



FastAPI 框架, 高性能, 易于学习, 高效编码, 生产可用

 Test **passing**  coverage **100%**  pypi package **v0.81.0**

文档: <https://fastapi.tiangolo.com>

源码: <https://github.com/tiangolo/fastapi>



FastAPI 是一个用于构建 API 的现代、快速（高性能）的 web 框架，使用 Python 3.6+ 并基于标准的 Python 类型提示。

关键特性:

- **快速**: 可与 **NodeJS** 和 **Go** 比肩的极高性能（归功于 Starlette 和 Pydantic）。[最快的 Python web 框架之一](#)。
- **高效编码**: 提高功能开发速度约 200% 至 300%。*
- **更少 bug**: 减少约 40% 的人为（开发者）导致错误。*
- **智能**: 极佳的编辑器支持。处处皆可自动补全，减少调试时间。
- **简单**: 设计的易于使用和学习，阅读文档的时间更短。
- **简短**: 使代码重复最小化。通过不同的参数声明实现丰富功能。bug 更少。
- **健壮**: 生产可用级别的代码。还有自动生成的交互式文档。
- **标准化**: 基于（并完全兼容）API 的相关开放标准: [OpenAPI](#) (以前被称为 Swagger) 和 [JSON Schema](#)。

* 根据对某个构建线上应用的内部开发团队所进行的测试估算得出。

Sponsors

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

[Other sponsors](#)

评价

「[...] 最近我一直在使用 **FastAPI**。[...] 实际上我正在计划将其用于我所在的**微软**团队的所有**机器学习服务**。其中一些服务正被集成进核心 **Windows** 产品和一些 **Office** 产品。」

Kabir Khan - **微软** ([ref](#))

「我们选择了 **FastAPI** 来创建用于获取**预测结果**的 **REST** 服务。[用于 Ludwig]」

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

「**Netflix** 非常高兴地宣布，正式开源我们的**危机管理**编排框架：**Dispatch**！[使用 **FastAPI** 构建]」

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

「**FastAPI** 让我兴奋的欣喜若狂。它太棒了！」

Brian Okken - **Python Bytes** 播客主持人 ([ref](#))

「老实说，你的作品看起来非常可靠和优美。在很多方面，这就是我想让 **Hug** 成为的样子 - 看到有人实现了它真的很鼓舞人心。」

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

「如果你正打算学习一个**现代框架**用来构建 REST API，来看下 **FastAPI** [...] 它快速、易用且易于学习 [...]」



「我们已经将 **API** 服务切换到了 **FastAPI** [...] 我认为你会喜欢它的 [...]」

Ines Montani - Matthew Honnibal - **Explosion AI** 创始人 - **spaCy** 作者 ([ref](#)) - ([ref](#))

Typer，命令行中的 FastAPI

Typer

如果你正在开发一个在终端中运行的**命令行应用**而不是 web API，不妨试下 **Typer**。

Typer 是 FastAPI 的小同胞。它想要成为**命令行中的 FastAPI**。  

依赖

Python 3.6 及更高版本

FastAPI 站在以下巨人的肩膀之上：

- [Starlette](#) 负责 web 部分。
- [Pydantic](#) 负责数据部分。

安装

```
$ pip install fastapi
```

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

你还会需要一个 ASGI 服务器，生产环境可以使用 [Uvicorn](#) 或者 [Hypercorn](#)。

```
$ pip install uvicorn[standard]

---> 100%
```

示例

创建

- 创建一个 `main.py` 文件并写入以下内容:

```
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}
```

► 或者使用 `async def...`

运行

通过以下命令运行服务器:

```
$ 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.
```

► 关于 `uvicorn main:app --reload` 命令.....

