywanie programu w określonym miejscu. Oto niektóre typowe wyjątki, z którymi możesz się zetknąć, programując w Pythonie:

- `IndentationError`

  Python zgłasza ten wyjątek, jeśli kod zawiera błędne wcięcia lub zawiera wcięcia utworzone zarówno za pomocą spacji, jak i znaków tabulacji.

- `SyntaxError`

  Python zgłasza ten wyjątek, jeśli kod ma błędną składnię. Oto przykład:

```
>>> a === 1
  File "<stdin>", line 1
    a === 1
        ^
SyntaxError: invalid syntax
```

- `ImportError`

  Ten wyjątek zostanie zgłoszony, gdy nie powiedzie się instrukcja importu, np. jeśli chcesz zaimportować komponent z nieistniejącego pliku lub taki, który nie jest zdefiniowany w pliku:

```
>>> import missing_file
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named missing_file
```

- `NameError`

  Ten wyjątek zostanie zgłoszony podczas próby dostępu do zmiennej, która nie została jeszcze zdefiniowana:

```
>>> a = b + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

- `KeyError`

  Ten wyjątek zostanie zgłoszony podczas próby dostępu do klucza, który nie został ustawiony w słowniku (lub w obiekcie o charakterze słownika):

```
>>> d = {'a': 1}
>>> d['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
```


Kup ksi k Pole ksi k

- `IndexError`

  Ten wyjątek zostanie zgłoszony podczas próby dostępu do indeksu wykraczającego poza długość listy:

  ```
  >>> l = ['a', 'b']
  >>> l[3]
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  IndexError: list index out of range
  ```

- `TypeError`

  Ten wyjątek zostanie zgłoszony podczas próby wykonania operacji na obiekcie, który jej nie obsługuje, lub podczas użycia dwóch obiektów innego typu — np. podczas próby dodania ciągu tekstowego do liczby całkowitej:

  ```
  >>> 1 + '1'
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: unsupported operand type(s) for +: 'int' and 'str'
  ```

Django zgłasza również własne wyjątki, z którymi zapoznasz się, czytając tę książkę.

Jeśli uruchomisz serwer roboczy Django z ustawieniem `DEBUG = True` zdefiniowanym w pliku *settings.py*, Django automatycznie przechwyci wyjątki występujące w kodzie (zapobiegając awarii programu). Następnie wygeneruje odpowiedź HTTP, prezentującą zrzut stosu i inne informacje ułatwiające debugowanie problemu. Gdy serwer jest uruchomiony w trybie produkcyjnym, opcja `DEBUG` powinna mieć wartość `False`. W tej sytuacji Django zwróci standardową stronę informującą o wewnętrznym błędzie serwera, pozbawioną wszystkich wrażliwych danych. Istnieje również możliwość wyświetlenia niestandardowej strony błędu.

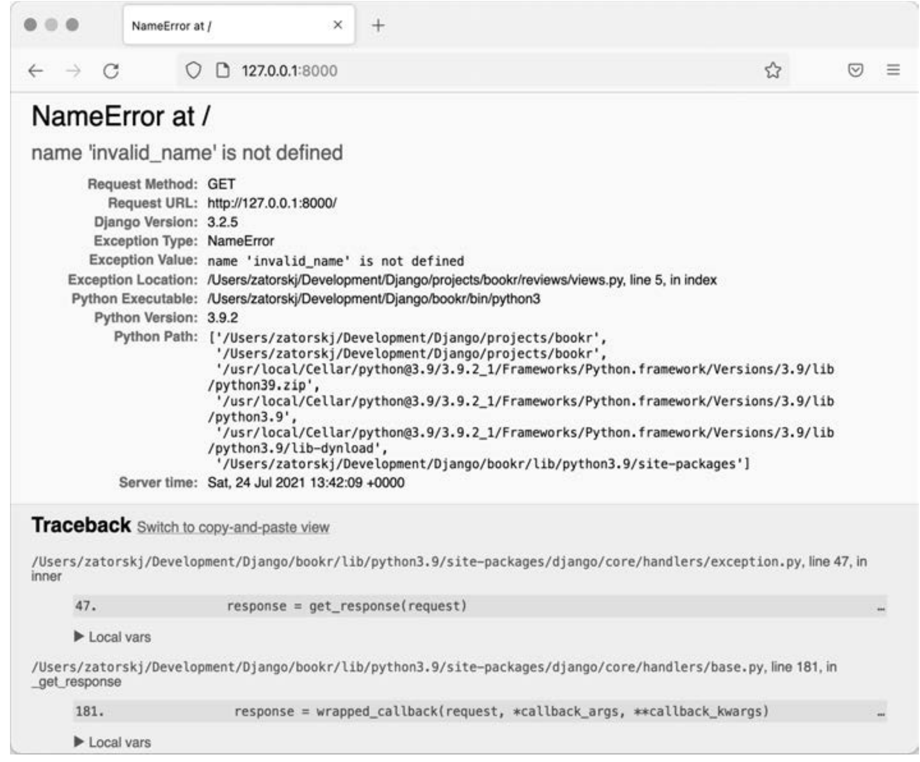
Ćwiczenie 1.8. Generowanie i wyświetlanie wyjątków

Utworzysz teraz prosty wyjątek w widoku, aby sprawdzić, jak Django wyświetla wyjątki. W tym przypadku spróbujesz użyć nieistniejącej zmiennej, co doprowadzi do wygenerowania wyjątku `NameError`.

1. Otwórz plik *views.py* w programie PyCharm. W funkcji widoku `index` zmień kontekst przekazywany do funkcji `render`, aby korzystał z nieistniejącej zmiennej. Spróbuj dodać do słownika kontekstu zmienną `invalid_name`, a nie zmienną `name`. Nie zmieniaj klucza słownika kontekstu, tylko jego wartość:
   ```
   return render(request, "base.html", {"name": invalid_name})
   ```
2. Powróć do przeglądarki i odśwież stronę. Powinieneś uzyskać wynik podobny do przedstawionego na rysunku 1.41.
3. Dwa początkowe nagłówki na stronie informują o tym, jaki błąd został wygenerowany:

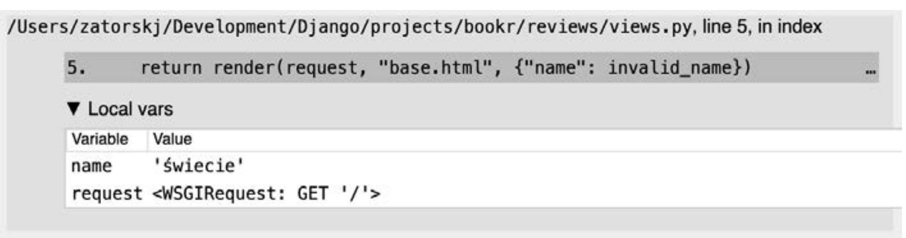
   ```
   NameError at /
   name 'invalid_name' is not defined
   ```


Kup ksi k Pole ksi k



Rysunek 1.41. Ekran wyjątku platformy Django

4. Poniżej nagłówka wyświetlony jest zrzut stosu prowadzący do miejsca zgłoszenia wyjątku. Możesz kliknąć poszczególne wiersze kodu, aby je rozwinąć i przejrzeć otaczający je kod, lub kliknąć napis `Local vars` w każdej ramce, aby ją rozwinąć i przejrzeć wartości zmiennych (zobacz rysunek 1.42).



