



FastAPI to szybki, prosty w nauce i gotowy do użycia w produkcji framework



Dokumentacja: <https://fastapi.tiangolo.com>

Kod źródłowy: <https://github.com/tiangolo/fastapi>



FastAPI to nowoczesny, wydajny framework webowy do budowania API z użyciem Pythona 3.6+ bazujący na standardowym typowaniu Pythona.

Kluczowe cechy:

- **Wydajność:** FastAPI jest bardzo wydajny, na równi z **NodeJS** oraz **Go** (dzięki Starlette i Pydantic). [Jeden z najszybszych dostępnych frameworków Pythonowych](#).
- **Szybkość kodowania:** Przyspiesza szybkość pisania nowych funkcjonalności o około 200% do 300%. *
- **Mniejsza ilość błędów:** Zmniejsza ilość ludzkich (dewelopera) błędów o około 40%. *
- **Intuicyjność:** Wspaniałe wsparcie dla edytorów kodu. Dostępne wszędzie automatyczne uzupełnianie kodu. Krótszy czas debugowania.
- **Łatwość:** Zaprojektowany by być prosty i łatwy do nauczenia. Mniej czasu spędzonego na czytanie dokumentacji.
- **Kompaktość:** Minimalizacja powtarzającego się kodu. Wiele funkcjonalności dla każdej deklaracji parametru. Mniej błędów.
- **Solidność:** Kod gotowy dla środowiska produkcyjnego. Wraz z automatyczną interaktywną dokumentacją.
- **Bazujący na standardach:** Oparty na (i w pełni kompatybilny z) otwartych standardach API: [OpenAPI](#) (wcześniej znane jako Swagger) oraz [JSON Schema](#).

* oszacowania bazowane na testach wykonanych przez wewnętrzny zespół deweloperów, budujących aplikacje używane na środowisku produkcyjnym.

Sponsorzy

{% if sponsors %} {% for sponsor in sponsors.gold -%}  {% endfor -%} {% for sponsor in sponsors.silver -%}  {% endfor %} {% endif %}

[Inni sponsorzy](#)

Opinie

"[...] I'm using **FastAPI** a ton these days. [...] I'm actually planning to use it for all of my team's **ML services at Microsoft**. Some of them are getting integrated into the core **Windows** product and some **Office** products."

Kabir Khan - **Microsoft** ([ref](#))

"We adopted the **FastAPI** library to spawn a **REST** server that can be queried to obtain **predictions**. [for Ludwig]"

Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala - **Uber** ([ref](#))

"**Netflix** is pleased to announce the open-source release of our **crisis management** orchestration framework: **Dispatch!** [built with **FastAPI**]"

Kevin Glisson, Marc Vilanova, Forest Monsen - **Netflix** ([ref](#))

"I'm over the moon excited about **FastAPI**. It's so fun!"

Brian Okken - **Python Bytes** podcast host ([ref](#))

"Honestly, what you've built looks super solid and polished. In many ways, it's what I wanted **Hug** to be - it's really inspiring to see someone build that."

Timothy Crosley - **Hug** creator ([ref](#))

"If you're looking to learn one **modern framework** for building REST APIs, check out **FastAPI** [...] It's fast, easy to use and easy to learn [...]"

"We've switched over to **FastAPI** for our **APIs** [...] I think you'll like it [...]"

Ines Montani - Matthew Honnibal - **Explosion AI** founders - **spaCy** creators ([ref](#)) - ([ref](#))

Typer, FastAPI aplikacji konsolowych

Typer

Jeżeli tworzysz aplikację `CLI`, która ma być używana w terminalu zamiast API, sprawdź [Typer](#).

Typer to młodsze rodzeństwo FastAPI. Jego celem jest pozostanie **FastAPI aplikacji konsolowych** .  

Wymagania

Python 3.6+

FastAPI oparty jest na:

- [Starlette](#) dla części webowej.
- [Pydantic](#) dla części obsługujących dane.

