O'REILLY®

Szybki jak FastAPI

Projektowanie aplikacji WWW w Pythonie

Bill Lubanovic

Tytuł oryginału: FastAPI: Modern Python Web Development

Tłumaczenie: Janusz Grabis

ISBN: 978-83-289-1296-0

© 2024 Helion S.A.

Authorized Polish translation of the English edition of *FastAPI* ISBN 9781098135508
© 2024 Bill Lubanovic.

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.

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. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/szyfas
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

Część I. Co nowego? 17

1. Współczesna sieć WWW 19
Wprowadzenie 19
Usługi i API 19
Rodzaje API 20
HTTP 21
REST(ful) 21
Formaty danych JSON i API 23
JSON:API 23
GraphQL 24
Współbieżność 24
Warstwy 25
Dane 29
Podsumowanie 29

2. Współczesny Python 30
Wprowadzenie 30
Narzędzia 30
Zaczynamy 31
Python 31
Zarządzanie pakietami 32
Środowiska wirtualne 32
Poetry 33
Formatowanie źródeł 33
Testowanie 33
Kontrola wersji i ciągła integracja 34
Narzędzia webowe 34

Kup ksi k Pole ksi k

API i usługi 34
Zmienne są nazwami 35
Podpowiedzi typów 36
Struktury danych 36
Frameworki webowe 36
Django 37
Flask 37
FastAPI 37
Podsumowanie 37

Część II. Przegląd FastAPI 39

3. Przegląd FastAPI ... 41
Wprowadzenie 41
Czym jest FastAPI? 41
Aplikacja FastAPI 42
Żądania HTTP 45
Ścieżka URL 46
Parametry zapytania 47
Ciało 48
Nagłówek HTTP 49
Różne rodzaje danych w żądaniu 50
Która metoda jest najlepsza? 50
Odpowiedzi HTTP 51
Kod statusu 51
Nagłówki 51
Rodzaje odpowiedzi 52
Konwersja typów 53
Modele i response_model 53
Automatyczna dokumentacja 55
Dane złożone 57
Podsumowanie 57

4. await, współbieżność i wycieczka po Starlette ... 58
Wprowadzenie 58
Starlette 58
Rodzaje współbieżności 59
Przetwarzanie rozproszone i równoległe 59
Procesy systemu operacyjnego 59
Wątki systemu operacyjnego 60
Zielone wątki 60

Kup ksi k Pole ksi k

Wywołania zwrotne 61
Generatory Pythona 61
async, await i asyncio w Pythonie 62
FastAPI i async 63
Bezpośrednie użycie Starlette 65
Interludium: sprzątanie domu zagadek 66
Podsumowanie 67

5. Pydantic, podpowiedzi typów i wycieczka po modelach 68
Wprowadzenie 68
Podpowiedzi typów 68
Grupowanie danych 70
Alternatywy 74
Prosty przykład 75
Walidacja typów 77
Walidacja wartości 78
Podsumowanie 80

6. Zależności 81
Wprowadzenie 81
Czym jest zależność? 81
Problemy z zależnościami 81
Wstrzykiwanie zależności 82
Zależności FastAPI 82
Pisanie zależności 83
Zakres zależności 83
Pojedyncza ścieżka 83
Wiele ścieżek 84
Zakres globalny 85
Podsumowanie 85

7. Porównanie frameworków 86
Wprowadzenie 86
Flask 86
Ścieżka 86
Parametry zapytania 87
Ciało 88
Nagłówek 88
Django 89
Inne cechy frameworków webowych 89
Bazy danych 90

Kup ksi k Pole ksi k

Zalecenia 90
Inne frameworki Pythona 91
Podsumowanie 91

Część III. Tworzymy serwis WWW 93

8. Warstwa sieci ... 95
Wprowadzenie 95
Z góry na dół, z dołu do góry, od środka na zewnątrz? 96
Projekt RESTful API 97
Struktura plików i katalogów w serwisie 98
Pierwszy kod źródłowy 100
Żądania 102
Wiele obiektów trasowania 103
Budowanie warstwy sieci 104
Definiowanie modeli danych 104
Dane wpisane na sztywno 104
Tworzenie funkcji wspólnych dla całego serwisu 105
Tworzenie sztucznych danych 105
Test 109
Użycie zautomatyzowanych formularzy testowych FastAPI 110
Komunikacja z warstwami usług i danych 112
Podział na strony i sortowanie 112
Podsumowanie 113

9. Warstwa usług ... 114
Wstęp 114
Definiowanie usług 114
Układ 115
Ochrona 115
Funkcje 115
Test 117
Inne rzeczy w warstwie usług 118
Logowanie 119
Statystyki, monitorowanie, obserwowalność 119
Śledzenie 119
Inne 119
Podsumowanie 120

Kup książkę Poleć książkę

10. Warstwa danych 121
Wstęp 121
DB-API 121
SQLite 122
Układ 124
Uruchomienie 124
Test 128
Testy pełne 128
Testy jednostkowe 136
Podsumowanie 138

11. Uwierzytelnianie i autoryzacja 139
Wstęp 139
Czy potrzebujesz uwierzytelniania? 140
Metody uwierzytelniania 140
Uwierzytelnianie ogólne: współdzielony sekret 141
Proste uwierzytelnianie indywidualne 143
Uwierzytelnianie zaawansowane 145
OAuth2 145
Model użytkownika 146
Użytkownicy w warstwie danych 146
Użytkownicy w warstwie sztucznych danych 148
Użytkownicy w warstwie usług 149
Użytkownicy w warstwie sieci 151
Test 152
Główna warstwa serwisu 152
Kroki autoryzacji 153
JWT 153
Autoryzacja przez partnerów zewnętrznych — OIDC 154
Autoryzacja 154
Warstwa pośrednia 155
CORS 156
Pakiety innych dostawców 157
Podsumowanie 157