Rysunek 1.42. Wiersz odpowiedzialny za wyjątek

5. W tym przykładzie widać, że wyjątek został zgłoszony w wierszu 6. pliku *views.py*, a po rozwinięciu zawartości `Local vars` widać, że zmienna `name` ma wartość `świecie`, ponadto jedyną inną zmienną jest przychodzące żądanie, czyli `request`.
6. Powróć do pliku *views.py* i napraw błąd `NameError`, zmieniając nazwę zmiennej `invalid_name` na `name`.

Kup ksi k Pole ksi k

7. Zapisz plik i odśwież stronę w przeglądarce. Na ekranie powinien się ponownie pojawić napis `Witaj, świecie` (jak na rysunku 1.40).

W tym ćwiczeniu napisałeś kod Django generujący wyjątek (`NameError`), próbując użyć nieistniejącej zmiennej. Mogłeś się przekonać, że Django automatycznie wysyła błędy wyjątku i zrzut stosu do przeglądarki, aby ułatwić znalezienie przyczyny błędu. Następnie przywróciłeś poprzednią wersję kodu, aby się upewnić, że widok działa poprawnie.

Debugowanie

Podczas prób znalezienia problematycznych miejsc w kodzie warto się wspomóc debugerem. Jest to narzędzie, które umożliwia wykonywanie wierszy kodu po kolei, a nie wszystkich naraz. Gdy zatrzymasz debuger w określonym wierszu kodu, możesz sprawdzić wartości wszystkich bieżących zmiennych. Ułatwia to bardzo znajdowanie w kodzie błędów, które nie generują wyjątków.

Na przykład w projekcie Bookr rozważaliśmy widok pobierający listę książek z bazy danych i renderujący ją w szablonie HTML. Po wyświetleniu tej strony w przeglądarce może się okazać, że lista zawiera tylko jedną książkę, chociaż powinna zawierać kilka elementów. Można wtedy wstrzymać wykonywanie programu w funkcji widoku i sprawdzić, jakie wartości zostały pobrane z bazy danych. Jeśli widok pobiera tylko jedną książkę z bazy danych, wiadomo, że błąd dotyczy pobierania danych z bazy. Jeśli widok z powodzeniem pobiera wiele książek, a na ekranie wyświetlone są dane tylko jednej z nich, prawdopodobnie problem dotyczy szablonu. Dzięki debugowaniu można zawęzić zakres poszukiwań przyczyny błędu.

W programie PyCharm dostępny jest wbudowany debuger, który ułatwia wykonywanie programu krok po kroku i sprawdzanie, co się dzieje podczas wykonywania każdego wiersza. Aby debuger wstrzymał wykonywanie kodu w określonym miejscu, trzeba ustawić *punkt przerwania* w co najmniej jednym wierszu kodu. Nazwa punktu przerwania wynika z tego, że w tym *punkcie* wykonywanie kodu zostanie *przerwane* (wstrzymane).

Gdy chce się aktywować punkty przerwania, trzeba tak skonfigurować PyCharm, aby uruchamiał projekt w swoim debugerze. W tym trybie obniża się nieco wydajność działania programu, co jest zwykle niezauważalne, a zatem warto zawsze uruchamiać kod w debugerze, aby móc szybko ustawić punkt przerwania bez konieczności zatrzymywania i restartowania serwera roboczego Django.

Uruchomienie serwera roboczego Django w debugerze jest proste i wymaga kliknięcia ikony debugowania, a nie ikony odtwarzania (jak na rysunku 1.19).

Ćwiczenie 1.9. Debugowanie kodu

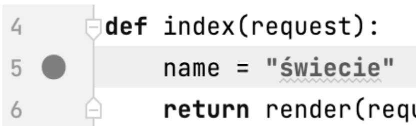
W tym ćwiczeniu poznasz postawy działania debugera programu PyCharm. Uruchomisz serwer roboczy Django w debugerze, a następnie wstawisz punkt przerwania w funkcji widoku, aby wstrzymać wykonywanie programu i przeanalizować zmienne.

Kup ksi k Pole ksi k

1. Jeśli serwer roboczy Django jest uruchomiony, zatrzymaj go, klikając przycisk stop w prawym górnym rogu okna programu PyCharm.
2. Ponownie uruchom serwer roboczy Django w debugerze, klikając ikonę debugowania, znajdującą się z lewej strony przycisku stop (zobacz rysunek 1.43).

Rysunek 1.43. Przycisk stop w prawym górnym rogu okna programu PyCharm

3. Serwer będzie się uruchamiał przez kilka sekund. Następnie będzie można odświeżyć stronę w przeglądarce, aby się upewnić, że się poprawnie wczytuje — nie powinieneś zauważyć żadnych zmian; kod powinien działać tak samo jak wcześniej.
4. Teraz można wstawić punkt przerwania, który spowoduje wstrzymanie wykonywania programu i umożliwi sprawdzenie jego stanu. W programie PyCharm kliknij z prawej strony numeru wiersza 5., na pasku z lewej strony panelu edytora (zobacz rysunek 1.44). W tym miejscu pojawi się czerwone kółko oznaczające aktywny punkt przerwania.



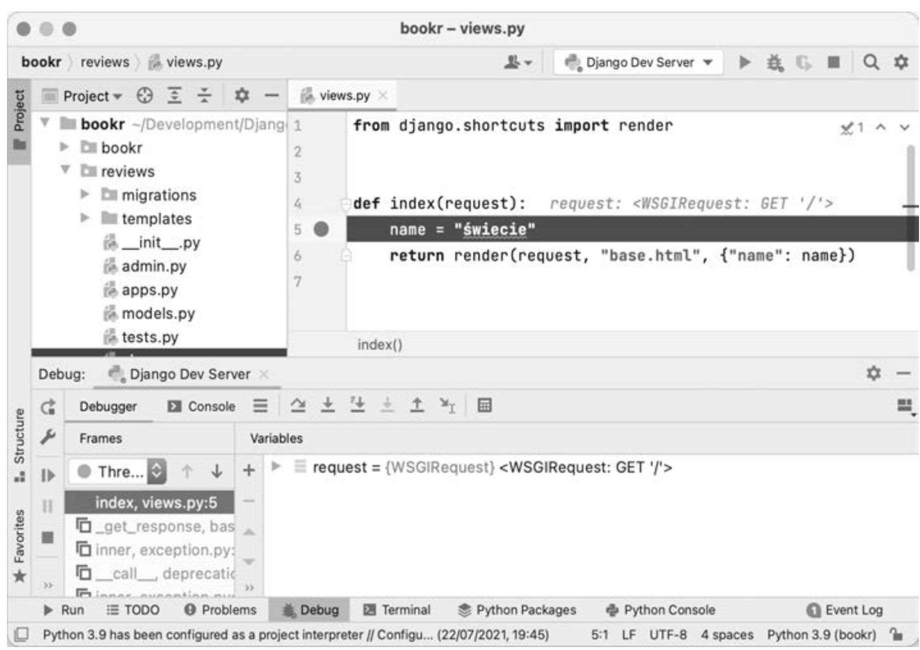
Rysunek 1.44. Punkt przerwania w wierszu nr 5

5. Powróć do przeglądarki i odśwież stronę. Przeglądarka niczego nie wyświetli; będzie nadal wczytywać stronę. W zależności od systemu operacyjnego program PyCharm powinien się ponownie uaktywnić; jeśli nie, powróć do niego. Wiersz 5. powinien być podświetlony, a w dolnej części okna powinien być widoczny debuger (zobacz rysunek 1.45). Ramki stosu (łańcuch funkcji, które zostały wywołane, aby dotrzeć do bieżącego wiersza) znajdują się z lewej strony, a bieżące zmienne funkcji są widoczne z prawej strony.