检查

使用浏览器访问 <http://127.0.0.1:8000/items/5?q=somequery>。

你将会看到如下 JSON 响应:

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

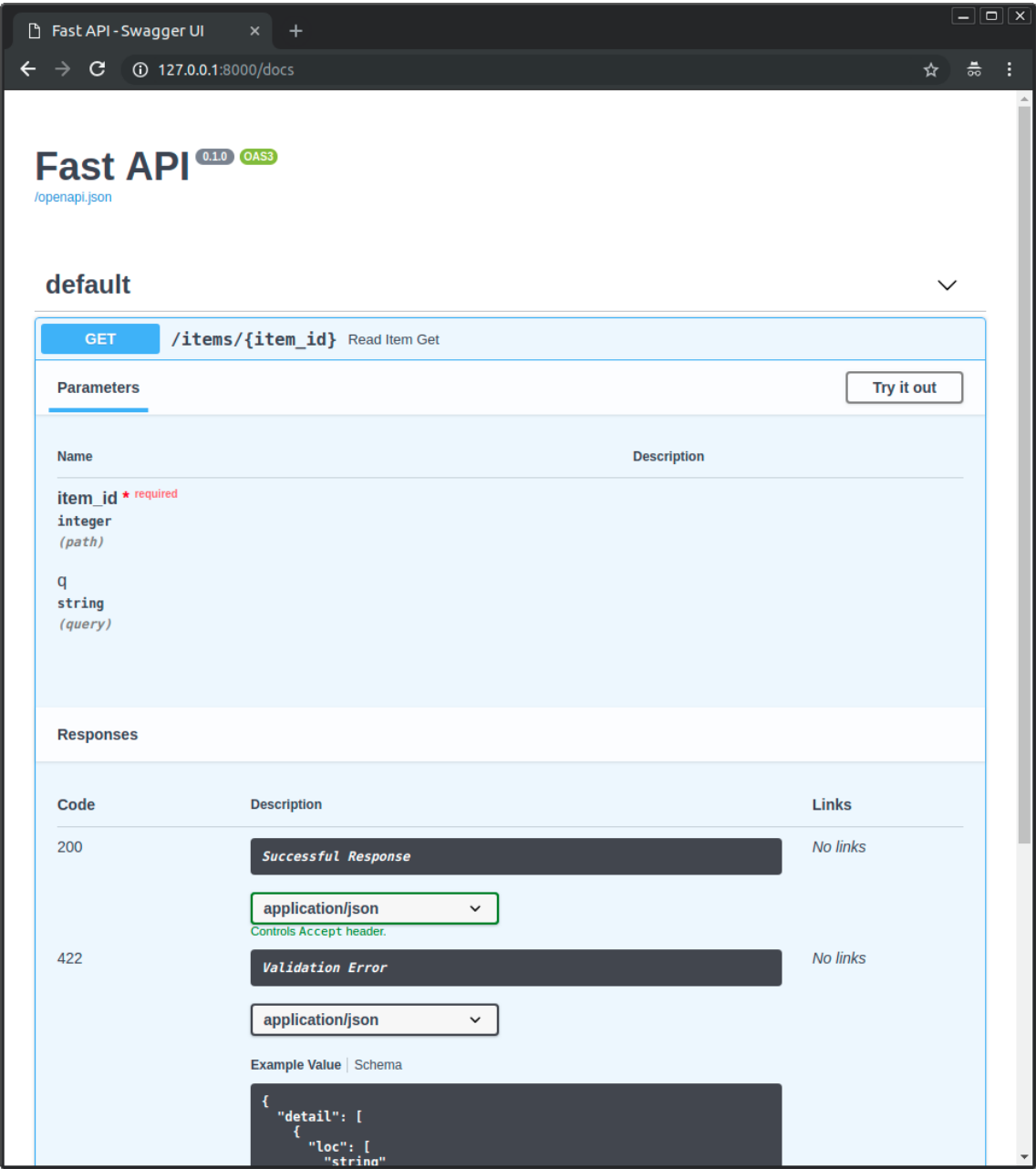
你已经创建了一个具有以下功能的 API：

- 通过 路径 `/` 和 `/items/{item_id}` 接受 HTTP 请求。
- 以上 路径 都接受 `GET` 操作（也被称为 HTTP 方法）。
- `/items/{item_id}` 路径 有一个 路径参数 `item_id` 并且应该为 `int` 类型。
- `/items/{item_id}` 路径 有一个可选的 `str` 类型的 查询参数 `q`。

