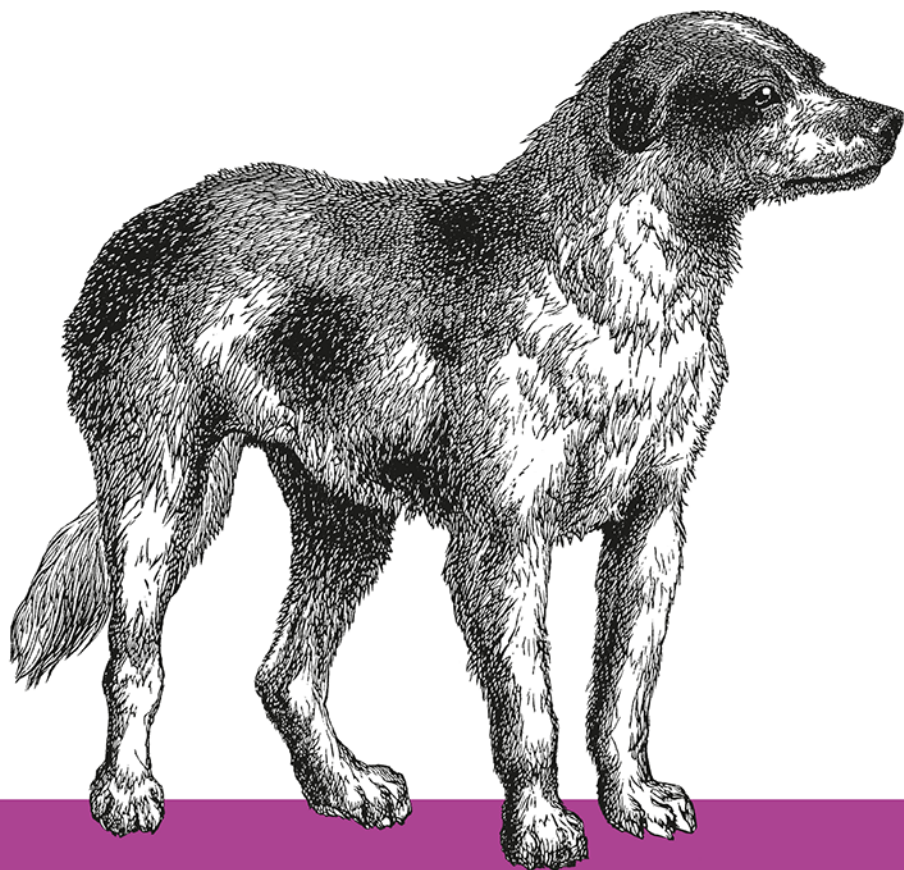


O'REILLY®

Wydanie II



Flask

TWORZENIE APLIKACJI INTERNETOWYCH W PYTHONIE

Helion 

Miguel Grinberg

Tytuł oryginału: Flask Web Development: Developing Web Applications with Python, 2nd Edition

Tłumaczenie: Wojciech Moch

ISBN: 978-83-283-6383-0

© 2020 Helion SA

Authorized Polish translation of the English edition of Flask Web Development 2E ISBN 9781491991732 © 2018 Miguel Grinberg

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Flask Web Development, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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 jakiegokolwiek 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 Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/flask2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Wstęp	11
-------------	----

Część I . Wprowadzenie do Flaska

1. Instalacja	21
Tworzenie katalogu aplikacji	22
Wirtualne środowiska	22
Tworzenie wirtualnego środowiska w Pythonie 3	23
Tworzenie wirtualnego środowiska w Pythonie 2	23
Praca z wirtualnymi środowiskami	24
Instalowanie pakietów Pythona za pomocą narzędzia pip	25
2. Podstawowa struktura aplikacji	27
Inicjalizacja	27
Trasy i funkcje widoku	27
Kompletna aplikacja	29
Roboczy serwer WWW	29
Trasy dynamiczne	31
Tryb debugowania	32
Opcje wiersza polecenia	33
Cykl żądanie – odpowiedź	35
Kontekst aplikacji i żądania	35
Przesyłanie żądania	36
Obiekt żądania	37
Hooki w żądaniach	37
Odpowiedzi	38
Rozszerzenia Flaska	40

3. Szablony	41
Mechanizm szablonów Jinja2	41
Renderowanie szablonów	42
Zmienne	43
Struktury sterujące	44
Integracja Bootstrapa z Flask-Bootstrap	45
Niestandardowe strony błędów	48
Łączy	51
Pliki statyczne	51
Lokalizowanie dat i czasu za pomocą pakietu Flask-Moment	52
4. Formularze internetowe	57
Konfiguracja	57
Klasy formularzy	58
Renderowanie formularzy HTML	59
Obsługa formularzy w funkcjach widoku	61
Przekierowania i sesje użytkownika	64
Wyświetlanie komunikatów	66
5. Bazy danych	69
Bazy danych SQL	69
Bazy danych NoSQL	70
SQL czy NoSQL?	71
Frameworki baz danych w Pythonie	71
Zarządzanie bazą danych za pomocą Flask-SQLAlchemy	73
Definicja modelu	74
Relacje	75
Operacje na bazach danych	77
Tworzenie tabel	77
Wstawianie wierszy	78
Modyfikowanie wierszy	79
Usuwanie wierszy	79
Zapytanie o wiersze	79
Wykorzystanie bazy danych w funkcjach widoku	81
Integracja z powłoką Pythona	82
Migrowanie baz danych za pomocą pakietu Flask-Migrate	83
Tworzenie repozytorium migracji	83
Tworzenie skryptu migracji	84
Aktualizacja bazy danych	85
Dodawanie kolejnych migracji	86

6. Wiadomości e-mail	87
Obsługa e-mail za pomocą rozszerzenia Flask-Mail	87
Wysyłanie wiadomości e-mail z powłoki Pythona	88
Integrowanie wiadomości e-mail z aplikacją	89
Asynchroniczne wysyłanie e-maila	90
7. Struktura dużej aplikacji	93
Struktura projektu	93
Opcje konfiguracji	94
Pakiet aplikacji	96
Korzystanie z fabryki aplikacji	96
Implementacja funkcji aplikacji w projekcie	97
Skrypt aplikacji	100
Plik wymagań	100
Testy jednostkowe	101
Konfiguracja bazy danych	103
Uruchamianie aplikacji	103