Instalacja

```
$ pip install fastapi
```

```
---> 100%
```

Na serwerze produkcyjnym będziesz także potrzebował serwera ASGI, np. [Uvicorn](#) lub [Hypercorn](#).

```
$ pip install uvicorn[standard]
```

```
---> 100%
```

Przykład

Stwórz

- Utwórz plik o nazwie `main.py` z:

```
from typing import Optional

from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

- Albo użyj `async def...`

Uruchom

Uruchom serwer używając:

```
$ uvicorn main:app --reload
```

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [28720]
INFO:      Started server process [28722]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

- O komendzie `uvicorn main:app --reload...`

Wypróbuj

Otwórz link <http://127.0.0.1:8000/items/5?q=somequery> w przeglądarce.

Zobaczysz następującą odpowiedź JSON:

```
{"item_id": 5, "q": "somequery"}
```

Właśnie stworzyłeś API które:

- Otrzymuje żądania HTTP w *ścieżce* `/ i /items/{item_id}` .
- Obie *ścieżki* używają *operacji* `GET` (znane także jako *metody* HTTP).
- *Ścieżka* `/items/{item_id}` ma *parametr ścieżki* `item_id` który powinien być obiektem typu `int` .
- *Ścieżka* `/items/{item_id}` ma opcjonalny *parametr zapytania* typu `str` o nazwie `q` .

Interaktywna dokumentacja API

Otwórz teraz stronę <http://127.0.0.1:8000/docs>.

Zobaczysz automatyczną interaktywną dokumentację API (dostarczoną z pomocą [Swagger UI](#)):

Fast API - Swagger UI x +

127.0.0.1:8000/docs

Fast API 0.1.0 OAS3

/openapi.json

default

GET /items/{item_id} Read Item Get

Parameters Try it out

Name	Description
item_id * required integer (path)	
q string (query)	

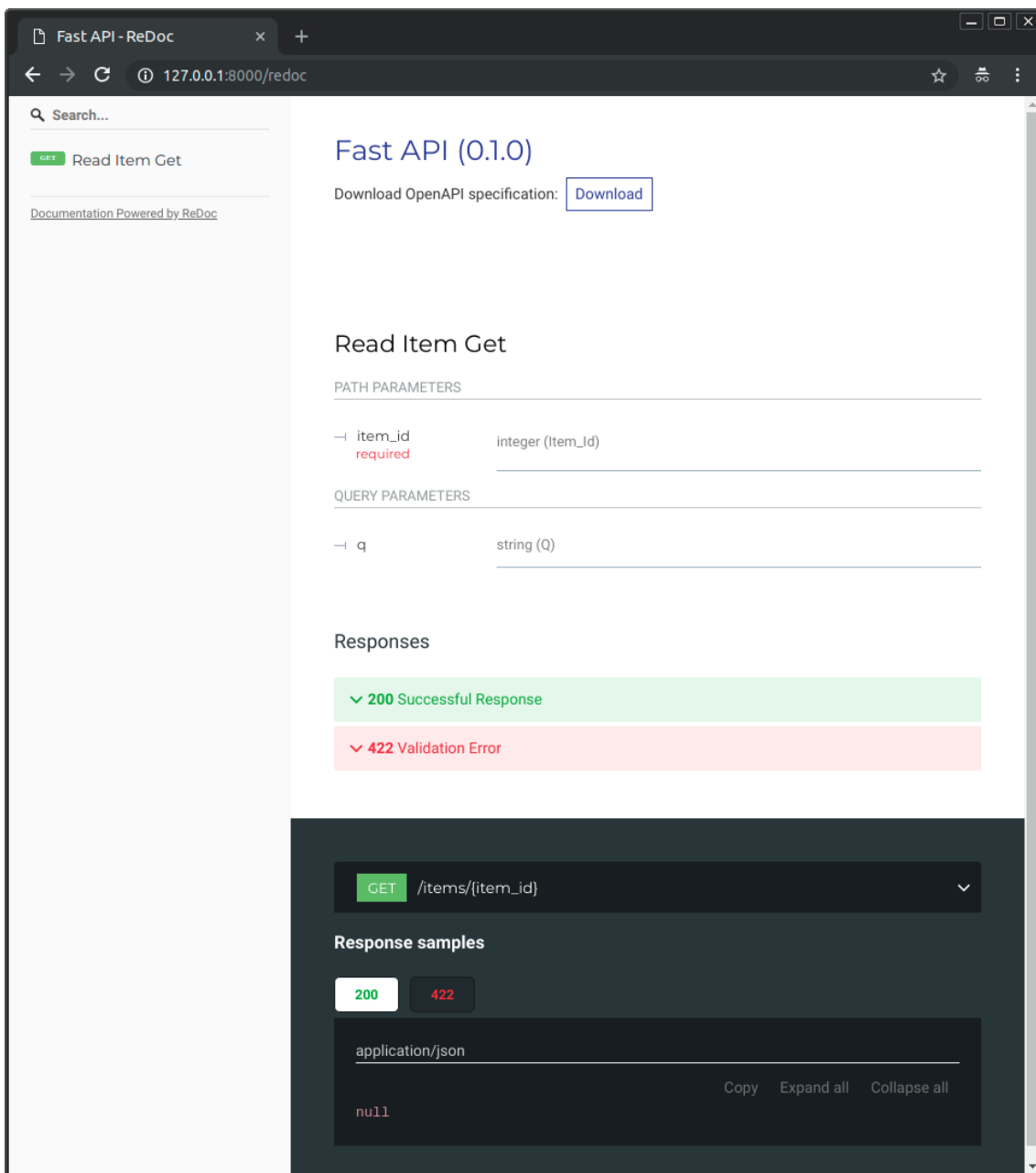
Responses

Code	Description	Links
200	Successful Response application/json Controls Accept header.	No links
422	Validation Error application/json Example Value Schema <pre>{ "detail": [{ "loc": ["string"] }]}</pre>	No links

Alternatywna dokumentacja API

Otwórz teraz <http://127.0.0.1:8000/redoc>.

Zobaczysz alternatywną, lecz wciąż automatyczną dokumentację (wygenerowaną z pomocą [ReDoc](#)):



Aktualizacja przykładu

Zmodyfikuj teraz plik `main.py`, aby otrzymywał treść (body) żądania `PUT`.

Zadeklaruj treść żądania, używając standardowych typów w Pythonie dzięki Pydantic.