6. W bieżącym zakresie zdefiniowana jest obecnie tylko jedna zmienna `request`. Jeśli klikniesz trójkątny przełącznik z lewej strony jej nazwy, to wyświetlisz lub ukryjesz przypisane do niej atrybuty (zobacz rysunek 1.46).

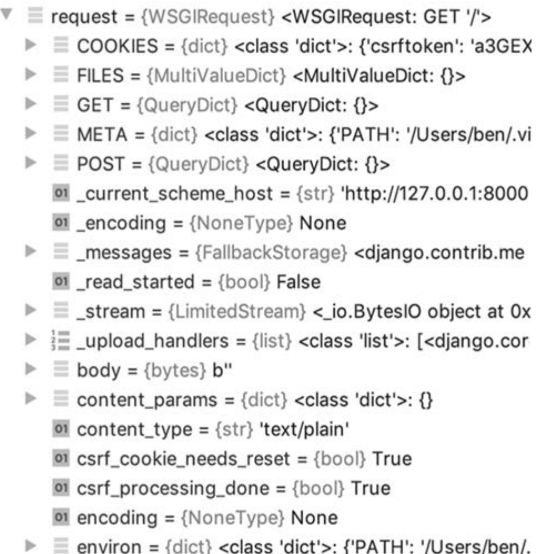
   Jeśli przykładowo przewiniesz w dół listę atrybutów, zauważysz, że metoda ma wartość `GET`, a ścieżka ma postać `/`.
7. Nad ramkami stosu i zmiennymi znajduje się pasek akcji, widoczny na rysunku 1.47. Znajdują się na nim następujące przyciski (od lewej do prawej):
   - *Step Over*

     Wykonuje bieżący wiersz kodu i przechodzi do następnego wiersza.

Kup ksi k Pole ksi k



Rysunek 1.45. Debuger wstrzymujący działanie programu i podświetlający bieżący wiersz (5.)



Rysunek 1.46. Atrybuty zmiennej request

Rysunek 1.47. Pasek akcji

Kup ksi k Pole ksi k

- *Step Into*

  Wchodzi do bieżącego wiersza. Jeśli np. w wierszu jest wywoływana funkcja, debugowanie będzie kontynuowane wewnątrz tej funkcji.

- *Step Into My Code*

  Wejście do bieżącego wiersza, ale kontynuowane aż do znalezienia kodu napisanego przez programistę. Jeśli np. wchodzisz do kodu z zewnętrznej biblioteki, który następnie wywołuje Twój kod, nie zostanie pokazany kod tej biblioteki, ale debuger będzie kontynuował wykonywanie programu, aż dotrze do kodu, który napisałeś.

- *Force Step Into*

  Wejście do kodu, który zwykle bywa pomijany podczas debugowania. Dotyczy to np. kodu biblioteki standardowej Pythona. Ta funkcja jest dostępna tylko w niektórych rzadkich sytuacjach i zwykle nie jest wykorzystywana.

- *Step Out*

  Powraca z bieżącego kodu do funkcji lub metody, która go wywołała. Przeciwieństwo akcji *Step In*.

- *Run To Cursor*

  Jeśli chcesz wykonać wiersz kodu oddalony od bieżącej pozycji debugera i nie chcesz klikać przycisku *Step Over* we wszystkich wierszach znajdujących się po drodze, kliknij w docelowym wierszu, aby umieścić w nim kursor. Następnie kliknij przycisk *Run To Cursor*, a debuger będzie kontynuować wykonywanie programu aż do tego wiersza.

  Zauważ, że nie wszystkie przyciski są potrzebne w każdej sytuacji. Łatwo można np. wyjść poza swój widok i znaleźć się przypadkiem w kodzie biblioteki Django.

8. Kliknij raz przycisk *Step Over*, aby wykonać kod z wiersza 5.
9. Możesz zauważyć, że do listy zmiennych w widoku debugera została dodana zmienna `name` o wartości `świecie` (zobacz rysunek 1.48).

```
01 name = {str} 'świecie'
▶ request = {WSGIRequest} <WSGIRequest: GET '/'>
```

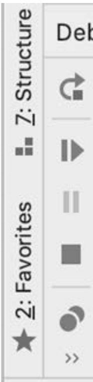
Rysunek 1.48. W zakresie znajduje się obecnie zmienna name z przypisaną wartością świecie

10. Znajdujesz się teraz na końcu funkcji widoku `index` i gdybyś teraz przekroczył ten wiersz kodu, debuger znalazłby się w kodzie biblioteki Django, co jest niepożądane. Aby kontynuować wykonywanie i odesłać odpowiedź do przeglądarki, kliknij przycisk *Resume Program* z lewej strony okna (zobacz rysunek 1.49). Powinieneś zauważyć, że tym razem przeglądarka ponownie wczyta stronę.

    Na rysunku 1.49 znajduje się więcej przycisków; od góry są to przyciski *Rerun* (zatrzymuje wykonywanie programu i go restartuje), *Resume Program* (kontynuuje wykonywanie do następnego punktu przerwania), *Stop* (zatrzymuje działanie debugera), *View Breakpoints* (otwiera okno wyświetlające wszystkie ustawione punkty przerwania) oraz *Mute Breakpoints* (włącza lub wyłącza wszystkie punkty przerwania, ale ich nie usuwa).

Kup ksi k

Pole ksi k



Rysunek 1.49. Akcje kontrolujące wykonywanie — zielona ikona odtwarzania to przycisk Resume Program

11. Na razie kliknij w programie PyCharm punkt przerwania, aby go wyłączyć (czerwone kółko obok wiersza 5.). Powinieneś zobaczyć to, co na rysunku 1.50.

```
4   def index(request):
5       name = "świecie"
6       return render(req
```

Rysunek 1.50. Kliknięcie punktu przerwania w wierszu 5. spowoduje jego wyłączenie

Jest to tylko krótkie wprowadzenie do ustawiania punktów przerwania w programie PyCharm. Jeśli korzystałeś z funkcji debugowania w innych środowiskach IDE, opisane tu koncepcje powinny być znajome — możesz wykonywać kod krok po kroku, przechodzić do funkcji i z nich wychodzić lub sprawdzać wynik wyrażeń. Po ustawieniu punktu przerwania możesz kliknąć go prawym przyciskiem myszy, aby zmienić jego opcje. Np. możesz ustawić warunkowy punkt przerwania, który wstrzyma wykonywanie tylko w pewnych okolicznościach. Wszystkie te funkcje wykraczają poza zakres tej książki, ale warto wiedzieć, że można z nich skorzystać podczas rozwiązywania problemów ze swoim kodem.