Część II. Przykład: Aplikacja do blogowania społecznościowego

8. Uwierzytelnianie użytkownika	107
Rozszerzenia uwierzytelnienia dla Flaska	107
Bezpieczeństwo hasła	107
Haszowanie hasel za pomocą pakietu Werkzeug	108
Tworzenie schematu uwierzytelnienia	110
Uwierzytelnianie użytkownika za pomocą Flask-Login	112
Przygotowywanie modelu User na potrzeby logowania	112
Ochrona tras	113
Dodawanie formularza logowania	114
Logowanie użytkowników	115
Wylogowywanie użytkowników	117
Jak działa Flask-Login?	117
Testowanie	118
Rejestrowanie nowego użytkownika	119
Tworzenie formularza rejestracji użytkownika	119
Rejestracja nowych użytkowników	121
Potwierdzenie konta	122
Generowanie tokenów potwierdzających za pomocą pakietu itsdangerous	122
Wysyłanie wiadomości e-mail z potwierdzeniem	124
Zarządzanie kontem	127

9. Role użytkowników	129
Reprezentacja ról w bazie danych	129
Przypisanie ról	132
Weryfikacja roli	133
10. Profile użytkowników	137
Informacje o profilu	137
Strona profilu użytkownika	138
Edytor profilu	140
Edytor profilu z poziomu użytkownika	140
Edytor profilu z poziomu administratora	142
Awatary użytkownika	144
11. Posty na blogu	149
Przesyłanie i wyświetlanie postów na blogu	149
Wpisy na blogach na stronach profilu	152
Stronicowanie długich list postów na blogu	152
Tworzenie fałszywych danych w postach na blogu	153
Renderowanie na stronach	154
Dodawanie widżetu stronicowania	155
Posty z formatowaniem przy użyciu pakietów Markdown i Flask-PageDown	158
Korzystanie z pakietu Flask-PageDown	158
Obsługa tekstu sformatowanego na serwerze	160
Stałe linki do postów na blogu	161
Edytor postów	162
12. Obserwatorzy	165
I znowu relacje w bazach danych	165
Relacje typu wiele-do-wielu	165
Relacje samoreferencyjne	167
Zaawansowane relacje wiele-do-wielu	168
Obserwujący na stronie profilu	171
Uzyskiwanie śledzonych postów za pomocą operacji Join	173
Wyświetlanie obserwowanych postów na stronie głównej	176
13. Komentarze użytkowników	181
Zapisywanie komentarzy w bazie danych	181
Przesyłanie i wyświetlanie komentarzy	182
Moderowanie komentarzy	184

14. Interfejsy programowania aplikacji	189
Wprowadzenie do architektury REST	189
Zasoby są wszystkim	190
Metody żądania	190
Treści żądań i odpowiedzi	191
Kontrola wersji	192
Flask i usługi sieciowe typu REST	193
Tworzenie schematu interfejsu API	193
Obsługa błędów	194
Uwierzytelnianie użytkownika za pomocą Flask-HTTPAuth	195
Uwierzytelnianie za pomocą tokenów	198
Serializacja zasobów do i z formatu JSON	199
Implementacja punktów końcowych dla zasobów	202
Podział dużych kolekcji zasobów na strony	204
Testowanie usług internetowych za pomocą HTTPie	205

Część III. Ostatnie kroki

15. Testowanie	209
Uzyskiwanie raportów pokrycia kodu	209
Klient testowy Flaska	212
Testowanie aplikacji internetowych	212
Testowanie usług internetowych	215
Kompleksowe testy z użyciem Selenium	217
Czy warto?	221
16. Wydajność	223
Niska wydajność bazy danych	223
Profilowanie kodu źródłowego	225
17. Wdrożenie	227
Etapy prac wdrożenia	227
Protokołowanie błędów na produkcji	228
Wdrożenie w chmurze	229
Platforma Heroku	230
Przygotowanie aplikacji	230
Testowanie z wykorzystaniem Heroku Local	237
Wdrażanie za pomocą polecenia git push	238
Wdrażanie aktualizacji	239

Kontenery na platformie Docker	240
Instalowanie Dockera	240
Budowanie obrazu kontenera	241
Uruchamianie kontenera	244
Sprawdzanie działającego kontenera	245
Przekazywanie obrazu kontenera do rejestru zewnętrznego	246
Korzystanie z zewnętrznej bazy danych	247
Orkiestracja kontenerów za pomocą Docker Compose	248
Sprzątanie starych kontenerów i obrazów	251
Korzystanie z platformy Docker podczas produkcji	252
Tradycyjne wdrożenia	252
Konfiguracja serwera	253
Importowanie zmiennych środowiskowych	253
Konfigurowanie protokołowania	254
18. Dodatkowe zasoby	255
Korzystanie ze zintegrowanego środowiska programistycznego (IDE)	255
Wyszukiwanie rozszerzeń	256
Uzyskiwanie pomocy	256
Angażowanie się w społeczność Flaska	257

Podstawowa struktura aplikacji

W tym rozdziale poznasz różne części aplikacji Flaska. Napiszesz i uruchomisz też swoją pierwszą aplikację internetową Flaska.

Inicjalizacja

Wszystkie aplikacje Flaska muszą utworzyć *instancję aplikacji* (ang. *application instance*). Serwer WWW przekazuje wszystkie żądania otrzymane od klientów do tego obiektu obsługującego, używając przy tym protokołu o nazwie Web Server Gateway Interface (WSGI, wymawiane „wiz-gii”). Instancja aplikacji jest obiektem klasy `Flask`, który zwykle tworzony jest w ten sposób:

```
from flask import Flask
app = Flask(__name__)
```

Jedynym wymaganym argumentem konstruktora klasy `Flask` jest nazwa głównego modułu lub pakietu tej aplikacji. W przypadku większości aplikacji w Pythonie poprawną wartością dla tego argumentu jest zmienna `__name__`.



Argument `__name__`, przekazywany do konstruktora klasy `Flask`, jest źródłem zamieszania wśród nowych programistów frameworka Flask. Flask używa tego argumentu do wyznaczenia lokalizacji aplikacji, co z kolei pozwala zlokalizować inne pliki aplikacji, takie jak obrazy lub szablony.

Troszkę później poznasz bardziej złożone sposoby inicjalizowania aplikacji, ale w przypadku prostych aplikacji to naprawdę wszystko, czego potrzeba.

Trasy i funkcje widoku

Klienci, takie jak przeglądarki internetowe, wysyłają *żądania* do serwera WWW, który z kolei przesyła je do instancji aplikacji Flaska. Instancja aplikacji Flaska musi wiedzieć, jaki kod musi uruchomić w celu obsłużenia adresu URL podanego w żądaniu, dlatego wstępnie definiowane jest odwzorowanie adresów URL na funkcje Pythona. Powiązanie adresu URL z funkcją, która go obsługuje, nazywane jest *trasą* (ang. *route*).