```
from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()
```

```
class Item(BaseModel):
    name: str
    price: float
    is_offer: Optional[bool] = None

@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}

@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {"item_name": item.name, "item_id": item_id}
```

Serwer powinien przeładować się automatycznie (ponieważ dodałeś `--reload` do komendy `uvicorn` powyżej).

Zaktualizowana interaktywna dokumentacja API

Wejdź teraz na <http://127.0.0.1:8000/docs>.

- Interaktywna dokumentacja API zaktualizuje się automatycznie, także z nową treścią żądania (body):

Fast API - Swagger UI

127.0.0.1:8000/docs

Fast API 0.1.0 OAS3

/openapi.json

default

GET / Read Root Get

GET /items/{item_id} Read Item Get

PUT /items/{item_id} Save Item Put

Parameters Try it out

Name	Description
item_id * required	
integer	
(path)	

Request body required application/json

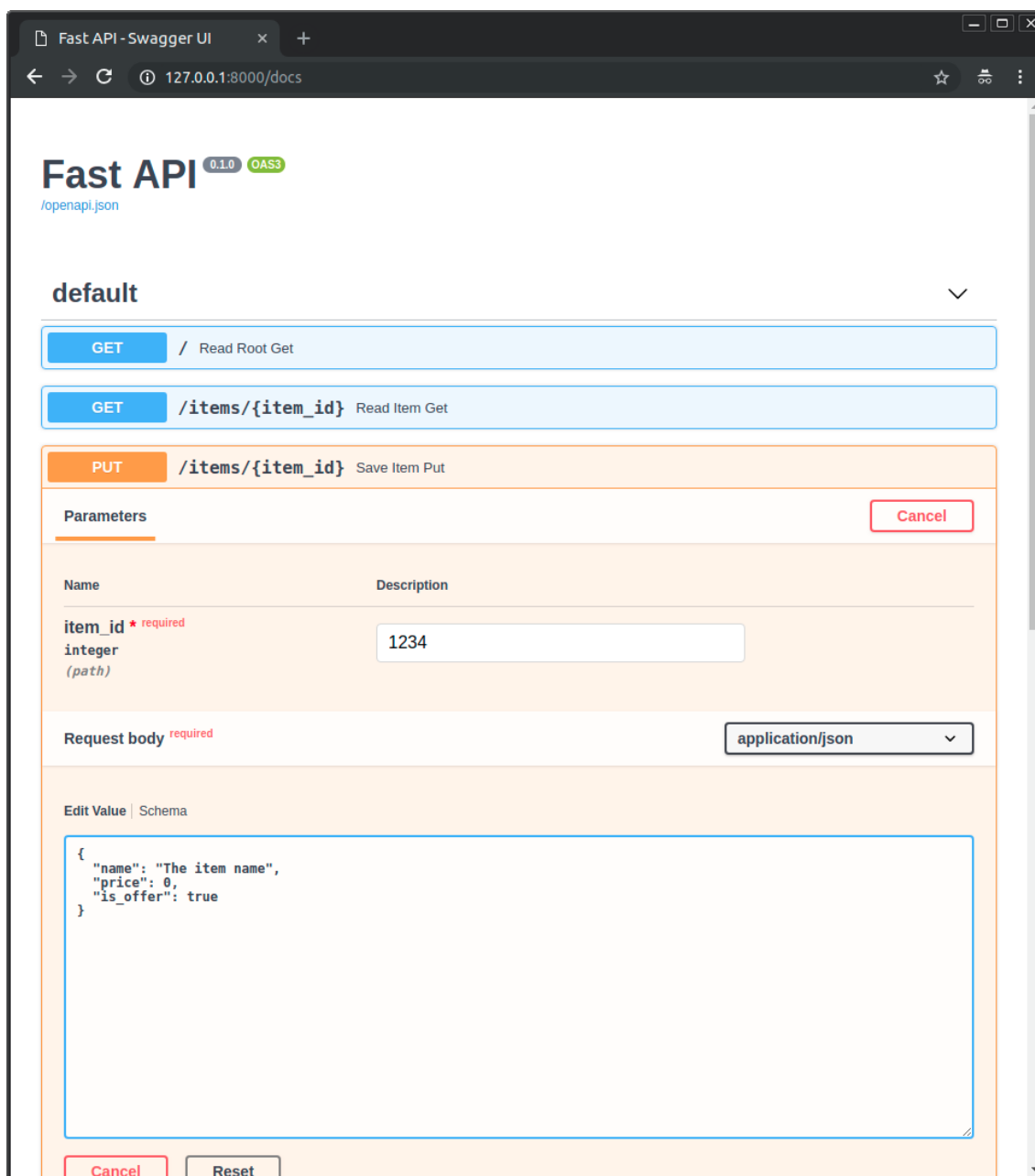
Example Value | Schema