12. Testowanie 158
Wstęp 158
Testowanie API sieciowego 158
Gdzie testować? 159
Co testować? 159
Pytest 160
Układ 161

Kup ksi k Pole ksi k

Automatyczne testy jednostkowe 161
Symulowanie 161
Dublerzy i sztuczne dane 164
Sieć 165
Usługi 167
Dane 169
Automatyczne testy integracyjne 170
Wzorzec repozytorium 171
Pełne testy automatyczne 171
Testy bezpieczeństwa 173
Testowanie wydajności 173
Podsumowanie 174

13. Produkcja 175
Wstęp 175
Publikacja 175
Wiele wątków roboczych 175
HTTPS 176
Docker 176
Usługi chmurowe 177
Kubernetes 177
Wydajność 178
Asynchroniczność 178
Pamięć podręczna 178
Bazy danych, pliki i pamięć 178
Kolejki 179
Sam Python 179
Rozwiązywanie problemów 179
Rodzaje problemów 180
Logowanie 180
Statystyki 180
Podsumowanie 181

Część IV. Galeria 183

14. Bazy danych, analiza danych i odrobinę sztucznej inteligencji 185
Wstęp 185
Alternatywne magazyny danych 185
Relacyjne bazy danych i SQL 186
SQLAlchemy 186
SQLModel 188

Kup ksi k Pole ksi k

SQLite 188
PostgreSQL 188
EdgeDB 189
Nierelacyjne (NoSQL) bazy danych 189
Redis 190
MongoDB 190
Cassandra 190
Elasticsearch 190
Cechy NoSQL w bazach danych SQL 190
Testy wydajnościowe baz danych 191
Analiza danych i sztuczna inteligencja 193
Podsumowanie 194

15. Pliki 195
Wstęp 195
Obsługa części 195
Wysyłanie plików 195
File() 195
UploadFile 196
Pobieranie plików 197
FileResponse 197
StreamingResponse 198
Serwowanie plików statycznych 199
Podsumowanie 200

16. Formularze i szablony 201
Wstęp 201
Formularze 201
Szablony 203
Podsumowanie 205

17. Eksploracja i wizualizacja danych 206
Wstęp 206
Python i dane 206
Wyjście w formacie PVS 206
CSV 207
python-tabulate 208
pandas 208
SQLite jako wejście i sieć jako wyjście 209
Pakiety do wykresów i grafik 210
Pierwszy przykład z testowym wykresem 210

Kup ksi k Pole ksi k

Drugi przykład z histogramem 212
Pakiety map 213
Przykład mapy 214
Podsumowanie 215

18. Gry ... 216
Wstęp 216
Pakiety Pythona dla gier 216
Rozdzielenie logiki gry 217
Projekt gry 217
Warstwa sieci — część I: inicjalizacja gry 218
Warstwa sieci — część II: kroki gry 219
Warstwa usług — część I: inicjalizacja 221
Warstwa usług — część II: wyliczanie wyniku 221
Test 222
Warstwa danych: inicjalizacja 222
Zagrajmy w Cryptonamicon 223
Podsumowanie 224

A Dalsza lektura ... 225

B Stworzenia i ludzie ... 228

Kup ksi k
Pole ksi k

ROZDZIAŁ 3.

Przegląd FastAPI

> FastAPI jest nowoczesnym, szybkim (o wysokiej wydajności) frameworkiem webowym do budowania API przy użyciu Python 3.6+ w oparciu o podpowiedzi typów.
>
> — Sebastián Ramírez, twórca FastAPI

Wprowadzenie

FastAPI (*https://fastapi.tiangolo.com*) został ogłoszony w 2018 roku przez Sebastiána Ramíreza (*https://tiangolo.com*). Pod pewnymi względami jest nowocześniejszy od większości frameworków w języku Python — wykorzystuje bowiem cechy Pythona, które zostały dodane do wersji 3 w ciągu ostatnich kilku lat. Ten rozdział stanowi szybkie omówienie FastAPI i jego głównych cech, z naciskiem na rzeczy, o których chciałbyś wiedzieć, tzn. jak obsługiwać żądania i odpowiedzi webowe.

Czym jest FastAPI?

Jak każdy inny framework webowy, FastAPI pomaga budować aplikacje webowe. Każdy framework ma za zadanie uprościć pewne operacje — przez własną funkcjonalność, pominięcia i wartości domyślne. Jak sugeruje nazwa, celem FastAPI jest tworzenie API, chociaż równie dobrze można wykorzystać ten framework do budowy tradycyjnych aplikacji webowych.

FastAPI informuje na swoich stronach, że ma przewagę w następujących obszarach:

Wydajność
: Szybkość porównywalna w niektórych przypadkach z Node.js i Go, co jest rzadko spotykane w przypadku frameworków Pythona.

Szybkie programowanie
: Brak ostrych krawędzi czy innych dziwactw.

Lepsza jakość kodu
: Podpowiadanie typów i modele pomagają unikać błędów.

Automatycznie generowana dokumentacja i strony testowe
: O wiele prostsze niż ręczne edytowanie opisów OpenAPI.

Kup ksi k Pole ksi k

FastAPI używa następujących cech języka:

- podpowiedzi typów,
- pakietu Starlette do infrastruktury webowej, łącznie ze wsparciem dla asynchroniczności,
- pakietu Pydantic do definicji danych i walidacji,
- własnej integracji mającej na celu wykorzystanie i rozszerzenie innych.

To połączenie tworzy przyjemne środowisko do rozwoju aplikacji webowych, w szczególności usług webowych opartych na RESTful.

Aplikacja FastAPI

Napiszmy malutką aplikację FastAPI — usługę webową z pojedynczym punktem końcowym. Na tę chwilę jesteśmy w warstwie, którą nazwałem warstwą webową, obsługującej jedynie żądania i odpowiedzi. Zacznij od zainstalowania podstawowych pakietów Pythona, z których będziemy korzystać:

- framework FastAPI (*https://fastapi.tiangolo.com*): `pip install fastapi`,
- serwer web Uvicorn (*https://www.uvicorn.org*): `pip install uvicorn`,
- tekstowy klient web HTTPie (*https://httpie.io*): `pip install httpie`,
- pakiet synchronicznego klienta web Requests (*https://requests.readthedocs.io*): `pip install requests`,
- pakiet synchronicznego/asynchronicznego klienta web HTTPX (*https://www.python-httpx.org*): `pip install httpx`.