Najwygodniejszym sposobem zdefiniowania trasy w aplikacji Flaska jest użycie dekoratora `app.route`, który jest udostępniany przez instancję aplikacji. Poniższy przykład pokazuje, jak można zadeklarować trasę przy użyciu tego dekoratora:

```
@app.route('/')
def index():
    return '<h1>Witaj, świecie!</h1>'
```



Dekoratory są standardowym elementem języka Python. Zazwyczaj są używane do rejestrowania funkcji jako funkcji obsługi zdarzeń. Takie funkcje są wywoływane w momencie wystąpienia określonego zdarzenia.

Poprzedni przykład rejestruje funkcję `index()` jako funkcję obsługi głównego adresu URL aplikacji. Co prawda dekorator `app.route` jest zalecaną metodą rejestrowania funkcji widoku, ale Flask udostępnia również bardziej tradycyjny sposób konfigurowania tras w aplikacji za pomocą metody `app.add_url_rule()`. W swojej najbardziej podstawowej formie funkcja ta przyjmuje trzy argumenty: adres URL, nazwę punktu końcowego oraz funkcję widoku. W poniższym przykładzie użyto metody `app.add_url_rule()` do zarejestrowania funkcji `index()`. Przedstawiony tu kod działa dokładnie tak samo jak ten pokazany wyżej:

```
def index():
    return '<h1>Witaj, świecie!</h1>'
app.add_url_rule('/', 'index', index)
```

Funkcje obsługujące adresy URL aplikacji, takie jak funkcja `index()`, nazywane są *funkcjami widoku* (ang. *view functions*). Jeżeli aplikacja zostanie umieszczona na serwerze z przypisaną domeną `www.przyklad.pl`, wówczas wpisanie w przeglądarce adresu `http://www.przyklad.pl/` spowoduje wywołanie na serwerze funkcji `index()`. Wartość zwracana przez funkcję widoku jest *odpowiedzią* przesyłaną do klienta. Jeśli klient jest przeglądarką internetową, to odpowiedzią jest dokument wyświetlany użytkownikowi w oknie przeglądarki. Odpowiedź zwrócona przez funkcję widoku może być prostym ciągiem znaków zawierającym kod HTML, ale może również przybierać bardziej złożone formy, o czym będziemy mówić później.



Osadzanie ciągów znaków z kodem HTML w plikach źródłowych Pythona powoduje, że utrzymanie takiego kodu jest bardzo trudne. Przykłady w tym rozdziale służą jedynie do zaprezentowania koncepcji odpowiedzi. W rozdziale 3. poznasz lepszy sposób generowania odpowiedzi HTML.

Jeśli zwrócisz uwagę na to, jak zbudowane są adresy URL usług, z których korzystasz na co dzień, to zauważysz, że wiele z nich ma zmienne sekcje. Na przykład adres URL strony profilu na Facebooku ma format `https://www.facebook.com/<twoja-nazwa>`, który zawiera Twoją nazwę użytkownika, dzięki czemu adres jest inny dla każdego użytkownika. Flask obsługuje te rodzaje adresów URL przy użyciu specjalnej składni dekoratora `app.route`. Poniższy przykład definiuje trasę, która ma składnik dynamiczny:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Witaj, {}!</h1>'.format(name)
```

Część adresu URL trasy, która zawarta jest w nawiasach ostrokątnych, to właśnie część dynamiczna. Wszystkie adresy URL pasujące do części statycznych zostaną zmapowane na tę trasę, a po wywołaniu funkcji widoku komponent dynamiczny zostanie przekazany jako jej argument. W poprzednim przykładzie argument `name` służył do wygenerowania odpowiedzi zawierającej spersonalizowane powitanie.

Dynamiczne komponenty w trasach są domyślnie ciągami znaków, ale mogą mieć też inne typy. Na przykład trasa `/user/<int:id>` będzie pasowała tylko do adresów URL, które mają liczbę całkowitą w segmencie dynamicznym `id`, taką jak na przykład `/user/123`. Flask obsługuje dla tras takie typy jak `string`, `int`, `float` i `path`. Typ `path` to specjalny typ ciągu znaków, który w przeciwieństwie do typu `string` może zawierać ukośniki.

Kompletna aplikacja

W poprzednich podrozdziałach dowiedziałeś się o różnych częściach aplikacji internetowej Flask, a teraz nadszedł czas, aby napisać pierwszą taką aplikację. Skrypt aplikacji `hello.py` pokazany na listingu 2.1 definiuje instancję aplikacji oraz pojedynczą trasę i funkcję widoku w sposób, w który opisano to już wcześniej.

Listing 2.1. `hello.py`: Kompletna aplikacja Flaska

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
    return '<h1>Witaj, świecie!</h1>'
```



Jeśli sklonowałeś repozytorium Git naszej przykładowej aplikacji z GitHuba, możesz teraz uruchomić polecenie `git checkout 2a`, aby pobrać tę wersję aplikacji.

Roboczy serwer WWW

W aplikacjach Flaska dostępny jest też roboczy serwer WWW, który można uruchomić za pomocą instrukcji `flask run`. Instrukcja ta szuka w zmiennej środowiskowej `FLASK_APP` nazwy skryptu Pythona zawierającej instancję aplikacji.

Aby uruchomić aplikację `hello.py` z poprzedniego podrozdziału, proszę się najpierw upewnić, że utworzone wcześniej wirtualne środowisko jest aktywne i ma zainstalowany w sobie Flask. W przypadku użytkowników systemów Linux i macOS proszę uruchomić serwer WWW w następujący sposób:

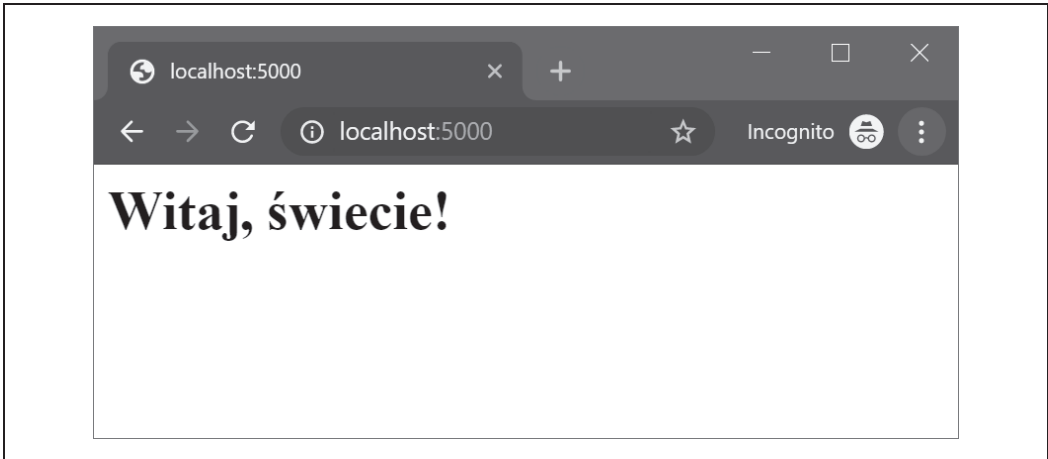
```
(venv) $ export FLASK_APP=hello.py
(venv) $ flask run
* Serving Flask app "hello"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Natomiast w przypadku użytkowników systemu Microsoft Windows jedyną różnicą jest sposób ustawienia zmiennej środowiskowej `FLASK_APP`:

```
(venv) $ set FLASK_APP=hello.py
(venv) $ flask run
* Serving Flask app "hello"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Po uruchomieniu serwer wchodzi w pętlę, która przyjmuje żądania i obsługuje je. Pętla ta jest wykonywana do momentu zatrzymania aplikacji przez naciśnięcie kombinacji klawiszy *Ctrl+C*.

Przy uruchomionym serwerze otwórz przeglądarkę internetową i wpisz w pasku adresu **http://localhost:5000/**. Na rysunku 2.1 przedstawiam, co zobaczysz po połączeniu z aplikacją.



Rysunek 2.1. Aplikacja Flaska *hello.py*

Natomiast jeśli wpiszesz coś innego po podstawowym adresie URL, aplikacja nie będzie wiedziała, jak sobie z tym poradzić, i zwróci przeglądarce błąd o kodzie 404 — jest to znany błąd, który pojawia się, gdy próbujesz przejść do nieistniejącej strony internetowej.



Serwer internetowy dostarczony przez Flaska jest przeznaczony wyłącznie do tworzenia oprogramowania i do testowania. O produkcyjnych serwerach WWW dowiesz się więcej w rozdziale 17.



Roboczy serwer WWW dołączany do pakietu Flask można również uruchomić programowo, wywołując metodę `app.run()`. Starsze wersje pakietu Flask nie miały instrukcji `flask`, dlatego wymagały uruchomienia serwera w ramach głównego skryptu aplikacji. Skrypt ten musiał zawierać na końcu poniższy fragment kodu:

```
if __name__ == '__main__':
    app.run()
```

Instrukcja `flask run` sprawia, że takie działanie nie jest już potrzebne, jednak w niektórych przypadkach metoda `app.run()` może okazać się nadal przydatna. Na przykład podczas tworzenia testów jednostkowych, o których dowiedziesz się więcej w rozdziale 15.

Trasy dynamiczne

Nowa wersja aplikacji, pokazana na listingu 2.2, definiuje drugą trasę, która tym razem jest dynamiczna. Po wprowadzeniu w przeglądarce dynamicznego adresu URL pojawi się spersonalizowane powitanie zawierające imię podane w adresie.

Listing 2.2. hello.py: Aplikacja Flaska z dynamiczną trasą

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Witaj, świecie!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Witaj, {}!</h1>'.format(name)
```



Jeśli sklonowałeś repozytorium Git naszej aplikacji z GitHuba, możesz teraz uruchomić polecenie `git checkout 2b`, aby pobrać wersję aplikacji.

W celu przetestowania dynamicznej trasy upewnij się, że serwer jest uruchomiony, a następnie wprowadź w przeglądarce adres <http://localhost:5000/user/Dave>. Aplikacja odpowie spersonalizowanym powitaniem, wykorzystując dynamiczny argument `name`. Spróbuj użyć różnych imion w adresie URL, aby zobaczyć, w jaki sposób funkcja widoku wygeneruje odpowiedź na podstawie podanego imienia. Przykład działania aplikacji przedstawiam na rysunku 2.2.



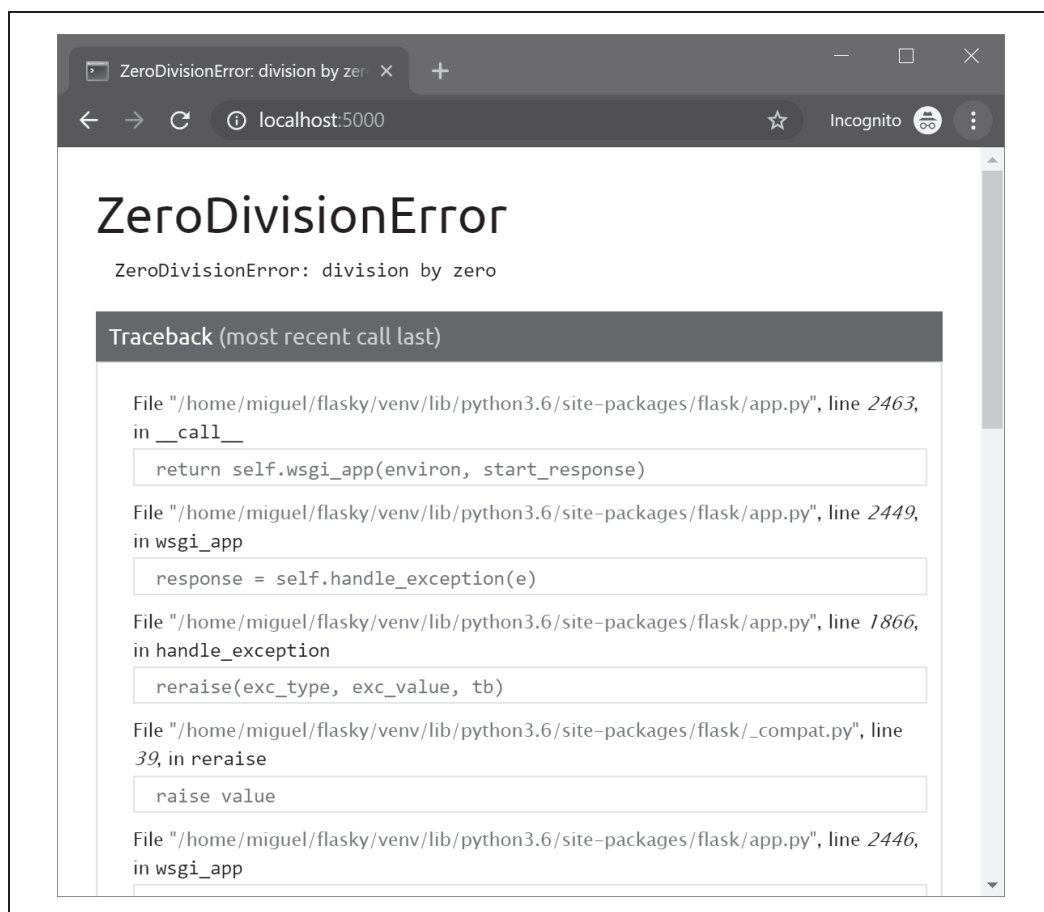
Rysunek 2.2. Trasa dynamiczna

Tryb debugowania

Aplikacje Flaska można opcjonalnie uruchamiać w *trybie debugowania* (ang. *debug mode*). W tym trybie domyślnie włączone są dwa bardzo wygodne moduły serwera roboczego, o nazwie *reloader* i *debugger*.

Gdy włączony jest moduł reloader, Flask obserwuje wszystkie pliki kodu źródłowego Twojego projektu i automatycznie restartuje serwer, gdy którykolwiek z plików zostanie zmodyfikowany. Serwer działający z włączonym reloaderem jest niezwykle przydatny podczas programowania, ponieważ za każdym razem, gdy zmodyfikujesz i zapiszesz plik źródłowy, serwer automatycznie ponownie się uruchomi i pobierze zmianę.

Debugger to narzędzie internetowe, które pojawia się w przeglądarce, gdy aplikacja zgłosi nieobsłużony wyjątek. Okno przeglądarki internetowej przekształca się w interaktywny stos wywołań, który umożliwia sprawdzanie kodu źródłowego aplikacji i skontrolowanie wartości wyrażeń w dowolnym miejscu stosu wywołań. Na rysunku 2.3 można zobaczyć, jak wygląda taki debugger.



Rysunek 2.3. Debugger Flaska

Domyślnie tryb debugowania jest wyłączony. Aby go włączyć, przed wywołaniem polecenia `flask run` zdefiniuj zmienną środowiskową `FLASK_DEBUG=1`:

```
(venv) $ export FLASK_APP=hello.py
(venv) $ export FLASK_DEBUG=1
(venv) $ flask run
* Serving Flask app "hello"
* Forcing debug mode on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 273-181-528
```

Jeśli korzystasz z systemu Microsoft Windows, użyj instrukcji `set` zamiast `export`, aby zdefiniować zmienną środowiskową.



Jeśli serwer zostanie uruchomiony za pomocą wywołania metody `app.run()`, to zmienne środowiskowe `FLASK_APP` i `FLASK_DEBUG` nie będą używane. Aby programowo włączyć tryb debugowania, użyj instrukcji `app.run(debug=True)`.



Nigdy nie włączaj trybu debugowania na serwerze produkcyjnym. Przede wszystkim dlatego, że debugger pozwala klientowi zażądać zdalnego wykonania kodu, przez co serwer produkcyjny będzie bardziej podatny na ataki. Na szczęście można tu zastosować prosty środek ochrony i w przypadku próby uruchomienia debuggera wymagać podania numeru PIN wyświetlonego w konsoli po wprowadzeniu polecenia `flask run`.

Opcje wiersza polecenia

Instrukcja `flask` obsługuje wiele opcji. Aby przekonać się, jakie opcje są dostępne, możesz uruchomić instrukcję `flask --help` lub po prostu samo polecenie `flask` bez żadnych argumentów:

```
(venv) $ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...
```

A general utility script for Flask applications.

Provides commands from Flask, extensions, and the application. Loads the application defined in the `FLASK_APP` environment variable, or from a `wsgi.py` file. Setting the `FLASK_ENV` environment variable to 'development' will enable debug mode.

```
$ export FLASK_APP=hello.py
$ export FLASK_ENV=development
$ flask run
```

Options:

```
--version Show the flask version
--help Show this message and exit.
```

Commands:

```
routes Show the routes for the app.
run Run a development server.
shell Run a shell in the app context.
```

Instrukcja `flask shell` służy do uruchamiania sesji powłoki Pythona w kontekście aplikacji. Tej sesji można użyć do uruchamiania zadań konserwacyjnych lub testów albo wykorzystać ją do debugowania problemów. Przykłady sytuacji, w których to polecenie jest naprawdę przydatne, przedstawię później, w kilku następnych rozdziałach.

Znasz już instrukcję `flask run`, która — jak sama nazwa wskazuje — uruchamia aplikację z roboczym serwerem WWW. To polecenie ma wiele różnych opcji:

```
(venv) $ flask run --help
Usage: flask run [OPTIONS]
```

Run a local development server.

This server is for development purposes only. It does not provide the stability, security, or performance of production WSGI servers.

The reloader and debugger are enabled by default if `FLASK_ENV=development` or `FLASK_DEBUG=1`.

Options:

```
-h, --host TEXT           The interface to bind to.
-p, --port INTEGER       The port to bind to.
--cert PATH              Specify a certificate file to use HTTPS.
--key FILE               The key file to use when specifying a
                        certificate.
--reload / --no-reload   Enable or disable the reloader. By default
                        the reloader is active if debug is enabled.
--debugger / --no-debugger
                        Enable or disable the debugger. By default
                        the debugger is active if debug is enabled.
--eager-loading / --lazy-loader
                        Enable or disable eager loading. By default
                        eager loading is enabled if the reloader is
                        disabled.
--with-threads / --without-threads
                        Enable or disable multithreading.
--extra-files PATH       Extra files that trigger a reload on change.
                        Multiple paths are separated by ':'.
--help                   Show this message and exit.
```

Szczególnie przydatny jest argument `--host`, ponieważ mówi serwerowi, na jakim interfejsie sieciowym ma nasłuchiwać połączeń od klientów. Domyślnie roboczy serwer WWW Flaska nasłuchuje połączeń na *hoście lokalnym* (ang. *localhost*), więc przyjmowane są wyłącznie połączenia pochodzące z tego samego komputera. Poniższe polecenie sprawia, że serwer WWW nasłuchuje połączeń w publicznym interfejsie sieciowym, przyjmując również połączenia od innych komputerów w tej samej sieci:

```
(venv) $ flask run --host 0.0.0.0
* Serving Flask app "hello"
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Serwer WWW powinien być teraz dostępny z dowolnego komputera w sieci `http://a.b.c.d:5000`, gdzie `a.b.c.d` to adres IP komputera, na którym działa ten serwer.