Zadanie 1.1. Tworzenie ekranu powitalnego witryny

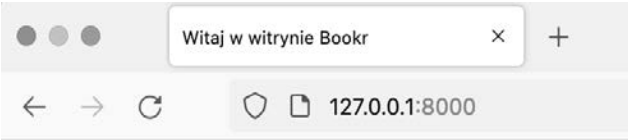
Tworzona w tej książce witryna Bookr wymaga ekranu początkowego, który powita użytkowników i poinformuje ich, w jakiej witrynie się znajdują. Strona ta będzie również zawierać łącza do innych części witryny. Dodamy je jednak w następnych rozdziałach. Teraz utworzysz stronę zawierającą komunikat powitalny.

Wykonaj następujące kroki, aby wykonać to zadanie:

1. Wyrenderuj szablon *base.html* w widoku `index`.
2. Uaktualnij szablon *base.html*, umieszczając w nim komunikat powitalny. Powinien się on znaleźć zarówno w znaczniku `<title>` w elemencie `<head>`, jak i w nowym znaczniku `<h1>` w elemencie `<body>`.

Kup ksi k   Pole ksi k

Po wykonaniu tego zadania przeglądarka powinna wyświetlić stronę podobną do tej przedstawionej na rysunku 1.51.



Rysunek 1.51. Strona powitalna witryny Bookr

> Rozwiązanie tego zadania znajduje się w dodatku A.

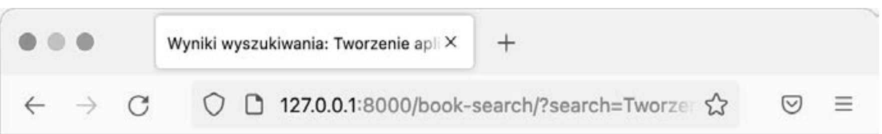
Zadanie 1.2. Szkielet wyszukiwarki w witrynie Book

W witrynach podobnych do witryny Bookr przydaje się możliwość przeszukiwania danych w celu szybkiego znalezienia potrzebnych danych. W witrynie Bookr zaimplementujesz wyszukiwarkę książek, aby użytkownicy mogli znaleźć określoną książkę na podstawie fragmentu jej tytułu. Chociaż jeszcze nie ma żadnych książek do wyszukania, można już utworzyć stronę pokazującą tekst szukany przez użytkownika. Użytkownik wpisze szukany tekst w parametrze `URL`. Wyszukiwanie i formularz do łatwego wpisywania szukanego tekstu zaimplementujesz w rozdziale 6., „Formularze".

Oto kroki potrzebne do wykonania tego zadania.

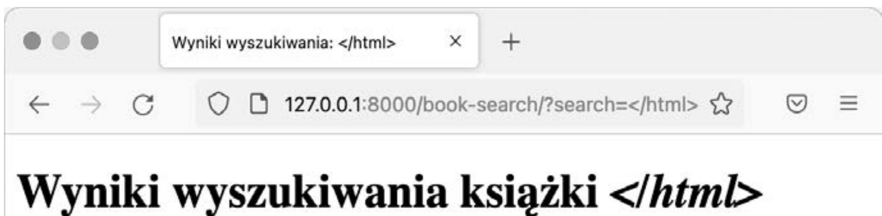
1. Utwórz szablon HTML dla wyników wyszukiwania. Powinien on zawierać tymczasową zmienną, reprezentującą wyszukiwane słowo(-a), przekazane w kontekście renderowania. Pokaż przekazaną zmienną w znacznikach `<title>` i `<h1>`. Szukany termin umieść w znaczniku `<em>` wewnątrz elementu `<body>`, aby wyświetlić go kursywą.
2. Dodaj funkcję widoku wyszukiwania do pliku *views.py*. Widok powinien wczytywać tekst wyszukiwania z parametru URL (dostępnego poprzez atrybut GET żądania). Następnie powinien wyrenderować szablon utworzony w poprzednim kroku, przekazując do niego szukany tekst w słowniku kontekstu.
3. Dodaj do pliku *urls.py* mapowanie URL dotyczące nowego widoku. Ścieżka URL może mieć postać `/book-search`.

Po wykonaniu tego zadania powinieneś móc przekazać szukany tekst poprzez parametry URL i zobaczyć go na wyrenderowanej stronie. Przykładowy podgląd uzyskanej strony wygląda tak jak na rysunku 1.52.



Rysunek 1.52. Szukanie terminu Tworzenie aplikacji internetowych w Django

Powinieneś też móc przekazać specjalne znaki HTML, takie jak < i >, aby sprawdzić, że Django automatycznie stosuje sekwencje ucieczki w zmiennych szablonu (zobacz rysunek 1.53).



Rysunek 1.53. Zauważ, że dla znaków HTML zastosowano sekwencje ucieczki, aby zapobiec wstrzykiwaniu znaczników

> Rozwiązanie tego zadania znajduje się w dodatku A.

Utworzyłeś już szkielet widoku wyszukiwarki książek i możesz pokazać, jak odczytuje się zmienne z parametrów `GET`. Możesz też skorzystać z tego widoku, aby przetestować automatyczne stosowanie sekwencji ucieczki w Django względem znaków specjalnych HTML w szablonie. Widok wyszukiwania jeszcze niczego nie wyszukuje ani nie pokazuje wyników, ponieważ w bazie danych nie ma jeszcze książek. Ten problem rozwiążemy w rozdziale 6., „Formularze".

Podsumowanie

Ten rozdział zawiera krótkie wprowadzenie do platformy Django. Najpierw zapoznałeś się z protokołem HTTP i strukturą żądań i odpowiedzi HTTP. Następnie dowiedziałeś się, jak Django wykorzystuje paradygmat MVT, jak przetwarza ciągi URL, generuje żądania HTTP i przesyła je do widoku, aby uzyskać odpowiedź HTTP. Utworzyłeś szkielet projektu Bookr i dodałeś do niego nową aplikację `reviews`. Następnie utworzyłeś dwa przykładowe widoki, aby pokazać, jak pobrać dane z żądania i wykorzystać je podczas renderowania szablonów. Przećwiczyłeś stosowanie sekwencji ucieczki dla znaków HTML przez Django podczas renderowania szablonu.

Kup ksi k   Pole ksi k

Wszystkie zadania wykonałeś w środowisku IDE PyCharm IDE i dowiedziałeś się, jak je skonfigurować w celu debugowania aplikacji. Debuger ułatwia znajdowanie przyczyn błędnego działania programu. W następnym rozdziale zaczniesz poznawać tematykę integracji z bazą danych w Django, a także poznasz system modeli umożliwiający zapisywanie i pobieranie rzeczywistych danych w aplikacji.