Chociaż najbardziej znanym tekstowym klientem sieci web jest `curl` (*https://curl.se*), myślę, że HTTPie jest prostszy w użyciu. Poza tym jego domyślną metodą kodowania tekstu jest JSON, co lepiej odpowiada pracy z FastAPI. W dalszej części tego rozdziału zobaczysz zrzut ekranu pokazujący składnię wywołania `curl` z wiersza poleceń, potrzebną, aby wysłać zapytanie do punktu końcowego.

Prześledźmy prosty kod z listingu 3.1, zaczynając od zapisania go w pliku *hello.py*.

Listing 3.1. Nieśmiały punkt końcowy (hello.py)

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Witaj? Świecie?"
```

Oto kilka punktów wartych odnotowania:

- `app` jest głównym obiektem FastAPI, który reprezentuje całą aplikację webową.
- `@app.get("/hi")` jest **dekoratorem ścieżki**. Informuje on FastAPI o następujących rzeczach:
  - żądanie dla adresu `"/hi"` na tym serwerze powinno być skierowane do poniższej funkcji;
  - ten dekorator ma zastosowanie tylko do czasownika `GET` w protokole HTTP. Możesz również odpowiedzieć na inne czasowniki protokołu (`PUT`, `POST` itd.) — każdemu z nich w oddzielnej funkcji.

Kup ksi k   Pole ksi k

- `def get` jest **funkcją ścieżki** — głównym punktem kontaktu z żądaniami i odpowiedziami HTTP. W tym konkretnym przypadku funkcja nie ma argumentów, ale jak się przekonasz później, FastAPI ma znacznie więcej rzeczy pod maską.

Kolejny krok to uruchomienie tej aplikacji na serwerze web. FastAPI nie posiada własnego serwera, ale rekomenduje użycie Uvicorn. Serwer Uvicorn i FastAPI możesz uruchomić na dwa sposoby: zewnętrznie lub wewnętrznie.

Spójrz na listing 3.2 — tutaj Uvicorn jest uruchamiany zewnętrznie, z wiersza poleceń.

Listing 3.2. Uruchomienie Uvicorn z wiersza poleceń

```
$ uvicorn hello:app --reload
```

`hello` odnosi się do pliku *hello.py*, a `app` jest zmienną FastAPI zawartą w środku.

Druga metoda to uruchomienie Uvicorn wewnętrznie w samej aplikacji — pokazuje to listing 3.3.

Listing 3.3. Uruchomienie Uvicorn wewnętrznie

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Witaj? Świecie?"

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("hello:app", reload=True)
```

W obu przypadkach `reload` mówi serwerowi Uvicorn, aby ten przeładował się, jeśli *hello.py* ulegnie zmianie. Z tego automatycznego przeładowywania będzie korzystać bardzo często.

W każdym przypadku będziemy korzystać domyślnie z portu 8000 na Twoim komputerze (`localhost`). Jeśli wolisz inne ustawienia, obie metody — zewnętrzna i wewnętrzna — posiadają argumenty `host` i `port`.

Serwer ma jeden punkt końcowy (`/hi`) i jest gotowy na przyjmowanie żądań.

Przetestujmy go, używając kilku różnych klientów webowych:

- W przypadku przeglądarki wpisz URL w pasku adresu.
- Dla klienta HTTPie wpisz poniższą komendę w wierszu poleceń ($ oznacza jedynie znak zachęty w terminalu, nie przepisuj go).
- Dla bibliotek Requests i HTTPX uruchom interaktywny tryb Pythona i wpisz polecenie za znakiem zachęty (>>>).

Jak zostało wspomniane wcześniej, wpisz tylko część **`pogrubioną czcionką`** — wynik polecenia pokazany jest `normalną czcionką`.

Listingi od 3.4 do 3.7 pokazują różne sposoby przetestowania nowego punktu końcowego serwera `/hi`.

Listing 3.4. Test /hi w przeglądarce

```
http://localhost:8000/hi
```

Listing 3.5. Test /hi z użyciem modułu Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi")
>>> r.json()
'Witaj? Świecie?'
```

Listing 3.6. Test /hi z użyciem HTTPX jest niemal identyczny jak w przypadku Requests

```
>>> import httpx
>>> r = httpx.get("http://localhost:8000/hi")
>>> r.json()
'Witaj? Świecie?'
```

Nie ma znaczenia, czy do testowania FastAPI będziesz używał HTTPX, czy też Requests. Ja posłużę się HTTPX w rozdziale 13., aby pokazać, jak pomocna jest ta biblioteka przy wykonywaniu zapytań asynchronicznych. Dlatego w tym rozdziale będę się posługiwał przykładami z Requests.

Listing 3.7. Test /hi z użyciem HTTPie

```
$ http localhost:8000/hi
HTTP/1.1 200 OK
content-length: 18
content-type: application/json
date: Mon, 22 Jan 2024 07:16:44 GMT
server: uvicorn

"Witaj? Świecie?"
```

W przykładzie z listingu 3.8 użyj przełącznika `-b`, aby pominąć nagłówki odpowiedzi i wyświetlić jedynie ciało.

Listing 3.8. Test /hi z użyciem HTTPie — wyświetlenie jedynie ciała odpowiedzi

```
$ http -b localhost:8000/hi
"Witaj? Świecie?"
```

Listing 3.9 pobiera wszystkie nagłówki żądania oraz odpowiedzi przez dodanie przełącznika `-v`.

Listing 3.9. Test /hi z użyciem HTTPie — pobranie całej odpowiedzi

```
$ http -v localhost:8000/hi
GET /hi HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.2

HTTP/1.1 200 OK
content-length: 18
content-type: application/json
date: Mon, 22 Jan 2024 07:22:08 GMT
server: uvicorn