交互式 API 文档

现在访问 <http://127.0.0.1:8000/docs>。

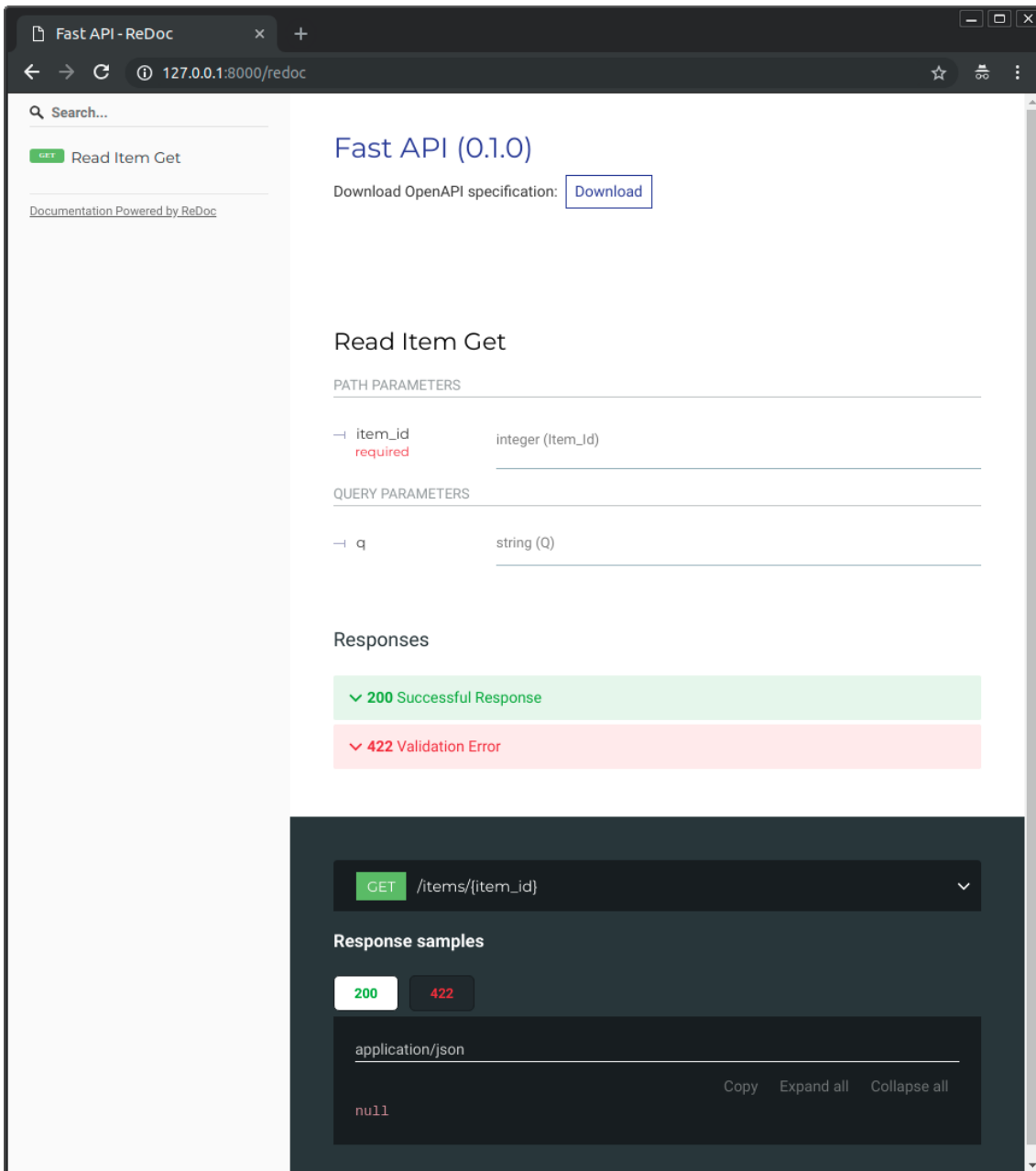
你会看到自动生成的交互式 API 文档（由 [Swagger UI](#)生成）：