Kup ksi k

Pole ksi k

Skorowidz

A

admin.site, 426
administrator, *Patrz* aplikacja administracyjna
AdminSite, 434
    atrybuty, 428
adresy URL, 53, 561
    generowanie, 210
    mapowanie, 434
    statyczne, 210
aktualizacja formularza, 701
analiza ustawień Django, 61
API, Application Programming Interface, 28, 441, 472
API REST-owe, 472
    tworzenie, 474
aplikacja administracyjna
    aktualizowanie, 156
    dodawanie użytkowników, 159
    domyślny interfejs, 421
    dostosowywanie, 646
        interfejsu, 173
        tekstu, 428
        witryny, 421
    ekran
        logowania, 177
        wylogowania, 163
    filtrowanie, 187
    formularz recenzji, 195
    grupowanie pól, 193
    model danych, 196
    modyfikowanie grup, 159
    nadpisywanie właściwości admin.site, 426
    operacje CRUD, 153
    pasek wyszukiwania, 191
    pobieranie danych, 154
    strona
        główna, 180
        początkowa, 165
        wylogowania, 180, 182
    struktura plików, 422
    szablon
        logowania, 394
        wylogowania, 430
    tworzenie konta, 153
    usuwanie, 172
        konta, 158
    witryna niestandardowa, 424, 425, 428
    wykluczanie pól, 193
    wyszukiwanie, 156
aplikacje
    reviews, 236, 648
    uwierzytelniające, 388, 390, 593
    Django, 44
argument
    help_text, 300
    label, 300
arkusz, 510
    tworzenie, 508
    zapisywanie danych, 508
asercja, 528
    assertEquals, 531
    assertIsInstance, 531
    assertIsNone, 530
    assertRaises, 531
atak CSRF, 255

Kup książkę   Poleć książkę

atrybut
- charset, 345
- content_type, 345
- enctype, 343
- method, 343
- name, 345
- size, 345
- type=text/babel, 617

atrybuty
- klasy AdminSite, 175, 428
- zmiennej request, 77

B

Babel, 617, 618
backend, 471, 601
bazy danych
- nierelacyjne, 85
- relacyjne, 85
  - klucze główne, 100
  - konfiguracja, 93
  - pobieranie obiektów, 121
  - przeszukiwanie relacji, 124
  - tworzenie, 86
  - tworzenie wpisu, 110, 111
  - typy danych, 86
  - wypełnianie danymi, 127
  - wyszukiwanie, 123
  - znajdowanie obiektów, 124
- SQL, 85

bezpieczeństwo, 345, 388, 408
- formularza, 255

biblioteka
- Babel, 617
- Bootstrap, 143
- crispy_forms_tags, 585, 587
- dj_database_url, 561
- Django Crispy Forms, 589
- django-allauth, 592
- django-configurations, 553, 555, 557
- django-crispy-forms, 584
- DRF, 473, 478
- Forms, 266, 282
- PIL, 333, 355, 369
- Pillow, 355, 357
- plotly, 514, 515, 518
- static, 616
- unittest, 531
- weasyprint, 511

biblioteki
- szablonów, 444, 450
- zewnętrzne, 549

bloki warunkowe, 405, 681
błąd
- 404, 39
- dotyczący typu, 409
- KeyError, 59
- ochrony klucza obcego, 173
- ValidationError, 297
- weryfikacji CSRF, 255

BookMediaForm, 384

C

CBV, Class-Based View, 458
charset, 345
ClickCounter, 612
content_type, 345
cookie, 408
CRUD
- w aplikacji administracyjnej, 153
- w Django, 110
- w języku SQL, 85

CSRF, Cross-Site Request Forgery, 245, 255
CSS, Cascading Style Sheets, 199
- ulepszenia, 651

CSV, Comma-Separated Value, 496

D

DatabaseURLValue, 562
DB Browser, 20
- instalowanie w systemie Linux, 22
- instalowanie w systemie macOS, 21
- instalowanie w systemie Windows, 20

debugowanie, 71, 75, 550
dekorator login_required, 399
dekoratory
- przekierowania, 399
- uwierzytelniania, 399, 401

diagram sekwencji uwierzytelniania, 594
DictReader, 503
DictWriter, 503
dj_database_url, 561
Django
- instalowanie, 19

Django Crispy Forms, 589
Django Debug Toolbar, 565
- instalacja biblioteki, 582
- konfiguracja narzędzia, 580
- panel
  - Cache, 578
  - Headers, 571
  - Logging, 580

Kup ksi k Pole ksi k

Profiling, 582
Request, 572
Settings, 570
Signals, 579
SQL, 573–575
Static, 576
Templates, 577
Time, 569
Versions, 568
pasek narzędzi, 566, 567
przełącznik panelu, 583
Django REST Framework, 473
instalacja, 473
konfiguracja, 473
django-allauth, 592
funkcje, 598
inicjalizacja uwierzytelniania, 597
instalacja, 595
konfiguracja, 595
SocialAccount, 595
SocialApplication, 595
SocialApplicationToken, 595
django-configurations, 553
konfiguracja biblioteki, 557
django-crispy-forms, 584
instalacja biblioteki, 589
dj-database-url, 560
konfiguracja biblioteki, 563
dodawanie
aplikacji społecznościowej, 596
globalnego logo, 239
logo, 236, 648
strony profilu, 397
dostawca uwierzytelniania, 592
dostosowywanie
aplikacji administracyjnych, 646
obiektu SiteAdmin, 643
dziedziczenie szablonów, 141, 145

E

edycja
książki, 169
modeli, 165
modelu Publisher, 318
ekran logowania, 393
eksportowanie do pliku XLSX, 523, 695
element <form>, 246
Excel, 505

F

FBV, Function-Based View, 458
FieldFile, 366
FileField, 350–352, 362
magazynowanie plików, 367
w modelu, 371
z plikami, 374
filtr, 139
crispy, 585
date_hierarchy, 190
filtrowanie, 119
według dat, 188
wyszukiwanie pól, 119
wzorce, 120
filtry
niestandardowe, 443–446
szablonów, 442
tekstowe, 448
format
HTML, 34
JSON, 409, 603
PDF, 511
URL, 560
XLSX, 505
FormHelper, 598
aktualizacja formularza, 598
formularze, 244
aktualizacja, 598, 701
atrybuty HTML, 246
bezpieczeństwo, 255
czyszczenie pól, 296
Django, 266, 278
przesyłanie obrazów, 355, 358
przesyłanie plików, 350, 352
dodanie klas Bootstrapa, 586
domyślny styl, 586
edycja recenzji, 331
HTML, 247
przesyłanie plików, 343
metody oczyszczania, 298
niepowiązane, 282
pobieranie danych POST, 258
pola, 245
pola wejściowe, 247
pole DateField, 273
powiązane, 282
recenzji, 193–195
renderowanie, 274, 278
SearchForm, 589
stany, 245

strona
szczegółów książki, 329
tworzenia recenzji, 330
stylowanie i integracja, 324, 663
szablon crispy, 585
w funkcji widoku, 316
walidacja, 266, 282
niestandardowa, 296, 303
pól, 289, 299
w widoku, 286
wartości
początkowe, 310, 312