"Witaj? Świecie?"
```


Kup ksi k Pole ksi k

Niektóre przykłady w tej książce pokazują standardowy wynik działania HTTPie (nagłówki odpowiedzi i ciało), a inne jedynie ciało.

Żądania HTTP

Listing 3.9 zawierał tylko jedno zapytanie: zapytanie `GET` dla adresu `/hi` na serwerze `localhost` i porcie 8000.

Zapytania webowe umieszczają dane w różnych częściach żądania HTTP. FastAPI pozwala na szybki dostęp do nich, niezależnie od tego, gdzie się znajdują. Listing 3.10 pokazuje całą zawartość żądania HTTP, które zostało wysłane na serwer po wykonaniu polecenia `http` w listingu 3.9.

Listing 3.10. Żądanie HTTP

```
GET /hi HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.2
```

Żądanie zawiera następujące elementy:

- czasownik (w powyższym przypadku `GET`) i ścieżkę (tutaj `/hi`),
- **parametry zapytania** (tekst za znakiem `?`, w tym przypadku brak),
- inne nagłówki HTTP,
- ciało żądania (w powyższym zapytaniu nie ma ciała).

FastAPI przepakowuje te elementy do poręcznych definicji:

`Header`
: Nagłówki HTTP.

`Path`
: Adres URL.

`Query`
: Parametry zapytania (po znaku `?`, na końcu adresu URL).

`Body`
: Ciało żądania HTTP.

Sposób, w jaki FastAPI dostarcza dane z poszczególnych części żądania HTTP, jest jednym z najlepszych udogodnień w porównaniu do realizacji tego zadania przez inne frameworki webowe Pythona. Wszystkie argumenty, których potrzebujesz, mogą być zadeklarowane bezpośrednio w funkcji ścieżki przy użyciu definicji z powyższej listy (`Path`, `Query` itd.) i przez funkcje, które sam napiszesz. Jest to technika zwana **wstrzykiwaniem zależności** (ang. *dependency injection*), o której będziemy mówić dalej, w szczególności w rozdziale 6.

Kup ksi k Pole ksi k

Spersonalizujmy odrobinę naszą poprzednią aplikację poprzez dodanie parametru `who`, który zaadresuje `Witaj?` do konkretnej osoby. Spróbujemy różnych sposobów przekazania tego parametru:

- w *ścieżce* URL,
- jako parametr *zapytania*, po znaku ? w URL,
- w *ciele* HTTP,
- jako *nagłówek* HTTP.

Ścieżka URL

Zmodyfikuj *hello.py* zgodnie z listingiem 3.11.

Listing 3.11. Zwracanie ścieżki powitania

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi/{who}")
def greet(who):
    return f"Witaj? {who}?"
```

Kiedy zapiszesz swoje zmiany w edytorze, Uvicorn powinien wystartować ponownie (w przeciwnym razie musielibyśmy tworzyć *hello2.py* i za każdym razem na nowo uruchamiać serwer). Jeśli zrobiłeś literówkę, popraw ją i spróbuj jeszcze raz. Możesz próbować do skutku, Uvicorn nie będzie stwarzał problemów.

Dodanie `{who}` w ścieżce URL (po `@app.get`) nakazuje FastAPI oczekiwać w tym miejscu zmiennej o nazwie `who`. Ta zmienna zostanie następnie przypisana argumentowi `who` w znajdującej się poniżej funkcji `greet()`. Tak wygląda koordynacja pomiędzy dekoratorem ścieżki a funkcją ścieżki.

Nie traktuj tego łańcucha URL (`"/hi/{who}"`) jak tradycyjnego łańcucha formatowanego w języku Python, gdzie wyrażeniu w nawiasach klamrowych powinna odpowiadać zmienna. Wyrażenie jest potrzebne FastAPI do dopasowania elementów adresu URL jako parametrów ścieżki.

Listingi od 3.12 do 3.14 pokazują, jak przetestować ten zmodyfikowany punkt końcowy przy użyciu omówionych wcześniej metod.

Listing 3.12. Test /hi/Mamo w przeglądarce

```
localhost:8000/hi/Mamo
```

Listing 3.13. Test /hi/Mamo z użyciem HTTPie

```
$ http localhost:8000/hi/Mamo
HTTP/1.1 200 OK
content-length: 14
content-type: application/json
date: Tue, 23 Jan 2024 11:34:00 GMT
server: uvicorn

"Witaj? Mamo?"
```


Kup ksi k   Pole ksi k

Listing 3.14. Test /hi/Mamo z modułem Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi/Mamo")
>>> r.json()
'Witaj? Mamo?'
```

W każdym przypadku łańcuch `"Mamo"`, przekazywany jako część adresu URL, trafia do funkcji ścieżki `greet()` jako zmienna `who`, a następnie staje się częścią zwróconej odpowiedzi.

Odpowiedzią w każdym przypadku jest łańcuch JSON (w zależności od użytego klienta w pojedynczych lub podwójnych cudzysłowach): `"Witaj? Mamo?"`.

Parametry zapytania

Parametry zapytania mają postać łańcuchów `nazwa=wartość` umieszczonych za znakiem zapytania (`?`) w adresie URL. Znakiem rozdzielającym kolejne pary nazwa – wartość jest `&`. Zmodyfikuj `hello.py` jeszcze raz zgodnie z listingiem 3.15.

Listing 3.15. Zwracanie parametru powitania z zapytania

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet(who):
    return f"Witaj? {who}?"
```

Funkcja punktu końcowego jest zdefiniowana jako `greet(who)`, ale `{who}` nie występuje teraz w znajdującym się wyżej wierszu dekoratora, zatem FastAPI zakłada, że `who` jest parametrem zapytania. Sprawdzają to listingi 3.16 i 3.17.

Listing 3.16. Test listingu 3.15 z użyciem przeglądarki

```
localhost:8000/hi?who=Mamo
```

Listing 3.17. Test listingu 3.15 z użyciem HTTPie

```
$ http -b localhost:8000/hi?who=Mamo
"Witaj? Mamo?"
```

W teście z listingu 3.18 możesz wywołać HTTPie z parametrem zapytania w formie argumentu (zwróć uwagę na podwójny znak równości).