Opcje `--reload`, `--no-reload`, `--debugger` i `--no-debugger` zapewniają większą kontrolę nad ustawieniami trybu debugowania. Na przykład jeśli tryb debugowania jest włączony, to można użyć opcji `--no-debugger`, aby wyłączyć debugger przy jednoczesnym zachowaniu aktywnego trybu debugowania i reloadera.

Cykl żądanie – odpowiedź

Teraz, kiedy już pobawiłeś się podstawową aplikacją Flaska, możesz chcieć dowiedzieć się nieco więcej o tym, jaka magia napędza framework Flask. W poniższych punktach opisano niektóre elementy projektu tego frameworka.

Kontekst aplikacji i żądania

Gdy Flask otrzyma żądanie od klienta, musi udostępnić kilka obiektów funkcji widoku, która będzie obsługiwać to żądanie. Dobrym przykładem jest *obiekt żądania* (ang. *request object*), który zawiera dane żądania HTTP wysłanego przez klienta.

Oczywistym sposobem, w jaki Flask mógłby przyznać funkcji widoku dostęp do obiektu żądania, jest wysłanie tego obiektu jako argumentu, ale wymagałoby to, aby każda funkcja widoku w aplikacji miała dodatkowy argument. Sprawy trochę się komplikują, jeśli weźmiemy pod uwagę, że obiekt żądania nie jest jedynym obiektem wymaganym przez funkcję widoku do obsługi danego żądania.

Aby uniknąć zaśmiecania funkcji widoku wieloma argumentami, które nie zawsze są nam potrzebne, framework Flask używa *kontekstów* (ang. *contexts*), aby tymczasowo udostępniać globalnie wybrane obiekty. Dzięki kontekstom można tworzyć takie funkcje widoku jak poniższa:

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Twoją przeglądarką jest {}</p>'.format(user_agent)
```

Zwróć proszę uwagę, że w tej funkcji widoku zmienna `request` jest używana tak, jakby była zmienną globalną. W rzeczywistości zmienna `request` nie może być zmienną globalną. Na serwerze wielowątkowym kilka wątków może jednocześnie pracować nad obsługą żądań od różnych klientów, więc każdy wątek musi widzieć inny obiekt w zmiennej `request`. Konteksty umożliwiają Flaskowi globalne udostępnienie wątkowi pewnych zmiennych bez ingerencji w inne wątki.



Wątek to najmniejsza sekwencja instrukcji, którymi można zarządzać niezależnie. Proces często ma wiele aktywnych wątków, czasami współużytkujących pewne zasoby, takie jak pamięć lub uchwyty plików. Wielowątkowe serwery WWW uruchamiają pulę wątków i wybierają pojedyncze wątki z puli, które mają obsłużyć poszczególne, przychodzące żądania.

Flask ma dwa konteksty: *kontekst aplikacji* i *kontekst żądania* (ang. *application context*, *request context*). Tabela 2.1 pokazuje zmienne udostępniane przez każdy z tych kontekstów.

Tabela 2.1. Zmienne globalne w kontekstach Flaska

Nazwa zmiennej	Kontekst	Opis
<code>current_app</code>	Kontekst aplikacji	Instancja aktywnej aplikacji.
<code>g</code>	Kontekst aplikacji	Obiekt, którego aplikacja może użyć jako pamięci tymczasowej podczas obsługi żądania. Ta zmienna jest resetowana przy każdym żądaniu.
<code>request</code>	Kontekst żądania	Obiekt żądania, które hermetyzuje zawartość żądania HTTP wysłanego przez klienta.
<code>session</code>	Kontekst żądania	Sesja użytkownika, czyli słownik, którego aplikacja może używać do przechowywania wartości „zapamiętywanych” między żądaniem.

Flask aktywuje (lub *przekazuje*) konteksty aplikacji i żądania przed wysłaniem żądania do aplikacji i usuwa je po jego obsłużeniu. Po przekazaniu kontekstu aplikacji w wątku stają się dostępne zmienne `current_app` i `g`. Podobnie przekazanie kontekstu żądania sprawia, że dostępne są zmienne `request` i `session`. Jeśli nastąpi próba dostępu do dowolnej z tych zmiennych bez aktywnego kontekstu aplikacji lub żądania, to wygenerowany zostanie błąd. Nie martw się, jeśli nie jesteś w stanie zrozumieć, do czego mogą się przydać te zmienne. Wszystkie cztery zmienne kontekstu zostaną szczegółowo omówione w tym i w następnym rozdziałach.

Poniższa sesja powłoki Pythona pokazuje, w jaki sposób działa kontekst aplikacji:

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
Traceback (most recent call last):
...
RuntimeError: working outside of application context
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

W tym przykładzie, gdy w sesji kontekst aplikacji nie jest aktywny, wywołanie `current_app.name` nie działa, ale nagle zaczyna działać, gdy tylko kontekst aplikacji zostanie jej przekazany. Zwróć uwagę na to, jak kontekst aplikacji można uzyskać poprzez wywołanie w instancji aplikacji metody `app_context()`.

Przesyłanie żądania

Gdy aplikacja otrzyma żądanie od klienta, musi dowiedzieć się, która funkcja widoku będzie właściwa do obsługi tego żądania. W tym celu Flask poszukuje adresu URL podanego w żądaniu w *mapie adresów URL* aplikacji, która odwzorowuje adresy URL na obsługujące je funkcje widoku. Flask buduje tę mapę przy użyciu informacji podawanych w dekoratorach `app.route` lub działającej w ten sam sposób funkcji `app.add_url_rule()`.

Aby zobaczyć, jak wygląda mapa adresów URL w aplikacji Flaska, możesz w powłoce Pythona przejrzeć mapę utworzoną dla pliku *hello.py*. Zanim spróbujesz, upewnij się, że środowisko wirtualne jest aktywowane:

```
(venv) $ python
```

```
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
     <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
     <Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

Trasy / oraz /user/<name> zostały zdefiniowane w aplikacji za pomocą dekoratora `app.route`. Natomiast trasa /static/<filename> to specjalna trasa dodana przez Flaska, aby zapewnić dostęp do plików statycznych. Nieco więcej o plikach statycznych dowiesz się w rozdziale 3.