zastępcze, 310, 312
wbudowana walidacja pól, 288
wysłanie, 261, 284
wyszukiwania, 291, 417
zewnętrzne dane logowania, 592
znacznik szablonów crispy, 587
frontend, 471, 601
funkcja
execute_from_command_line, 555
fetch, 623, 626, 627
main, 555
MD5, 230
onClick, 612
React.createElement, 617
render, 67
skrótu, 230
funkcje strzałkowe, 608

G

generowanie
adresów URL, 210
dokumentu PDF, 512
pliku CSV, 501
wykresów, 514, 515
GET, 57
sprawdzanie wartości, 59
grupy, 159

H

hasła
przechowywanie, 397
help_text, 300
HTML, HyperText Markup Language, 34
HTTP, 36
odpowiedź, 38
żądanie, 37
HttpRequest
GET, 52
headers, 53
method, 52
path, 53
POST, 53

I

ImageField, 357–360, 369
w modelu, 371
z plikami, 374
zapisywanie obrazów, 369
implementowanie
funkcji filtra, 445
przestrzeni nazw, 209
przypadków testowych, 527
uwierzytelniania, 489
widoku, 146
widoku szczegółów, 641
znaczników włączających, 454
index_title, 429
informacje o widokach, 52
instalowanie
DB Browser, 20, 21, 22
django-allauth, 595
django-crispy-forms, 589
Django, 19
Pillow, 359
PyCharm Community Edition, 18
Pythona, 17
interfejs
administracyjny, 173
administracyjny z wyszukiwarką, 684
API, 28
tworzenia instancji modelu, 666

J

JavaScript, 601
funkcje, 606
funkcje strzałkowe, 608
klasy, 607
metoda map, 625
metody, 607
obiekty, 606
obiekty Promise, 622
platformy, 602
pobieranie i wyświetlanie listy, 626
tablice, 606
wczytywanie, 604
zmienne i stałe, 605
język
Django, 138
JavaScript, 601

Kup ksi k Pole ksi k

język
Python, 17
SQL, 33
JSX, JavaScript XML, 617, 618
właściwości komponentów Reacta, 619

K

kaskadowe arkusze stylów, CSS, 199
katalog
final, 24
media, 384
static, 218
templatetags, 444
klasa
AdminConfig, 427
AdminSite, 174, 423, 428
AppDirectoriesFinder, 203, 217
BookSerializer, 491
ClickCounter, 612
Configuration, 555
DatabaseURLValue, 562
DictReader, 503
DictWriter, 503
FieldFile, 366
FileField, 362
FileSystemFinder, 217
FormHelper, 598, 701
ImageField, 358, 365, 369
JSONSerializer, 410
Library, 445
ListValue, 556
LiveServerTestCase, 546
ModelAdmin, 180
ModelForm, 315, 376, 377
ModelViewSet, 484
PasswordInput, 317
ReadOnlyModelViewSet, 484
RequestFactory, 542
SearchForm, 589
SimpleTestCase, 546
TestCase, 528
TransactionTestCase, 546
Value, 556
klasy przypadków testowych, 545
klucz
główny, 100
obcy, 112, 172
sesji, 410
kody stanu, 39
komentarze, 139
komponenty w Reakcie, 610
komunikat, 383
wyświetlanie, 136
konfiguracja
bazy danych, 93
biblioteki
dj-database-url, 563
django-allauth, 595
django-configurations, 557
szablonów, 444, 450
Django Debug Toolbar, 580
Django REST Framework, 473
magazynu plików multimedialnych, 335
ManifestFilesStorage, 231
obiektu Figure, 515
programu PyCharm, 45
strony
wykorzystanie Reacta, 614
URL, 131
uwierzytelniania, 597
ze zmiennych środowiskowych, 556
konstruktor klasy
FileField, 362
ImageField, 365
konto superużytkownika, 151
kontrolka wyboru daty, 171

L

label, 300
Library, 445
lista
Action, 190
Books, 183–186, 189, 192
Bools, 187
Contributors, 197
obiektów, 183
Publisher, 171
Select publisher to change, 166
wyświetlanie, 140
LiveServerTestCase, 546
logo, 236
globalne, 239, 654

Ł

łącze
logowania i wylogowania, 404
Books, 169
Publishers, 165
łączenie szablonów i kodu, 617

Kup ksi k
Pole ksi k

M

magazyn, 231, 234
  format JSON, 409
  plików, 362, 367
  plików multimedialnych, 335
  sesji, 417, 682
ManifestFilesStorage, 231
mapowanie adresów URL, 53, 225, 265, 434
mechanizm ORM, 92, 110
metoda
  __str__(), 184
  add(), 115
  add_worksheet(), 508
  AdminSite.each_context(), 438
  all(), 118
  autodiscover(), 153
  clean, 298
  close(), 508
  create(), 111, 115
  delete(), 126
  each_context(), 434
  form_valid(), 462
  get(), 116, 117
  getlist, 58
  map, 625
  order_by(), 121
  plot, 518
  render, 612
  save, 317, 368
  set(), 115, 125
  setUp(), 531
  tearDown(), 532
  test_method_a(), 529, 532
  test_method_b(), 532
  update(), 125
  write_pdf(), 513
metody modeli, 106
middleware, 388
  moduły, 389
migracja aplikacji reviews, 108
migracje Django, 94
model, 33
  Books, 169
  Django
    testowanie, 532
  Publisher, 167, 663
    edytowanie, 318
    integracja formularza, 324
    stylowanie formularza, 324
    tworzenie, 318
  Review, 105, 666
    interfejs tworzenia instancji, 327
  User, 183
modele
  edytowanie, 314
  magazynowanie plików, 362
  pole FileField, 371
  pole ImageField, 371
  testowanie, 698
  tworzenie, 314
  zapisywanie obrazów, 365
ModelForm, 318, 327, 376, 377
  przesyłanie obrazów, 377
  przesyłanie plików, 376, 377
moduł
  csv, 496
  django.contrib.auth.forms, 391
  django.contrib.auth.views, 391
  middleware, 389
  models, 97
  pickle, 409
  uwierzytelniania
    dekoratory, 399
  xlsxwriter, 510
MVC, Model View Controller, 33
MVT, Model View Template, 33

N

nagłówek
  Content-Length, 38, 39
  Content-Type, 38, 39
  Cookie, 37
  Host, 37
  Server, 39
  Set-Cookie, 39
  User-Agent, 37
name, 345
narzędzie
  Django Debug Toolbar, 550, 565
  dj-database-url, 560
nawias klamrowy, 630
Node.js, 603

O

obiekt
  AdminSite, 174
  Figure, 515
  HttpRequest, 52
  HttpResponse, 40
  Promise, 622

Kup ksi k   Pole ksi k

obiekt
QueryDict, 57
request.user, 397, 403
SiteAdmin, 178, 643
ViewSet, 484
obrazy
przesyłanie, 355, 358, 377, 380, 674
zapisywanie, 365
zapisywanie w ImageField, 369
zmiana rozmiaru, 357
obsługa
błędów, 71
plików CSV, 496
przesłanych plików, 344
okno
administracyjne, 153
New HTML File, 66
Run/Debug Configurations, 208
wyboru pliku, 344
opcja
ALLOWED_HOSTS, 554, 556
MEDIA_ROOT, 334
MEDIA_URL, 334, 338–340
STATIC_ROOT, 334
STATIC_URL, 334
SECRET_KEY, 554, 557
STATICFILES_DIRS, 223
opcje pól modelu, 98