Listing 3.18. Test listingu 3.15 z użyciem HTTP i parametru

```
$ http -b localhost:8000/hi who==Mamo
"Witaj? Mamo?"
```

Możesz mieć więcej tego typu argumentów dla HTTPie, a rozdzielanie ich spacją ułatwia ich wprowadzanie.

Listingi 3.19 i 3.20 pokazują te same możliwości w przypadku biblioteki Requests.

Listing 3.19. Test listingu 3.15 z użyciem Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi?who=Mamo")
```


Kup ksi k   Pole ksi k

```
>>> r.json()
'Witaj? Mamo?'
```

Listing 3.20. Test listingu 3.15 z użyciem Requests i params

```
>>> import requests
>>> params = {"who" : "Mamo" }
>>> r = requests.get("http://localhost:8000/hi", params=params)
>>> r.json()
'Witaj? Mamo?
```

W każdym przypadku przekazujesz łańcuch `"Mamo"` w nowy sposób, dostarczasz go do funkcji ścieżki i w końcu do odpowiedzi.

Ciało

Do punktu końcowego `GET` możemy dostarczyć ścieżkę i parametry żądania, ale nie wartości znajdujące się w ciele żądania. Przyjmuje się, że w HTTP żądanie `GET` jest *niezmienne* (ang. *idempotent*) — określenie, które w informatyce można sprowadzić do stwierdzenia: *jeśli zadajesz to samo pytanie, otrzymujesz taką samą odpowiedź*. HTTP `GET` powinno jedynie zwracać rzeczy. Ciało żądania używane jest do wysyłania rzeczy serwerowi podczas tworzenia (`POST`) lub aktualizacji (`PUT` lub `PATCH`). Rozdział 9. pokazuje, jak obejść to ograniczenie.

Tymczasem zmieńmy punkt końcowy z `GET` na `POST`, jak na listingu 3.21. Technicznie rzecz biorąc, niczego nie tworzymy, zatem `POST` nie jest prawidłowym podejściem, ale raczej nie spodziewamy się pozwu sądowego od twórców RESTful.

Listing 3.21. Zwracanie ciała powitania

```
from fastapi import FastAPI, Body

app = FastAPI()

@app.post("/hi")
def greet(who:str = Body(embed=True)):
    return f"Witaj? {who}?"
```

`Body(embed=True)` jest potrzebne, aby powiedzieć FastAPI, że wartość `who` pochodzi tym razem ze sformatowanego w JSON-ie ciała żądania. Część `embed` oznacza, że ciało nie powinno być zwykłym słowem `"Mamo"`, ale powinno wyglądać tak: `{"who" : "Mamo"}`.

Spróbuj zrobić test z użyciem HTTPie i flagi `-v`, aby pokazać wygenerowane ciało żądania (zwróć uwagę na pojedynczy parametr ze znakiem równości, oznaczający dane JSON w ciele) — listing 3.22.

Listing 3.22. Test listingu 3.21 z użyciem HTTPie

```
$ http -v localhost:8000/hi who=Memo
POST /hi HTTP/1.1
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 15
Content-Type: application/json
Host: localhost:8000
```


Kup ksi k   Pole ksi k

```
User-Agent: HTTPie/3.2.2

{
    "who": "Memo"
}


HTTP/1.1 200 OK
content-length: 14
content-type: application/json
date: Wed, 24 Jan 2024 07:46:38 GMT
server: uvicorn

"Witaj? Memo?"
```

I jeszcze listing 3.23 z użyciem biblioteki Requests, gdzie do przekazania danych w formie JSON w ciele żądania używamy argumentu `json`.

Listing 3.23. Test listingu 3.21 przy użyciu biblioteki Requests

```
>>> import requests
>>> r = requests.post("http://localhost:8000/hi", json={"who" : "Mamo"})
>>> r.json()
'Witaj? Mamo?'
```

Nagłówek HTTP

Na końcu spróbujmy przekazać argument powitania jako nagłówek HTTP — listing 3.24.

Listing 3.24. Odpowiedź wartością przekazaną w nagłówku powitania

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.post("/hi")
def greet(who:str = Header()):
    return f"Witaj? {who}?"
```

Przetestujmy ten kod, używając HTTPie — listing 3.25. W środku widać wyrażenie *`name:value`* reprezentujące nagłówek HTTP.

Listing 3.25. Test listingu 3.24 przy użyciu HTTPie

```
$ http -v POST localhost:8000/hi who:Mamo
POST /hi HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 0
Host: localhost:8000
User-Agent: HTTPie/3.2.2
who: Mamo

HTTP/1.1 200 OK
content-length: 14
content-type: application/json
date: Wed, 24 Jan 2024 08:01:33 GMT
server: uvicorn

"Witaj? Mamo?"
```


Kup ksi k Pole ksi k

FastAPI zmienia wszystkie litery w nazwie nagłówka HTTP na małe, a znak myślnika (-) zastępuje podkreśleniem (_). Stąd nagłówek HTTP `User-Agent` mógłbyś zapisać tak, jak pokazano na listingu 3.26. Listing 3.27 testuje kod z listingu 3.26.

Listing 3.26. Odpowiedź wartością z nagłówka User-Agent (hello.py)

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.post("/agent")
def get_agent(user_agent:str = Header()):
    return user_agent
```

Listing 3.27. Test nagłówka User-Agent przy użyciu HTTPie

```
$ http -v POST localhost:8000/agent
POST /agent HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 0
Host: localhost:8000
User-Agent: HTTPie/3.2.2



HTTP/1.1 200 OK
content-length: 14
content-type: application/json
date: Wed, 24 Jan 2024 08:21:45 GMT
server: uvicorn