可选的 API 文档

访问 <http://127.0.0.1:8000/redoc>。

你会看到另一个自动生成的文档（由 [ReDoc](#) 生成）：



示例升级

现在修改 `main.py` 文件来从 `PUT` 请求中接收请求体。

我们借助 Pydantic 来使用标准的 Python 类型声明请求体。

```
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}

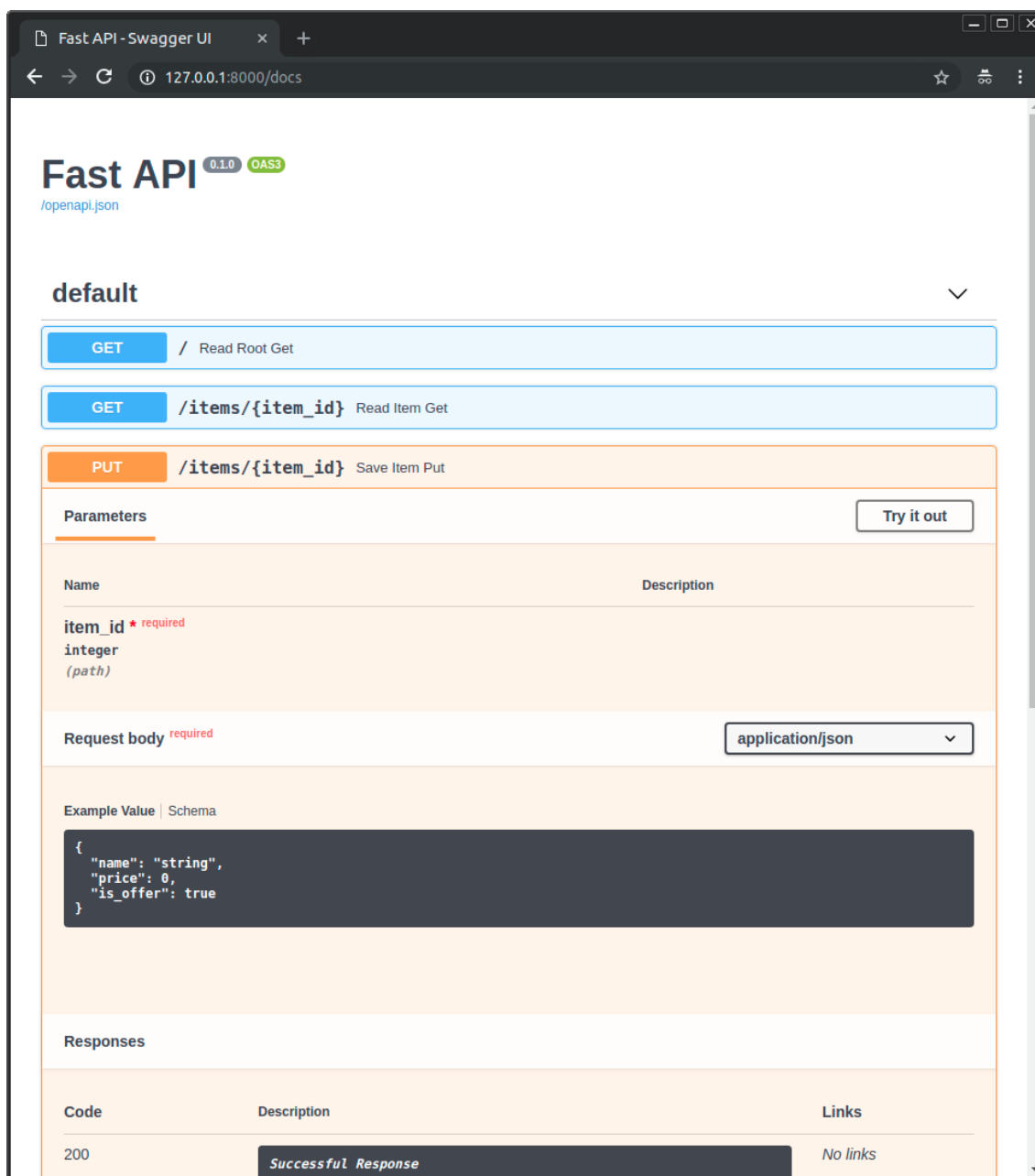

@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {"item_name": item.name, "item_id": item_id}
```