Elementy (HEAD, OPTIONS, GET) pokazane w mapie adresów URL to *metody żądania* (ang. *request methods*) obsługiwane przez daną trasę. Specyfikacja HTTP określa, że wszystkie żądania są wysyłane za pomocą metody, która samodzielnie wskazuje rodzaj działania, jakiego klient żąda od serwera. Flask dołącza metodę żądania do każdej trasy, dzięki czemu różne metody żądania wysyłane pod ten sam adres URL mogą być obsługiwane przez różne funkcje widoku. Flask automatycznie zarządza metodami HEAD i OPTIONS, więc w praktyce można powiedzieć, że w tej aplikacji trzy trasy na mapie adresów URL są powiązane z metodą GET, która jest używana, gdy klient zażąda informacji, takich jak na przykład strona internetowa. W rozdziale 4. dowiesz się więcej, jak tworzyć trasy dla innych metod żądań.

Obiekt żądania

Dowiedziałeś się już wcześniej, że Flask udostępnia obiekt żądania jako zmienną kontekstu o nazwie `request`. Jest to niezwykle przydatny obiekt, gdyż zawiera wszystkie informacje, które klient zawarł w żądaniu HTTP. W tabeli 2.2 przedstawiono najczęściej używane atrybuty i metody obiektu żądania Flask.

Hooki w żądaniach

Czasami przydatne jest wykonanie pewnego kodu przed obsłużeniem każdego żądania lub po tym. Na przykład na początku każdego żądania może być konieczne utworzenie połączenia z bazą danych lub uwierzytelnienie użytkownika wysyłającego żądanie. Zamiast w każdej funkcji widoku powielać kod wykonujący te czynności, możemy zarejestrować typowe funkcje, które będą wywoływane przed przekazaniem żądania do obsługi lub po wykonaniu tej czynności.

Hooki w żądaniach są implementowane jako dekoratory. Oto cztery hooki obsługiwane przez Flaska:

`before_request`

Rejestruje funkcję do uruchomienia przed każdym żądaniem.

`before_first_request`

Rejestruje funkcję do uruchomienia tylko przed obsłużeniem pierwszego żądania. Może to być wygodny sposób realizowania zadań związanych z inicjowaniem serwera.

`after_request`

Rejestruje funkcję do uruchomienia po każdym żądaniu, ale tylko wtedy, gdy nie wystąpiły nieobsłużone wyjątki.

Tabela 2.2. Metody żądania obiektu Flask

Atrybut lub metoda	Opis
form	Słownik ze wszystkimi polami formularza przesłanymi wraz z żądaniem.
args	Słownik ze wszystkimi argumentami przekazanymi w zapytaniu w adresie URL.
values	Słownik, który łączy wartości z pól form i args.
cookies	Słownik ze wszystkimi ciasteczkami zawartymi w żądaniu.
headers	Słownik ze wszystkimi nagłówkami HTTP zawartymi w żądaniu.
files	Słownik zawierający wszystkie przesłane pliki dołączone do żądania.
get_data()	Zwraca buforowane dane z treści żądania.
get_json()	Zwraca słownik Pythona ze sparsowanymi danymi w formacie JSON zawartymi w treści żądania.
blueprint	Nazwa schematu (ang. <i>blueprint</i>) Flaska, który obsługuje żądanie. Temat schematów przedstawiam w rozdziale 7.
endpoint	Nazwa punktu końcowego Flaska, który obsługuje żądanie. Flask używa nazwy funkcji widoku jako nazwy punktu końcowego trasy.
method	Metoda żądania HTTP, taka jak GET lub POST.
scheme	Schemat URL (<code>http</code> lub <code>https</code>).
is_secure()	Zwraca wartość <code>True</code> , jeśli żądanie nadeszło przez połączenie zabezpieczone (HTTPS).
host	Host podany w żądaniu, w tym również numer portu, o ile został podany przez klienta.
path	Ścieżka pobrana z adresu URL.
query_string	Zapytanie pobrane z adresu URL jako nieprzetworzona wartość binarna.
full_path	Ścieżka i zapytanie pobrane z adresu URL.
url	Pełny adres URL żądany przez klienta.
base_url	Taki sam jak adres <code>url</code> , ale bez komponentu zapytania.
remote_addr	Adres IP klienta.
environ	Bazowy słownik środowiska WSGI właściwy dla żądania.

teardown_request

Rejestruje funkcję do uruchomienia po każdym żądaniu, nawet jeśli wystąpiły nieobsłużone wyjątki.

Często stosowanym sposobem współdzielenia danych między funkcjami hooków w żądaniu a funkcjami widoku jest użycie zmiennej `g` będącej częścią globalnego kontekstu. Na przykład funkcja `before_request` może załadować zalogowanego użytkownika z bazy danych i zapisać go w zmiennej `g.user`. Wywoływana później funkcja widoku może stamtąd pobrać dane użytkownika.

Kolejne przykłady hooków w żądaniach zostaną zaprezentowane w następnych rozdziałach. Nie martw się zatem, jeśli nie widzisz jeszcze sensu ich stosowania.

Odpowiedzi

Gdy Flask wywołuje funkcję widoku, oczekuje, że jej wartość zwracana będzie odpowiedzią na żądanie. W większości przypadków odpowiedź jest prostym ciągiem znaków, który jest odsyłany do klienta jako strona HTML.

Ale protokół HTTP wymaga, aby odpowiedź na żądanie była czymś więcej niż tylko ciągiem znaków. Bardzo ważną częścią odpowiedzi HTTP jest *kod statusu* (ang. *status code*), której Flask domyślnie przypisuje wartość 200. Jest to kod wskazujący, że żądanie zostało pomyślnie obsłużone.

W przypadku gdy funkcja widoku musi odpowiedzieć innym kodem statusu, może dodać odpowiednią liczbę jako drugą wartość zwracaną obok tekstu odpowiedzi. Na przykład poniższa funkcja widoku zwraca kod statusu 400, który jest kodem nieprawidłowego żądania:

```
@app.route('/')
def index():
    return '<h1>Nieprawidłowe żądanie</h1>', 400
```

Odpowiedzi zwrócone przez funkcje widoku mogą zawierać też trzeci element — tak zwany słownik nagłówków (ang. *dictionary of headers*), który jest dodawany do odpowiedzi HTTP. Przykład niestandardowych nagłówków odpowiedzi przedstawię w rozdziale 14.