"HTTPie/3.2.2"
```

Różne rodzaje danych w żądaniu

W pojedynczej funkcji ścieżki możesz użyć więcej niż jednej metody przesyłania danych. To znaczy możesz pobrać dane z adresu URL, parametrów żądania, ciała HTTP, nagłówków HTTP, ciasteczek itd. Do przetwarzania danych (autoryzacji, łączenia, podziału na strony) możesz napisać własne funkcje. Przykłady zobaczysz w rozdziale 6. oraz w różnych rozdziałach części trzeciej.

Która metoda jest najlepsza?

Oto kilka rekomendacji:

- przy przekazywaniu argumentów w adresie URL najlepszym podejściem jest trzymanie się wytycznych RESTful;
- parametry żądania służą zazwyczaj do przekazywania argumentów opcjonalnych, takich jak podział na strony;
- ciało żądania przenosi często duże ładunki danych, np. części lub całości modeli.

Kup ksi k Pole ksi k

W każdym przypadku, jeśli w swoich definicjach użyjesz podpowiedzi typów, zostaną one automatycznie sprawdzone pod kątem poprawności typów przez Pydantic. W ten sposób będzie wiadomo, że są one obecne i prawidłowe.

Odpowiedzi HTTP

Cokolwiek zwrócisz ze swojej funkcji ścieżki, FastAPI domyślnie przekształci to na JSON. Odpowiedź HTTP posiada nagłówek `Content-type: application/json`, więc chociaż `greet()` zwraca łańcuch `"Witaj? Świecie?"`, FastAPI przekształca go na JSON. Jest to jedno z domyślnych zachowań FastAPI mających na celu ułatwienie programowania API.

W tym konkretnym przypadku łańcuch w języku Python `"Witaj? Świecie?"` jest przekształcany na łańcuch w JSON-ie: `"Witaj? Świecie?"` — czyli dokładnie na to samo. Jednak wszystko, co zwrócisz, zostanie przekonwertowane na JSON, niezależnie od tego, czy jest to typ w języku Python, czy model Pydantic.

Kod statusu

FastAPI zwraca domyślnie kod statusu żądania `200`, wyjątki powodują kody `4xx`.

Kod HTTP, jaki powinna zwrócić funkcja ścieżki, jeśli wszystko pójdzie dobrze, możesz zapisać w dekoratorze. Ewentualny wyjątek wygeneruje własny kod, który nadpisze wartość domyślną. Dodaj kod z listingu 3.28 do swojego pliku *hello.py* (nie będziemy tutaj za każdym razem wklejać całej zawartości pliku) i przetestuj go tak, jak pokazuje listing 3.29.

Listing 3.28. Specyfikacja kodu statusu HTTP (dodaj do hello.py)

```
@app.get("/happy")
def happy(status_code=200):
    return ":)"
```

Listing 3.29. Test kodu statusu HTTP

```
$ http localhost:8000/happy
HTTP/1.1 200 OK
content-length: 4
content-type: application/json
date: Wed, 24 Jan 2024 08:58:42 GMT
server: uvicorn

":)"
```

Nagłówki

Nagłówki HTTP możesz wstawiać tak, jak pokazuje to listing 3.30 (nie musisz zwracać obiektu `response`).

Listing 3.30. Ustawianie nagłówka HTTP (dodaj do hello.py)

```
from fastapi import Response

@app.get("/header/{name}/{value}")
```


Kup ksi k   Pole ksi k

```
def header(name: str, value: str, response:Response):
    response.headers[name] = value
    return "normalne ciało"
```

Zobaczmy, czy to działa (listing 3.31).

Listing 3.31. Test nagłówków odpowiedzi HTTP

```
$ http localhost:8000/header/marco/polo
HTTP/1.1 200 OK
content-length: 17
content-type: application/json
date: Wed, 24 Jan 2024 09:09:44 GMT
marco: polo
server: uvicorn

"normalne ciało"
```

Rodzaje odpowiedzi

Wśród typów odpowiedzi (ich klasy możesz zaimportować z `fastapi.responses`) znajdują się między innymi:

- `JSONResponse` (domyślna),
- `HTMLResponse`,
- `PlainTextResponse`,
- `RedirectResponse`,
- `FileResponse`,
- `StreamingResponse`.

Więcej o dwóch ostatnich opowiem w rozdziale 15.

Dla pozostałych formatów odpowiedzi (zwanych również *typami MIME*) możesz użyć klasy bazowej `Response`, która wymaga następujących elementów:

`content`
: Łańcuch lub bajty.

`media_type`
: Łańcuch określający typ MIME.

`status_code`
: Wartość całkowita określająca kod odpowiedzi HTTP.

`headers`
: Pythonowa struktura słownika (`dict`) zawierająca łańcuchy.

Kup ksi k Pole ksi k

Konwersja typów

Funkcja ścieżki może zwrócić cokolwiek, a FastAPI domyślnie (używając `JSONResponse`) zamieni to coś na łańcuch w formacie JSON i zwróci z dopasowanymi nagłówkami `Content-Length` i `Content-Type`. Dotyczy to również dowolnego modelu biblioteki Pydantic.

Pewnie jesteś ciekawy, jak to działa. Jeśli używałeś wcześniej biblioteki json w Pythonie, być może widziałeś, że w przypadku pewnych typów danych, takich jak `datetime`, rzuca wyjątek. FastAPI używa wewnętrznej funkcji `jsonable_encoder()` do przekształcania dowolnej struktury danych na „jsonowalną” strukturę danych języka Python, a następnie wywołuje `json.dumps()`, która zwraca łańcuch w formacie JSON. Pokazuje to listing 3.32, który możesz uruchomić za pomocą pytest.

Listing 3.32. Użycie jsonable_encoder() w celu uniknięcia „wylotu” z JSON-a

```
import datetime
import json
import pytest
from fastapi.encoders import jsonable_encoder