服务器将会自动重载（因为在上面的步骤中你向 `uvicorn` 命令添加了 `--reload` 选项）。

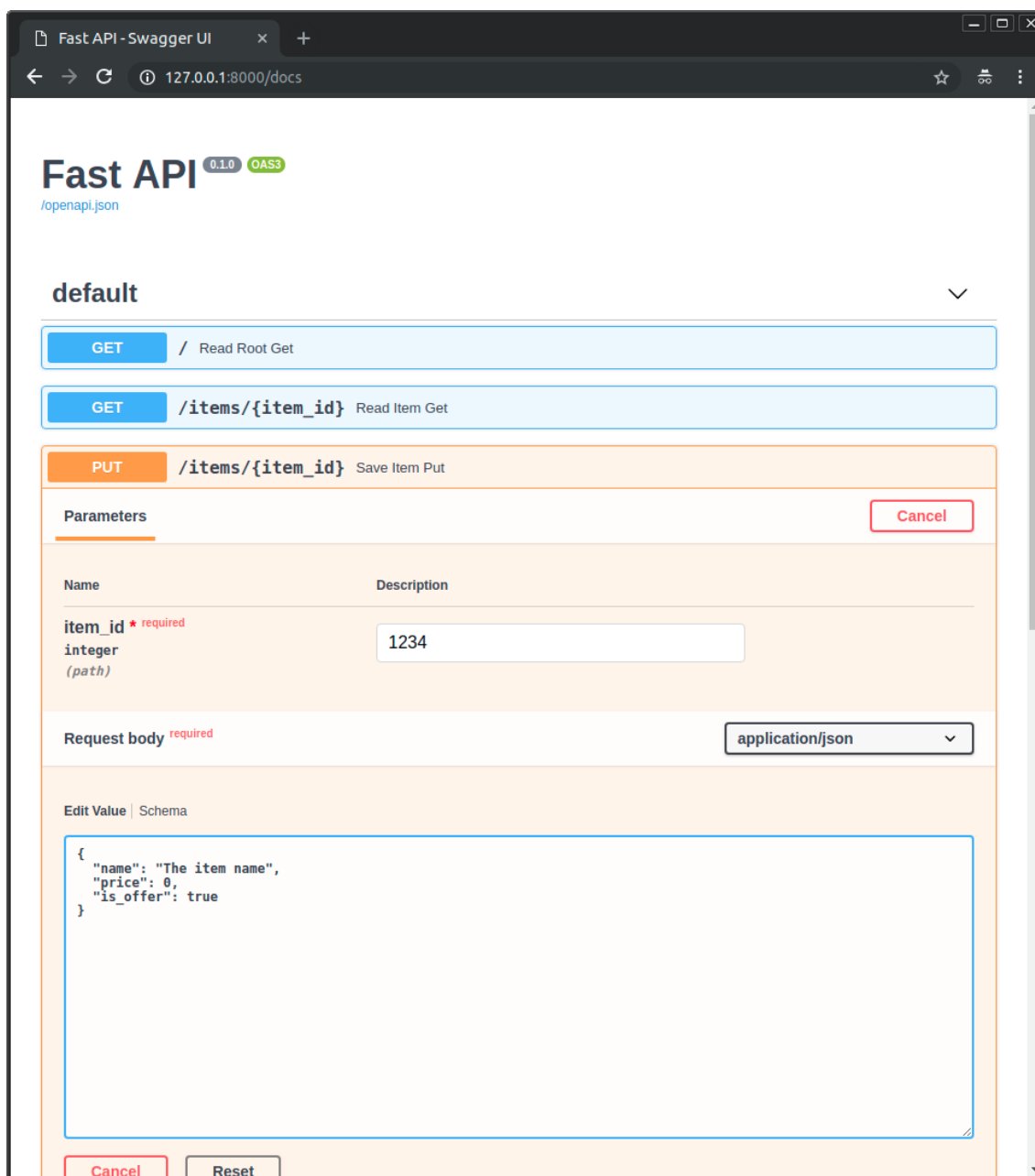
交互式 API 文档升级

访问 <http://127.0.0.1:8000/docs>。

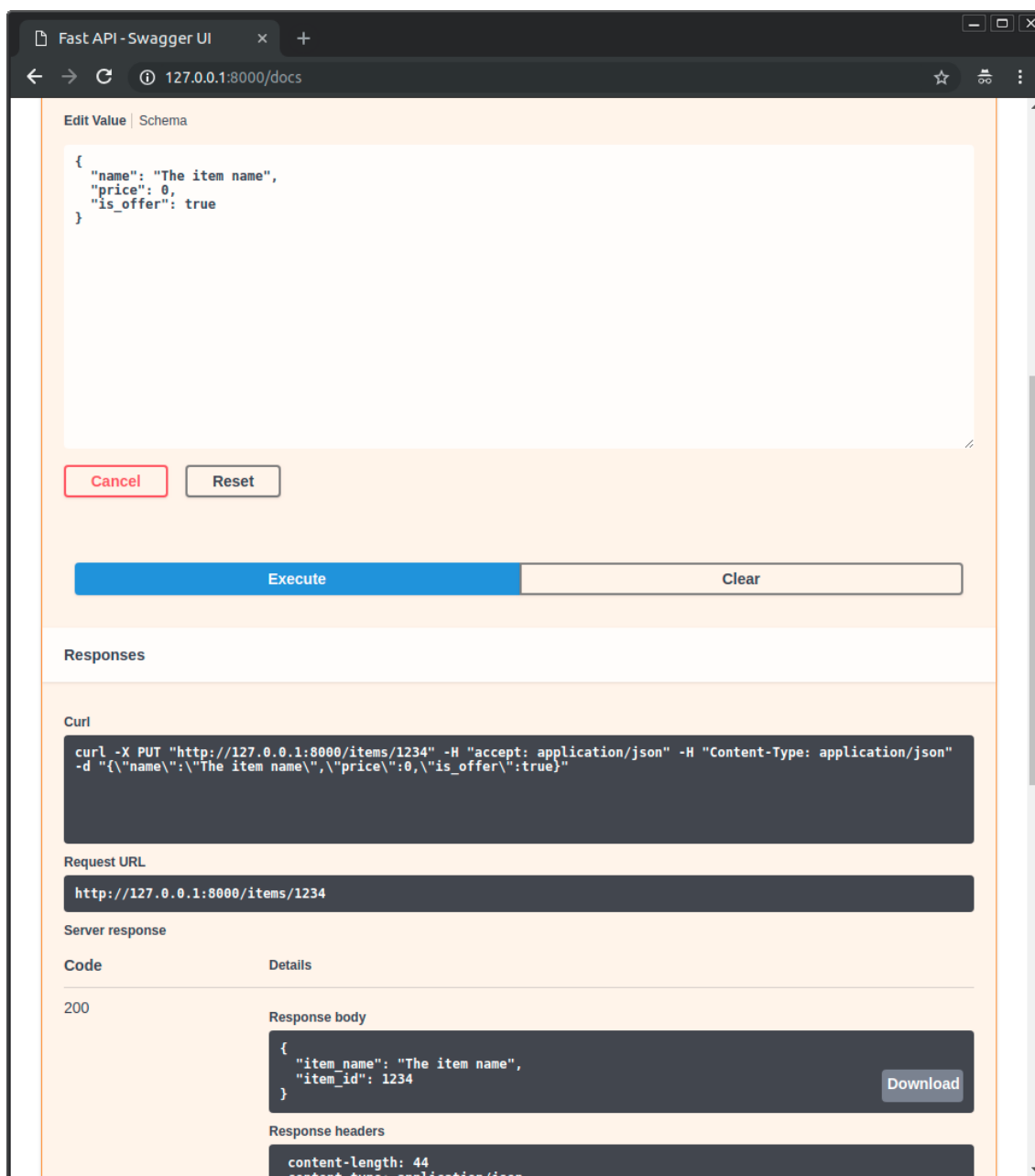
- 交互式 API 文档将会自动更新，并加入新的请求体：



- 点击「Try it out」按钮，之后你可以填写参数并直接调用 API：



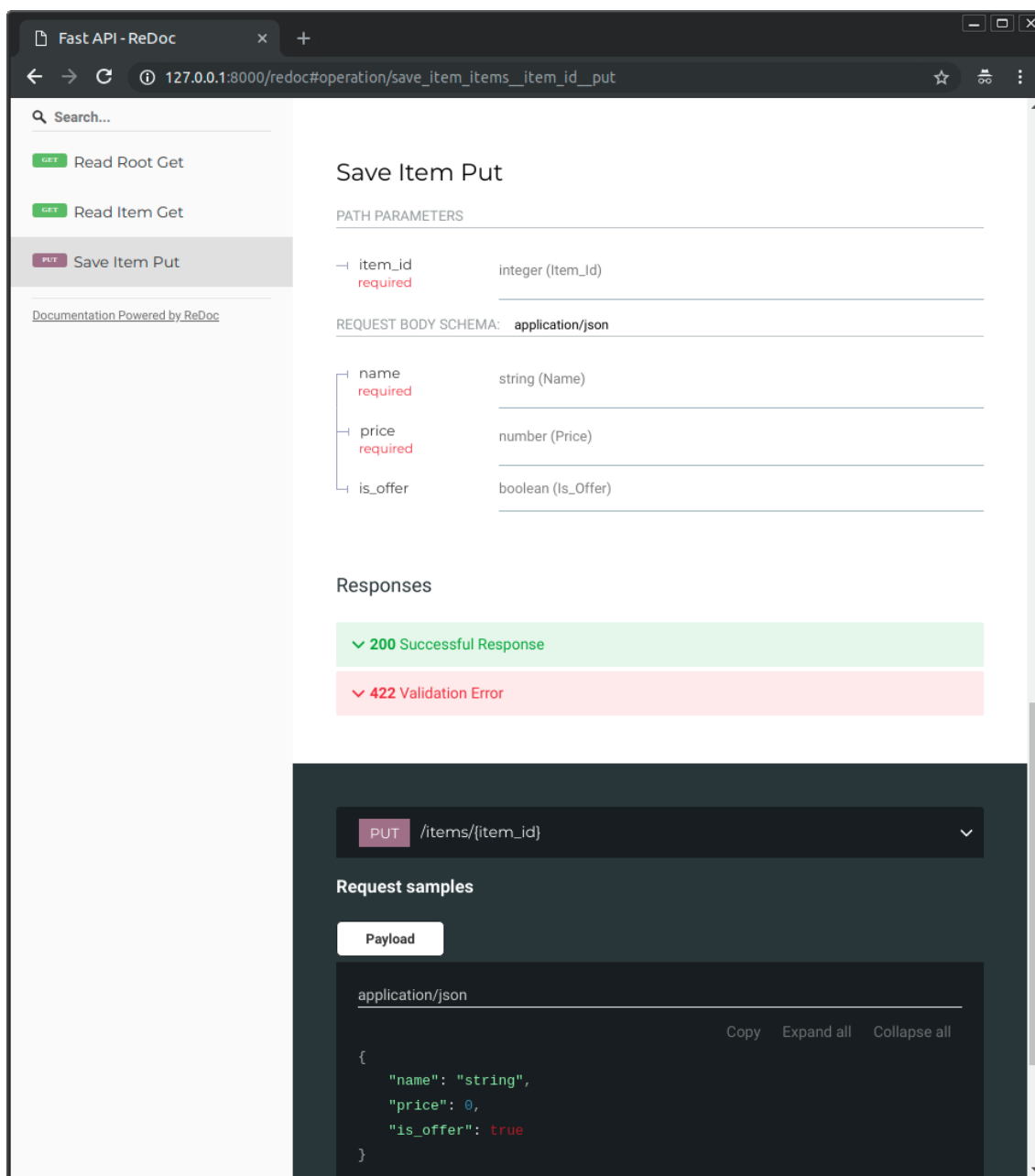
- 然后点击「Execute」按钮，用户界面将会和 API 进行通信，发送参数，获取结果并在屏幕上展示：