Funkcje widoku mogą też zwracać *obiekt odpowiedzi* (ang. *response object*) zamiast jednej, dwu lub trzech wartości zebranych w krotce. Funkcja `make_response()` przyjmuje jeden, dwa lub trzy argumenty (są to te same wartości, które można zwrócić z funkcji widoku) i zwraca przygotowany już obiekt odpowiedzi. Czasami przydatne jest wygenerowanie takiego obiektu w funkcji widoku, a następnie użycie jego metod w ramach dalszej konfiguracji odpowiedzi. Poniższy przykład tworzy obiekt odpowiedzi, po czym umieszcza w nim plik cookie:

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>Ten dokument zawiera plik cookie!</h1>')
    response.set_cookie('odpowiedz', '42')
    return response
```

W tabeli 2.3 prezentuję najczęściej używane atrybuty i metody dostępne w obiektach odpowiedzi.

Tabela 2.3. Obiekt odpowiedzi

Atrybut lub metoda	Opis
<code>status_code</code>	Liczbowy kod stanu HTTP.
<code>headers</code>	Obiekt podobny do słownika ze wszystkimi nagłówkami, które zostaną wysłane z odpowiedzią.
<code>set_cookie()</code>	Dodaje plik cookie do odpowiedzi.
<code>delete_cookie()</code>	Usuwa plik cookie.
<code>content_length</code>	Długość treści odpowiedzi.
<code>content_type</code>	Typ mediów treści odpowiedzi.
<code>set_data()</code>	Wstawia ciąg znaków lub sekwencję bajtów jako treść odpowiedzi.
<code>get_data()</code>	Pobiera treść odpowiedzi.

Istnieje specjalny rodzaj odpowiedzi zwany *przekierowaniem* (ang. *redirect*). Ta odpowiedź nie ma żadnego dokumentu strony; po prostu podaje przeglądarce nowy adres URL. Z przekierowań bardzo często korzysta się podczas pracy z formularzami internetowymi, o czym dowiesz się już w rozdziale 4.

Przekierowanie zwykle składa się z kodu statusu odpowiedzi 302 oraz docelowego adresu URL podanego w nagłówku `Location`. Odpowiedź przekierowania może być wygenerowana ręcznie w trój-elementowej wartości zwracanej lub z obiektem odpowiedzi. Ze względu na częste używanie przekierowań Flask zapewnia funkcję pomocniczą `redirect()`, która pozwala na utworzenie tego typu odpowiedzi:

```
from flask import redirect

@app.route('/')
def index():
    return redirect('http://www.przyklad.com')
```

Kolejna specjalna odpowiedź może zostać przygotowana za pomocą funkcji `abort()`, która używana jest do obsługi błędów. Poniższy przykład zwraca kod statusu 404, jeśli dynamiczny argument `id`, podany w adresie URL, nie jest związany z poprawnym użytkownikiem:

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Witaj, {}</h1>'.format(user.name)
```

Zauważ, że funkcja `abort()` nie wraca już do naszej funkcji, ponieważ zgłasza wyjątek.

Rozszerzenia Flaska

Flask został zaprojektowany do obsługi rozszerzeń. Celowo nie realizuje tak ważnych funkcji jak obsługa baz danych lub uwierzytelnianie użytkowników, dając Ci w ten sposób swobodę wyboru pakietów, które będą najlepiej pasowały do Twojej aplikacji. Nic nie stoi też na przeszkodzie, żeby napisać własne pakiety.

Spółeczność stworzyła szeroki wybór rozszerzeń Flaska przeznaczonych do wielu różnych celów, a jeśli to nie wystarczy, można również użyć dowolnego standardowego pakietu lub biblioteki Pythona. Pierwszego rozszerzenia Flaska użyjesz już w rozdziale 3.

W niniejszym rozdziale wprowadzono pojęcie odpowiedzi na żądania, ale o odpowiedziach można by opowiadać znacznie więcej. Flask zapewnia bardzo dobre wsparcie przy generowaniu odpowiedzi za pomocą *szablonów* (ang. *templates*), a stanowi to tak ważny temat, że został temu poświęcony następny rozdział.

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>

GRUPA
Helion 

Flask: wolność twórcza programisty!

Frameworki bardzo ułatwiają życie programistom. Pozwalają na szybkie tworzenie nawet rozbudowanych aplikacji, ale praca z frameworkiem najczęściej oznacza duże ograniczenia w doborze technologii. Wszystko jest w porządku, dopóki deweloper stosuje dokładnie te rozwiązania, które zaplanowali twórcy frameworka. Jeśli jednak zechce użyć innej bazy danych albo wykorzysta własną, autorską metodę uwierzytelniania użytkowników, może napotkać różne nieprzewidziane problemy. Szczęśliwie nie wszystkie mikrośrodowiska zachowują się w ten sposób. Framework napisany w Pythonie Flask, w odróżnieniu od typowych frameworków, umożliwia swobodne dobieranie technologii i komponentów aplikacji, a nawet tworzenie własnych rozwiązań. Oznacza to, że Flask pozwoli Ci zachować pełną kontrolę nad budowanym oprogramowaniem!

To książka przeznaczona dla twórców aplikacji internetowych, którzy chcą nauczyć się pisać rozbudowane oprogramowanie we Flasku. W praktyczny sposób przedstawia proces tworzenia kompletnej aplikacji, od programu zawierającego zaledwie kilka linii kodu aż po złożone oprogramowanie z wieloma zaawansowanymi rozwiązaniami technicznymi. Rozpoczyna się gruntownym wprowadzeniem do Flaska i stopniowo przechodzi do nieco trudniejszych zagadnień, również takich jak migracje baz danych i API. Porusza też tematykę usług sieciowych typu REST, obsługi błędów, serializacji zasobów oraz korzystania z takich narzędzi jak Selenium czy Heroku. Przemyślany układ treści, którą zilustrowano praktycznymi przykładami kodu, ułatwia prześledzenie procesu stopniowej rozbudowy aplikacji o nowe funkcjonalności.

Miguel Grinberg — jest inżynierem oprogramowania z 25-letnim doświadczeniem. W wolnym czasie zajmuje się fotografią i tworzeniem filmów. Prowadzi własnego bloga o różnorodnej tematyce (blog.miguelgrinberg.com). Urodził się w Buenos Aires w Argentynie, obecnie mieszka w Irlandii, w mieście Drogheda.

W tej książce znajdziesz między innymi:

- zasady programowania we Flasku
- opis struktury aplikacji Flasky i etapy jej budowy
- sposoby korzystania z szablonów
- strategie planowania testów jednostkowych i analizy wydajności aplikacji
- opcje wdrażania aplikacji Flask

Helion 

 helion.pl

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

Sprawdź nasze szkolenia!

SZKOLENIA



AKADEMIA IT & BUSINESS

HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-6383-0



9 788328 363830