operacje
bazodanowe, 85
CRUD, 89, 110, 464
odczytu, 116
ORM, Object Relational Mapping, 92, 602

P

pakiet
django.http, 55
weasyprint, 513
XlsxWriter, 506
pamięć podręczna, 229
paradygmat MVT, 33
pasek
akcji, 77
nawigacyjny Bootstrapa, 145
wyszukiwania, 191
PDF, Portable Document Format, 511
PIL, Python Imaging Library, 355
platforma
Node.js, 603
React, 604
plik
__init.py__, 44
__init__.py, 43
admin.py, 44, 422, 427
apps.py, 44
asgi.py, 43
db.sqlite3, 95, 109
manage.py, 42, 555
migrations, 45
models.py, 45
settings.py, 43, 335, 553, 554
tests.py, 45
urls.py, 43, 57
views.py, 45, 56, 68
wsgi.py, 43
pliki
administracyjne, 422
binarne, 506
CSV, 496
DictReader, 504
DictWriter, 503
generowanie, 501
odczyt, 503
odczyt danych, 497
użycie indeksów list, 503
zapis, 503
zapis danych, 499
graficzne, *Patrz* obrazy
magazynowanie, 362, 367
multimedialne
konfiguracja magazynu, 335
odwołania w szablonach, 371
przesyłanie, 334
zwracanie, 334, 335
PDF, 380, 511
przesyłanie, 674
strona WWW, 513
pobieranie, 347
przesyłanie, 343, 347–352, 376, 380
statyczne, 201
kopiowanie, 222
przestrzenie nazw, 204
wyszukiwanie, 202, 203, 220
znajdowanie, 226
zwracanie, 201, 206, 218
XLSX, 505, 506
importowanie danych, 523, 695
tworzenie, 508
z wartością skrótu, 232
złośliwe, 345
zwracanie, 361

Kup ksi k Pole ksi k

pobieranie
obiektów, 116
filtrowanie, 118
metoda order_by(), 121
wykluczanie, 120
plików, 347
podgląd recenzji, 633
polecenie
collectstatic, 202, 220, 231
delete, 91
findstatic, 225, 226
insert, 90
makemigrations, 98
pip, 581
pip3, 359
runserver, 336
select, 90
startapp, 44
startproject, 389
touch, 444
update, 91
POST, 57
procesory kontekstu, 338
program pip, 553
projekt
Bookr, 24
Django, 41
Promise, 622
przechowywanie
danych w sesji, 413
haseł, 397
przedrostek, 223
przeglądarki, 345
przekierowanie, 399
przekształcanie stron WWW, 511
przełączanie łączy logowania i wylogowania, 404
przepływ żądania i odpowiedzi, 41
przestrzenie nazw plików statycznych, 204
przesyłanie
obrazów, 355, 358, 380, 674
plików, 343, 347–352, 376, 380, 674
przeszukiwanie relacji, 124
przetwarzanie żądania, 40
punkt końcowy, 483, 692
PyCharm, 206
akcje, 79
ekran powitalny, 46
interpreter projektu, 47
konfiguracja
programu, 45
projektu, 46
serwera roboczego, 51
konsola, 51
odwzorowania URL, 54
okno Add Python Interpreter, 48
panel Project, 47
pasek akcji, 77
pisanie widoku, 54
ustawienia konfiguracyjne, 51
PyCharm Community Edition
instalowanie, 18
Python
instalowanie, 17

R

React, 604, 609
komponenty, 610
konfiguracja strony, 614
właściwości komponentu, 620
recenzje
wyświetlanie, 630, 634, 703
relacja
jeden do jednego, 105
wiele do jednego, 102, 112
wiele do wielu, 103, 114, 115, 124
relacyjne bazy danych, 85
renderowanie
formularzy, 587
szablonu, 67
szczegółów na stronie, 469
wykresu, 515
zmiennych, 69
RequestFactory, 542
REST, Representational State Transfer, 472
routery, 484

S

SearchForm, 589
sekwencje ucieczki, 621
serializery, 475
modeli, 479, 480
serwer roboczy Django, 43
sesje, 388, 407
analiza klucza, 410
przechowywanie danych, 413
wykorzystanie magazynu, 682
zapis danych, 413
zapis formularzy wyszukiwania, 417
silnik
sesji, 408
magazynowania
ManifestFilesStorage, 231
niestandardowy, 234, 366

Kup ksi k
Pole ksi k

SimpleTestCase, 546
site_header, 429
site_title, 429
site_url, 429
size, 345
skoroszyt
  tworzenie, 507
  zapisywanie danych, 508
słowo kluczowe this, 608
SocialAccount, 595
SocialApplication, 595
SocialApplicationToken, 595
SQL, Structured Query Language, 33
  operacje
    aktualizacji, 91
    odczytu, 90
    tworzenia, 90
    usuwania, 91
strona
  Change book, 171
  edycji książek, 170
  logowania, 396
  profilu, 397
  profilu użytkownika, 470
  szczegółów książki, 238
  uwierzytelniania, 393
  użytkownika
    modyfikowanie danych, 518
    renderowanie szczegółów, 689
  WWW
    konwertowanie do formatu PDF, 512
  wyszukiwania, 417
struktura
  katalogów, 204
  plików, 422
style CSS, 199, 237
superużytkownik, 150, 151
szablony
  bloki warunkowe, 405, 681
  dane uwierzytelniania, 403
  dostosowanie wylogowania, 430
  dziedziczenie, 141, 145
  filtry, 139, 442
    niestandardowe, 443–446
  komentarze, 139
  logowanie, 394
  odwołania do plików, 371
  opcja MEDIA_URL, 338, 340
  przekazywanie kluczy, 438
  przełączanie łączy, 404
  renderowanie
    formularza, 274
    w widoku, 67
    zmiennych, 69
  stylowanie, 143
  ustawienia, 340
  uwierzytelnianie, 390
  użycie zmiennych, 70
  witryna administracyjna, 429
  wyświetlenie komunikatu, 136
  zmienne, 138
  znacznik static, 214
  znaczniki, 139, 449
    niestandardowe, 453
    proste, 450
    włączające, 455
szablon crispy, 585
szablony, 34, 135
  bazowe, 64
  HTML, 64
szkielet wyszukiwarki, 638

Ś

ścieżki do szablonów, 392
środowisko
  produkcyjne
    kopiowanie plików, 222
  robocze
    pliki multimedialne, 335

T

tablice, 606
TestCase, 528
testowanie, 525, 527
  klasa RequestFactory, 543
  modeli, 547, 698
    Django, 532
  modularyzacja kodu, 547
  widoków, 547, 698
    Django, 536
    opartych na klasach, 545
    wymagających uwierzytelniania, 539
testy
  automatyczne, 526
  dymne, 527
  funkcjonalne, 527
  integracyjne, 527
  jednostkowe, 527
    tworzenie, 529
    w Django, 528
  regresyjne, 527

Kup książkę Poleć książkę

token, 489
  uwierzytelniania, 493
  wygenerowany, 492
TransactionTestCase, 546
trasowanie dla zbioru widoków, 485
tryb
  debugowania, 550
  STATICFILES_DIRS, 223
tworzenie
  API REST-owego, 474
  aplikacji Django, 93
  arkusza, 508
  ekranu powitalnego, 637
  filtra szablonów, 446
  formularza Django, 266, 278
  formularza HTML, 247
  funkcji widoku, 433