可选文档升级

访问 <http://127.0.0.1:8000/redoc>。

- 可选文档同样会体现新加入的请求参数和请求体：



总结

总的来说，你就像声明函数的参数类型一样只声明了一次请求参数、请求体等的类型。

你使用了标准的现代 Python 类型来完成声明。

你不需要去学习新的语法、了解特定库的方法或类，等等。

只需要使用标准的 **Python 3.6 及更高版本**。

举个例子，比如声明 `int` 类型：

```
item_id: int
```

或者一个更复杂的 `Item` 模型：

```
item: Item
```

.....在进行一次声明之后，你将获得：

- 编辑器支持，包括：
 - 自动补全
 - 类型检查
- 数据校验：
 - 在校验失败时自动生成清晰的错误信息
 - 对多层嵌套的 JSON 对象依然执行校验
- **转换** 来自网络请求的输入数据为 Python 数据类型。包括以下数据：
 - JSON
 - 路径参数
 - 查询参数
 - Cookies
 - 请求头
 - 表单
 - 文件
- **转换** 输出的数据：转换 Python 数据类型为供网络传输的 JSON 数据：
 - 转换 Python 基础类型（`str`、`int`、`float`、`bool`、`list` 等）
 - `datetime` 对象
 - `UUID` 对象
 - 数据库模型
 -以及更多其他类型
- 自动生成的交互式 API 文档，包括两种可选的用户界面：
 - Swagger UI
 - ReDoc