@pytest.fixture
def data():
    return datetime.datetime.now()

def test_json_dump(data):
    with pytest.raises(Exception):
        _ = json.dumps(data)

def test_encoder(data):
    out = jsonable_encoder(data)
    assert out
    json_out = json.dumps(out)
    assert json_out
```

Modele i response_model

Istnieje możliwość posiadania różnych klas zawierających te same pola, z wyjątkiem przeznaczonych na dane wprowadzane przez użytkownika, dane wyjściowe i dane wewnętrzne. Oto kilka powodów przemawiających za takim podejściem:

- usunięcie z danych wyjściowych informacji czułych — na przykład danych chronionych przez RODO,
- uzupełnienie danych wprowadzonych przez użytkownika (np. dodanie daty i czasu utworzenia).

Listing 3.33 przedstawia trzy klasy dla naszego wymyślonego przypadku:

- `TagIn` — klasa, która definiuje, jakie dane musi wprowadzić użytkownik (w tym przypadku tylko łańcuch o nazwie `tag`).
- Klasa `Tag` jest skopiowana z `TagIn` i dodaje dwa pola: `created` (kiedy obiekt został utworzony) i `secret` (wewnętrzny łańcuch, który może być przechowywany w bazie danych, ale nigdy nie powinien wydostać się na świat).

Kup książkę Poleć książkę

- `TagOut` jest klasą, która definiuje, co może zostać zwrócone użytkownikowi (przez punkt końcowy wyszukujący dane). Zawiera pole `tag` z pierwotnego obiektu klasy `TagIn` i obiektu klasy `Tag` plus pole `created` wygenerowane dla obiektu klasy `Tag`, ale nie ma pola `secret`.

Listing 3.33. Różne odmiany modeli

```
from datetime import datetime
from pydantic import BaseModel

class TagIn(BaseModel):
    tag: str

class Tag(BaseModel):
    tag: str
    created: datetime
    secret: str

class TagOut(BaseModel):
    tag: str
    created: datetime
```

Oprócz domyślnego typu JSON możesz w FastAPI zwracać z funkcji ścieżek również inne typy danych. Jednym ze sposobów jest użycie argumentu `response_model` w dekoratorze ścieżki do powiedzenia FastAPI, aby zwrócił inny typ danych. FastAPI opuści wszelkie pola, które znajdowały się w zwróconym obiekcie, ale nie zostały wyspecyfikowane w `response_model`.

Na listingu 3.34 udajmy, że napisałeś nowy serwis o nazwie *service/tag.py* z funkcjami `create()` i `get()`, dzięki którym moduł webowy ma co wywołać. Szczegóły tego wywołania nie mają tutaj znaczenia. To, co ma znaczenie, to funkcja ścieżki `get_one()` na samym dole i `response_model=TagOut` w dekoratorze ścieżki. To automatycznie zmienia wewnętrzny obiekt `Tag` na przefiltrowany obiekt `TagOut`.

Listing 3.34. Zwrócenie innego typu odpowiedzi przez response_model

```
import datetime
from fastapi import FastAPI
from model.tag import TagIn, Tag, TagOut
from service import tag as service


app = FastAPI()

@app.post('/')
def create(tag_in: TagIn) -> TagIn:
    tag: Tag = Tag(tag=tag_in.tag, created=datetime.utcnow(),
        secret="shhhh")
    service.create(tag)
    return tag_in