```
{
  "name": "string",
  "price": 0,
  "is_offer": true
}
```

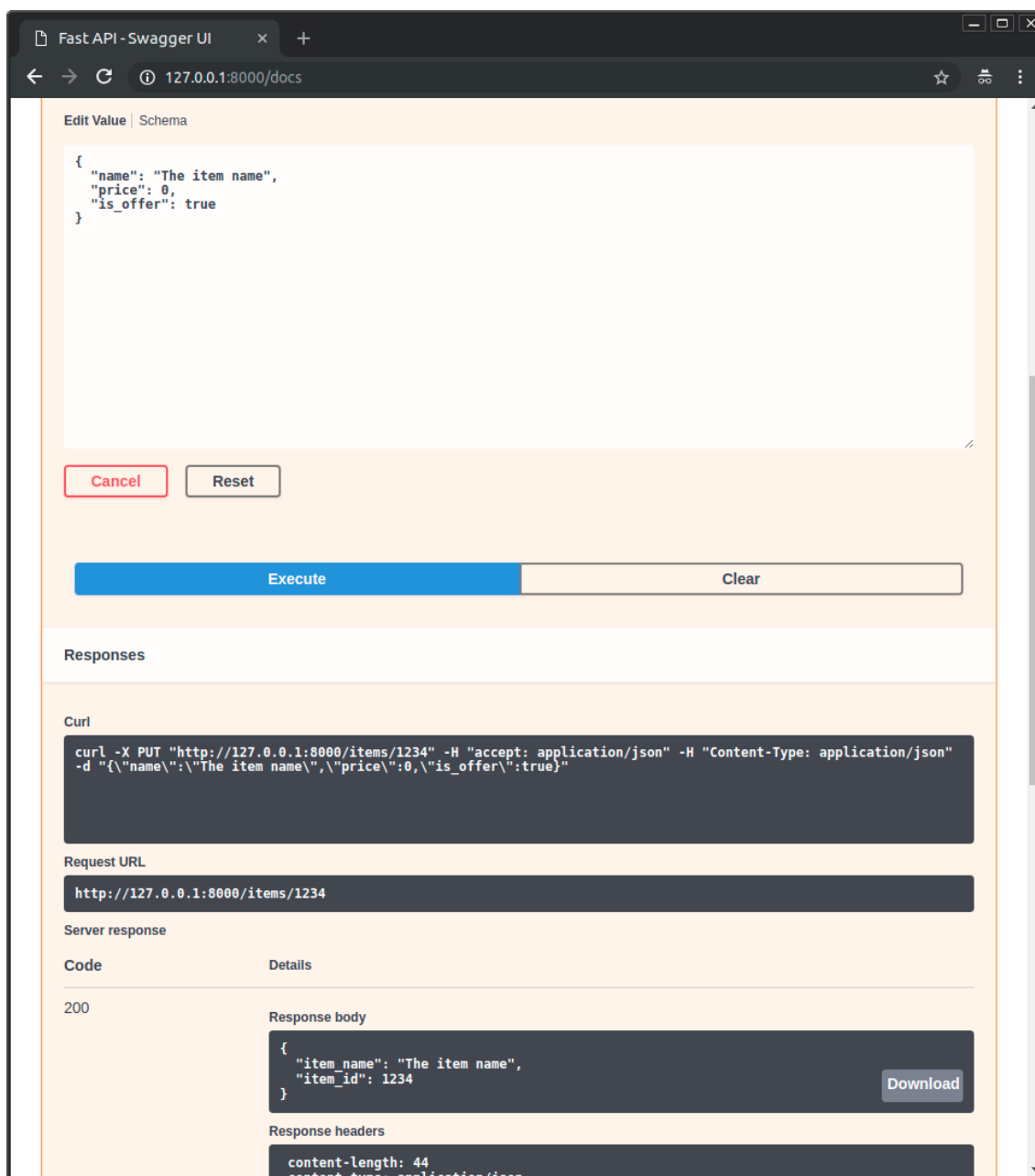
Responses

Code	Description	Links
200	Successful Response	No links

- Kliknij przycisk "Try it out" (wypróbuj), pozwoli Ci to wypełnić parametry i bezpośrednio użyć API:



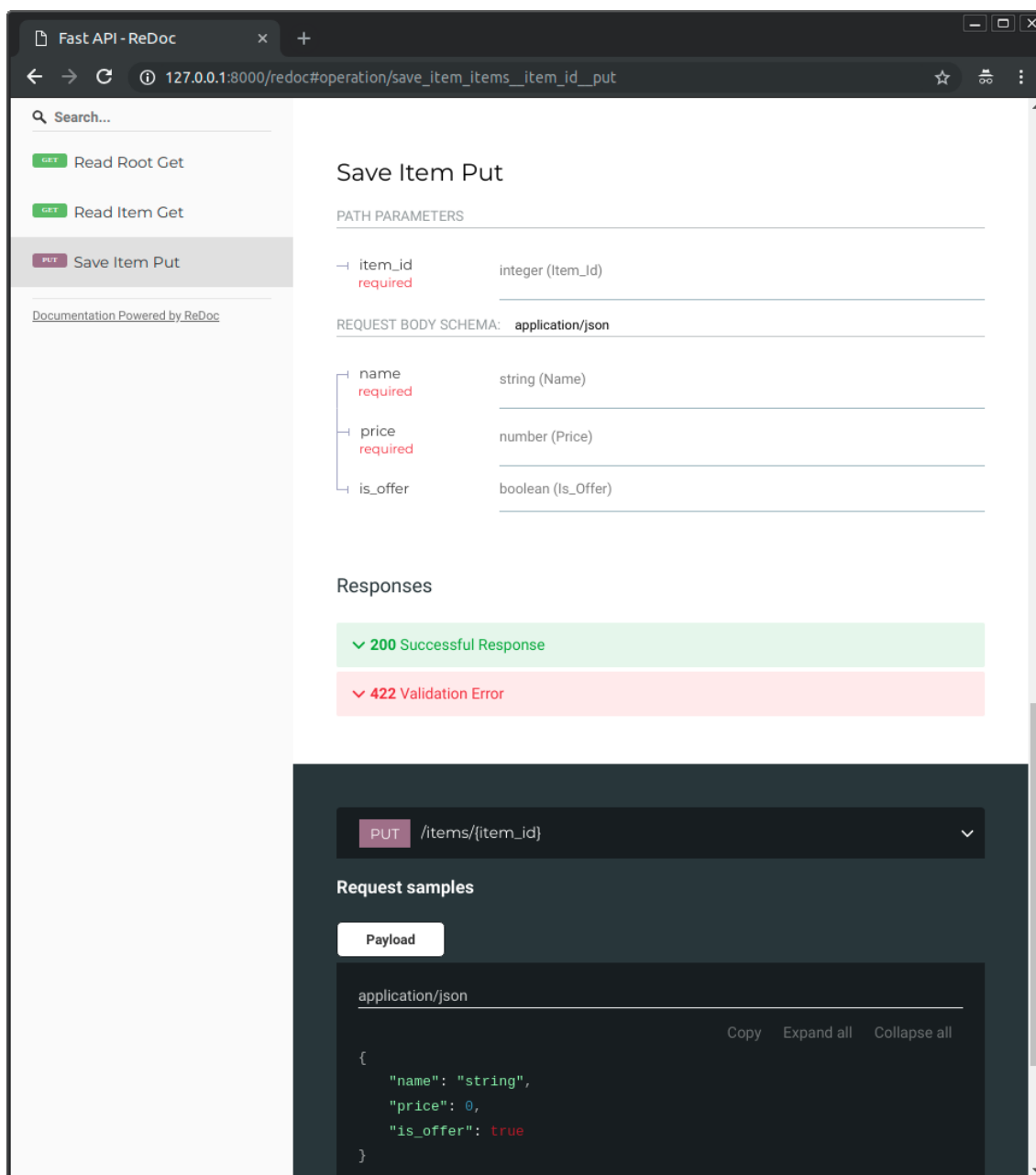
- Kliknij potem przycisk "Execute" (wykonaj), interfejs użytkownika połączy się z API, wyśle parametry, otrzyma odpowiedź i wyświetli ją na ekranie:



Zaktualizowana alternatywna dokumentacja API

Otwórz teraz <http://127.0.0.1:8000/redoc>.

- Alternatywna dokumentacja również pokaże zaktualizowane parametry i treść żądania (body):



Podsumowanie

Podsumowując, musiałeś zadeklarować typy parametrów, treści żądania (body) itp. tylko **raz**, i są one dostępne jako parametry funkcji.

Robisz to tak samo jak ze standardowymi typami w Pythonie.

Nie musisz się uczyć żadnej nowej składni, metod lub klas ze specyficznych bibliotek itp.

Po prostu standardowy **Python 3.6+**.

Na przykład, dla danych typu `int` :