  instancji modelu Review, 327
  katalogu templates, 64
  komponentu, 626
  konta superużytkownika, 150, 151
  migracji Django, 96
  modeli, 96, 127, 639
  modelu Publisher, 318
  niestandardowego prostego znacznika, 451
  plików HTML, 66
  plików XLSX, 508
  podklas AdminSite, 174
  projektu, 30
  prostych znaczników szablonów, 450
  punktu końcowego API, 481, 692
  serializerów modeli, 480
  skoroszytu, 507
  szablonu bazowego, 64
  widoków API, 476, 480
  witryny administracyjnej, 424
  wykresów, 514
typy
  asercji, 530
  pól, 97

U

UploadForm
  z polami, 374
URL
  konfiguracja, 131
  mapowanie adresów, 53
ustawienia szablonu, 340
uwierzytelnianie, 388, 390, 487
  dane, 403
  dekoratory, 399, 401
  implementacja
    szablonów, 390
    widoków, 390
  oparte na tokenach, 489
  udostępnianie treści, 405, 681
  za pomocą
    django-allauth, 597
    GitHuba, 597
    Google'a, 597
użytkownicy, 159

V

ViewSet, 484
virtualenv, 18

W

walidacja formularza, 282, 286
  niestandardowa, 297, 303
  pól, 168, 299
    dodatkowa, 289
    wbudowana, 288
  uwierzytelnionych użytkowników, 539
walidatory niestandardowe, 297
wartość __all__, 315
widok
  API
    oparty na funkcjach, 473
    oparty na klasach, 480, 479
    wyświetlenie listy, 476
  DetailView, 460
  Django, 458
    przypadki testowe, 536
    testowanie, 536
  operacji CRUD, 465–467
  RedirectView, 459
  TemplateView, 459
widoki, 34, 52
  dekoratory uwierzytelniania, 401
  dostęp
    do danych, 258
    do zmiennych szablonu, 433
  generyczne, 479
  mapowanie adresów URL, 434
  ograniczanie, 435
  oparte na funkcjach, 130, 133, 458
  oparte na klasach, 130, 458, 460
    operacje CRUD, 464
    testowanie, 545

Kup ksi k   Pole ksi k

pobieranie danych POST, 258
przesłanie plików, 344
testowanie, 698
klasa RequestFactory, 543
uwierzytelnianie, 390
walidacja formularza, 286
witryna administracyjna, 432
zapis instancji modelu, 376
wymagające uwierzytelniania
testowanie, 539
witryna administracyjna, 150, 421
dodawanie widoków, 432
dostosowanie
tekstu, 428
szablonów, 429
nadpisywanie właściwości, 427
niestandardowa, 424
niestandardowy
interfejs, 439
widok, 435
szablon wylogowania, 430
tworzenie, 424
właściwość admin.site, 426
z wyszukiwarką, 439
wizualizacja, 518
historycznych danych, 518
właściwości
JSX, 619
komponentu Reacta, 620
właściwość admin.site, 426
wydajność, 229
wyjątki, 72
generowanie, 73
wyświetlanie, 73
wykres, 514
generowanie, 515
renderowanie, 515
wykrywanie plików administracyjnych, 422
wyszukiwanie, 123, 417
książek, 656
pola, 124
za pomocą formularza, 591
wyszukiwarka FileSystemFinder, 217
wyszukiwarki plików statycznych, 202, 203, 220
wyświetlanie
listy, 140, 478
okładki i łącza, 384, 679
recenzji, 630, 703
komunikatu, 136
punktu końcowego, 482
wzorzec projektowy MVC, 33

Z

zapisywanie
filtrów szablonów, 444
obrazów, 365, 369
w sesji, 413
zarządzanie
projektami, 127
sesją, 388
zbiory widoków i routerów, 485
zmienna
request, 77
środowiskowa DATABASE_URL, 562
zmienne
szablonu, 138, 438
środowiskowe, 550
znacznik
<script>, 604
load, 616
szablonów, 139, 449
crispy, 587
static, 210, 214
verbatim, 630
znaczniki
proste, 449
niestandardowe, 451
szablonów, 139, 449
włączające, 449, 454, 689
niestandardowe, 455
renderowanie szczegółów, 469
zwracanie
obiektów, 117
plików
multimedialnych, 334, 335
przesłanych, 361
statycznych, 201, 206
z katalogu static, 218

Ż

żądanie
GET, 37, 57, 263
POST, 37, 57, 258, 263

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

DJANGO MA WSZYSTKO, CZEGO WYMAGA NAJLEPSZY PROJEKTANT APLIKACJI WWW!

Django zaskarbił sobie uznanie wielu programistów. Jest to platforma, która udostępnia wszystkie narzędzia potrzebne do tworzenia aplikacji internetowych w Pythonie. To również narzędzie pozwalające na sprawne rozpoczęcie pracy i rozwijanie solidnego i bezpiecznego kodu. Aby jednak zapewnić sobie satysfakcję, a tworzonym projektom profesjonalną jakość, trzeba przyswoić koncepcje i zasady pracy z Django.

Dzięki temu praktycznemu przewodnikowi po Django zdobędziesz wiedzę i pewność siebie potrzebne do budowania rzeczywistych aplikacji w Pythonie. W przystępny sposób opisano tu podstawowe koncepcje i funkcje Django, a następnie pokazano poszczególne etapy cyklu rozwoju rzeczywistej aplikacji internetowej. Dla celów dydaktycznych ten dość złożony projekt został podzielony na zbiór mniejszych zadań, dzięki czemu Twoja nauka będzie przebiegała w sposób efektywny i przemyślany. W trakcie wykonywania ćwiczeń zdobędziesz praktyczne umiejętności, niezbędne do budowy przyjemnych w użytkowaniu aplikacji WWW. Przekonasz się, że Django pozwala na efektywne i satysfakcjonujące budowanie nawet bardzo ambitnych projektów!

W KSIĄŻCE MIĘDZY INNYMI:

- konfiguracja projektu Django, szablony HTML i modele danych w Django
- podstawowe elementy aplikacji internetowej, w tym sesje i uwierzytelnianie
- dodawanie interfejsów API typu REST do aplikacji Django
- korzystanie z zewnętrznych bibliotek Django
- testowanie kodu za pomocą platform testowych Django i Pythona

Ben Shaw programuje w Django od 2007 roku. Interesuje się też uczeniem maszynowym, analizą danych i metodyką DevOps. **Saurabh Badhwar** tworzy rozwiązania zwiększające produktywność programistów. Obecnie zajmuje się wydajnością infrastruktury w LinkedInie. **Andrew Bird** kieruje zespołami programistów i analityków danych w Vesparum. Jest australijskim aktuariuszem. **Bharath Chandra KS** od lat korzysta z frameworków Flask i Django. Zdobył doświadczenie w pracy z mikrousługami. **Chris Guest** programuje w Pythonie od 20 lat, tworzył oprogramowanie dla różnych branż i za pomocą wielu platform Pythona.