@app.get('/{tag_str}', response_model=TagOut)
def get_one(tag_str: str) -> TagOut:
    tag: Tag = service.get(tag_str)
    return tag
```

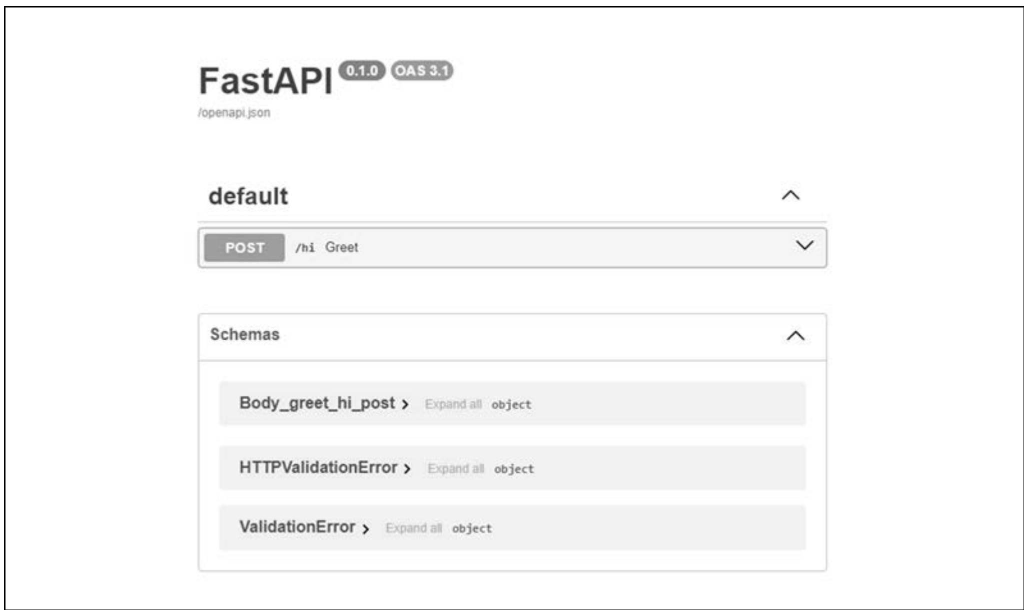
Mimo że zwróciliśmy obiekt klasy `Tag`, `response_model` przekonwertuje go na `TagOut`.

Automatyczna dokumentacja

W tej części zakładam, że wykonujesz aplikację z listingu 3.21 — czyli z wersji, która wysyła parametr `who` w ciele HTTP poprzez żądanie POST do *http://localhost:8000/hi*.

Zamiast tego adresu do przeglądarki wpisz: **`http://localhost:8000/docs`**.

Zobaczysz coś podobnego do zrzutu ekranu z rysunku 3.1 (zrzut został przycięty tak, aby pokazywać tylko to, co istotne).

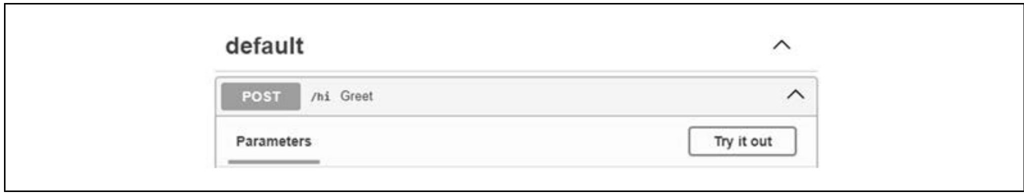


Rysunek 3.1. Wygenerowana strona dokumentacji

Skąd wzięła się ta strona?

FastAPI wygenerował specyfikację OpenAPI na podstawie Twojego kodu i dołączył tę stronę w celu wyświetlenia *i przetestowania* wszystkich Twoich punktów końcowych. Jest to tylko jeden z jego tajnych składników.

Kliknij strzałkę znajdującą się na prawym końcu zielonej ramki, aby otworzyć interfejs użytkownika do testowania (rysunek 3.2).



Rysunek 3.2. Otwarta strona dokumentacji

Kup ksi k Pole ksi k

Kliknij przycisk *Try it out* po prawej stronie. Pojawi się okienko tekstowe pozwalające na wpisanie wartości do ciała żądania (rysunek 3.3).

Rysunek 3.3. Okienko do wpisywania danych

Zaznacz `"string"` i zmień jego wartość na `"Kuzyn Edward"` (pozostaw podwójne cudzysłowy), następnie kliknij niebieski przycisk *Execute* poniżej.

Przyjrzyj się sekcji odpowiedzi poniżej przycisku *Execute* (rysunek 3.4).

Rysunek 3.4. Strona odpowiedzi

Kup ksi k Pole ksi k

Okienko *Response body* pokazuje wartość `Kuzyn Edward`.

Jest to zatem kolejna metoda na przetestowanie Twojej strony (obok przedstawionych już wcześniej metod z użyciem przeglądarki, HTTPie i Requests).

Przy okazji zwróć uwagę na okienko *Curl* w sekcji odpowiedzi — widać tam, jak długie polecenie, w porównaniu do HTTPie, musiałbyś wpisać, aby wykonać taki sam test z wiersza poleceń. To pokazuje, jak pomocne jest automatyczne kodowanie do formatu JSON przez HTTPie.

Automatyczne dokumentowanie jest naprawdę istotne. Wraz z rozwojem Twojej usługi pojawią się setki punktów końcowych — posiadanie zawsze aktualnej dokumentacji i strony testowej będzie bardzo pomocne.

Dane złożone

Powyższy przykład pokazał, jak przekazać pojedynczy łańcuch do punktu końcowego. Wielu punktom końcowym, w szczególności `GET` i `DELETE`, wystarczy tylko kilka prostych wartości łańcuchowych i liczbowych lub w ogóle obejdą się bez argumentów. Sytuacja wygląda inaczej w przypadku tworzenia (`POST`) lub modyfikowania (`PUT` bądź `PATCH`) zasobu — tutaj będziemy potrzebować bardziej złożonych struktur danych. W rozdziale 5. dowiesz się, jak FastAPI korzysta z biblioteki Pydantic i modeli danych do przejrzystej implementacji takich struktur.

Podsumowanie

W tym rozdziale użyliśmy FastAPI do stworzenia strony z pojedynczym punktem końcowym. Przetestowaliśmy tę stronę kilkoma klientami web: przeglądarką, tekstowym programem HTTPie, bibliotekami HTTPX i Requests. Zaczęliśmy od prostego żądania `GET`, w którym argumenty zostały przekazane w adresie URL jako parametry zapytania, a także jako nagłówek. Następnie wysłaliśmy dane w ciele żądania do punktu końcowego `POST`. Kolejnym krokiem było odesłanie różnych typów odpowiedzi HTTP. Na koniec przyjrzeliśmy się automatycznie wygenerowanej stronie z dokumentacją i formularzem, będącym czwartym typem testowego klienta.

Przegląd FastAPI będzie kontynuowany w rozdziale 8.

Kup ksi k   Pole ksi k

Kup ksi k

Pole ksi k

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

Książka zawiera wszechstronne omówienie FastAPI i otaczającego ten framework ekosystemu!

William Jamir Silva, starszy inżynier oprogramowania, Adjust GmbH

Internet rozwija się w niesamowitym tempie. Dawniej sieć WWW była prostsza — projektanci łączyli kod PHP, HTML i zapytania do MySQL w jednym pliku. Z czasem urosła do miliardów stron, co radykalnie zmieniło jej kształt. Zmieniły się też narzędzia i sposób pracy. Dziś idealnym wyborem dewelopera aplikacji WWW jest FastAPI, nowoczesne narzędzie, które wykorzystuje nowe cechy Pythona i z powodzeniem rywalizuje z podobnymi frameworkami języka Golang.

Dzięki znajomości Pythona i temu praktycznemu poradnikowi zaczniesz z sukcesem używać FastAPI i docenisz, jak szybko można budować aplikacje WWW. Zrozumiesz zasady pracy z tym frameworkiem i będziesz je stosować przy tworzeniu własnych projektów. Przyswoisz różne nieznane powszechnie techniki i dowiesz się, jakie praktyki najlepiej sprawdzają się w codziennej pracy. Poznasz takie zagadnienia jak formularze, dostęp do baz danych, grafika i mapy. Nauczysz się również korzystać z interfejsów RESTful API, prowadzić walidację danych, autoryzację i zapewniać wysoką wydajność swojego kodu.

FastAPI przedstawione w prosty sposób! Książka wyposaża w praktyczną wiedzę i umożliwia szybki start.

Ganesh Harke, starszy inżynier oprogramowania, Citibank

W książce:

- budowa aplikacji WWW z użyciem FastAPI
- różnice pomiędzy FastAPI, Starlette i Pydantic
- stosowanie funkcji asynchronicznych, sprawdzanie typów danych i walidacja
- nowe cechy Pythona 3.8+, w tym adnotacje typów
- tworzenie kodu synchronicznego i asynchronicznego
- korzystanie z zewnętrznych API i usług

Bill Lubanovic ma ponad czterdziestoletnie doświadczenie w tworzeniu oprogramowania. Specjalizuje się w Linuksie, Pythonie, budowaniu stron internetowych i aplikacji webowych. Jest autorem książki *Python. Nowoczesne programowanie w prostych krokach.* Od kilku lat używa FastAPI. Mieszka z rodziną w górach Sangre de Sasquatch w stanie Minnesota.

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-1296-0

9 788328 912960

Cena: 69,00 zł