```
item_id: int
```

albo dla bardziej złożonego obiektu `Item` :

```
item: Item
```

...i z pojedynczą deklaracją otrzymujesz:

- Wsparcie edytorów kodu, wliczając:
 - Auto-upełnianie.
 - Sprawdzanie typów.
- Walidacja danych:
 - Automatyczne i przejrzyste błędy gdy dane są niepoprawne.
 - Walidacja nawet dla głęboko zagnieżdżonych obiektów JSON.
- Konwersja danych wejściowych: przychodzących z sieci na Pythonowe typy. Pozwala na przetwarzanie danych:
 - JSON.
 - Parametrów ścieżki.
 - Parametrów zapytania.
 - Dane cookies.
 - Dane nagłówków (headers).
 - Formularze.
 - Pliki.
- Konwersja danych wyjściowych: wychodzących z Pythona do sieci (jako JSON):
 - Przetwarzanie Pythonowych typów (`str` , `int` , `float` , `bool` , `list` , itp).
 - Obiekty `datetime` .
 - Obiekty `UUID` .
 - Modele baz danych.
 - ...i wiele więcej.
- Automatyczne interaktywne dokumentacje API, wliczając 2 alternatywne interfejsy użytkownika:
 - Swagger UI.
 - ReDoc.

Wracając do poprzedniego przykładu, **FastAPI** :

- Potwierdzi, że w ścieżce jest `item_id` dla żądań `GET` i `PUT` .
- Potwierdzi, że `item_id` jest typu `int` dla żądań `GET` i `PUT` .
 - Jeżeli nie jest, odbiorca zobaczy przydatną, przejrzystą wiadomość z błędem.
- Sprawdzi czy w ścieżce jest opcjonalny parametr zapytania `q` (np. `http://127.0.0.1:8000/items/foo?q=somequery`) dla żądania `GET` .
 - Jako że parametr `q` jest zadeklarowany jako `= None` , jest on opcjonalny.
 - Gdyby tego `None` nie było, parametr ten byłby wymagany (tak jak treść żądania w żądaniu `PUT`).
- Dla żądania `PUT` z ścieżką `/items/{item_id}` , odczyta treść żądania jako JSON:
 - Sprawdzi czy posiada wymagany atrybut `name` , który powinien być typu `str` .
 - Sprawdzi czy posiada wymagany atrybut `price` , który musi być typu `float` .

- Sprawdzi czy posiada opcjonalny atrybut `is_offer`, który (jeżeli obecny) powinien być typu `bool`.
- To wszystko będzie również działać dla głęboko zagnieżdżonych obiektów JSON.
- Automatycznie konwertuje z i do JSON.
- Dokumentuje wszystko w OpenAPI, które może być używane przez:
 - Interaktywne systemy dokumentacji.
 - Systemy automatycznego generowania kodu klienckiego, dla wielu języków.
- Dostarczy bezpośrednio 2 interaktywne dokumentacje webowe.

To dopiero początek, ale już masz mniej-więcej pojęcie jak to wszystko działa.

Spróbuj zmienić linijkę:

```
return {"item_name": item.name, "item_id": item_id}
```

...Z:

```
... "item_name": item.name ...
```

...na:

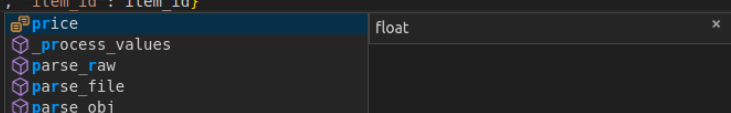
```
... "item_price": item.price ...
```

...i zobacz jak edytor kodu automatycznie uzupełni atrybuty i będzie znał ich typy:

```

1  from fastapi import FastAPI
2  from pydantic import BaseModel
3
4  app = FastAPI()
5
6
7  class Item(BaseModel):
8      name: str
9      price: float
10     is_offer: bool = None
11
12
13 @app.get("/")
14 def read_root():
15     return {"Hello": "World"}
16
17
18 @app.get("/items/{item_id}")
19 def read_item(item_id: int, q: str = None):
20     return {"item_id": item_id, "q": q}
21
22
23 @app.put("/items/{item_id}")
24 def save_item(item_id: int, item: Item):
25     return {"item_name": item.pr, "item_id": item_id}
26

```



Dla bardziej kompletnych przykładów posiadających więcej funkcjonalności, zobacz [Tutorial - User Guide](#).

Uwaga Spoiler: tutorial - user guide zawiera:

- Deklaracje **parametrów** z innych miejsc takich jak: **nagłówki**, **pliki cookies**, **formularze** i **pliki**.
- Jak ustawić **ograniczenia walidacyjne** takie jak `maksymalna długość` lub `regex`.
- Potężny i łatwy w użyciu system **Dependency Injection**.
- Zabezpieczenia i autentykacja, wliczając wsparcie dla **OAuth2** z **tokenami JWT** oraz autoryzacją **HTTP Basic**.
- Bardziej zaawansowane (ale równie proste) techniki deklarowania **głęboko zagnieżdżonych modeli JSON** (dzięki Pydantic).
- Wiele dodatkowych funkcji (dzięki Starlette) takie jak:
 - **WebSockets**
 - **GraphQL**
 - bardzo proste testy bazujące na `requests` oraz `pytest`
 - **CORS**
 - **Sesje cookie**
 - ...i więcej.

Wydajność

Niezależne benchmarki TechEmpower pokazują, że **FastAPI** (uruchomiony na serwerze Uvicorn) [jest jednym z najszybszych dostępnych Pythonowych frameworków](#), zaraz po Starlette i Uvicorn (używany wewnątrz przez FastAPI). (*)

Aby dowiedzieć się o tym więcej, zobacz sekcję [Benchmarks](#).

Opcjonalne zależności

Używane przez Pydantic:

- [ujson](#) - dla szybszego "parsowania" danych JSON.
- [email_validator](#) - dla walidacji adresów email.

Używane przez Starlette:

- [requests](#) - Wymagane jeżeli chcesz korzystać z `TestClient`.
- [aiofiles](#) - Wymagane jeżeli chcesz korzystać z `FileResponse` albo `StaticFiles`.
- [jinja2](#) - Wymagane jeżeli chcesz używać domyślnej konfiguracji szablonów.
- [python-multipart](#) - Wymagane jeżeli chcesz wsparcie "parsowania" formularzy, używając `request.form()`.
- [itsdangerous](#) - Wymagany dla wsparcia `SessionMiddleware`.
- [pyyaml](#) - Wymagane dla wsparcia `SchemaGenerator` z Starlette (z FastAPI prawdopodobnie tego nie potrzebujesz).
- [graphene](#) - Wymagane dla wsparcia `GraphQLApp`.
- [ujson](#) - Wymagane jeżeli chcesz korzystać z `UJSONResponse`.

Używane przez FastAPI / Starlette:

- [uvicorn](#) - jako serwer, który ładuje i obsługuje Twoją aplikację.
- [orjson](#) - Wymagane jeżeli chcesz używać `ORJSONResponse`.

Możesz zainstalować wszystkie te aplikacje przy pomocy `pip install fastapi[all]`.

Licencja

Ten projekt jest na licencji MIT.