回到前面的代码示例，**FastAPI** 将会：

- 校验 `GET` 和 `PUT` 请求的路径中是否含有 `item_id`。
- 校验 `GET` 和 `PUT` 请求中的 `item_id` 是否为 `int` 类型。
 - 如果不是，客户端将会收到清晰有用的错误信息。
- 检查 `GET` 请求中是否有命名为 `q` 的可选查询参数（比如 `http://127.0.0.1:8000/items/foo?q=somequery`）。
 - 因为 `q` 被声明为 `= None`，所以它是可选的。
 - 如果没有 `None` 它将会是必需的（如 `PUT` 例子中的请求体）。
- 对于访问 `/items/{item_id}` 的 `PUT` 请求，将请求体读取为 JSON 并：
 - 检查是否有必需属性 `name` 并且值为 `str` 类型。
 - 检查是否有必需属性 `price` 并且值为 `float` 类型。
 - 检查是否有可选属性 `is_offer`，如果有的话值应该为 `bool` 类型。
 - 以上过程对于多层嵌套的 JSON 对象同样也会执行
- 自动对 JSON 进行转换或转换成 JSON。
- 通过 OpenAPI 文档来记录所有内容，可被用于：
 - 交互式文档系统

- 许多编程语言的客户端代码自动生成系统
- 直接提供 2 种交互式文档 web 界面。

虽然我们才刚刚开始，但其实你已经了解了这一切是如何工作的。

尝试更改下面这行代码：

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

.....从：

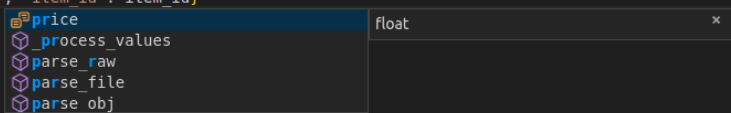
```
... "item_name": item.name ...
```

.....改为：

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

.....注意观察编辑器是如何自动补全属性并且还知道它们的类型：

```
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.name, "item_id": item_id}
26
```



[教程 - 用户指南](#) 中有包含更多特性的更完整示例。

剧透警告： 教程 - 用户指南中的内容有：

- 对来自不同地方的参数进行声明，如：**请求头**、**cookies**、**form 表单**以及**上传的文件**。
- 如何设置**校验约束**如 `maximum_length` 或者 `regex` 。
- 一个强大并易于使用的 **依赖注入** 系统。
- 安全性和身份验证，包括通过 **JWT 令牌**和 **HTTP 基本身份认证**来支持 **OAuth2**。
- 更进阶（但同样简单）的技巧来声明 **多层嵌套 JSON 模型**（借助 Pydantic）。
- 许多额外功能（归功于 Starlette）比如：

- **WebSockets**
- **GraphQL**
- 基于 `requests` 和 `pytest` 的极其简单的测试
- **CORS**
- **Cookie Sessions**
-以及更多

性能

独立机构 TechEmpower 所作的基准测试结果显示，基于 Uvicorn 运行的 **FastAPI** 程序是 [最快的 Python web 框架之二](#)，仅次于 Starlette 和 Uvicorn 本身（FastAPI 内部使用了它们）。(*)

想了解更多，请查阅 [基准测试](#) 章节。

可选依赖

用于 Pydantic:

- [ujson](#) - 更快的 JSON [\[解析\]](#)。
- [email_validator](#) - 用于 email 校验。

用于 Starlette:

- [requests](#) - 使用 `TestClient` 时安装。
- [aiofiles](#) - 使用 `FileResponse` 或 `StaticFiles` 时安装。
- [jinja2](#) - 使用默认模板配置时安装。
- [python-multipart](#) - 需要通过 `request.form()` 对表单进行 [\[解析\]](#) 时安装。
- [itsdangerous](#) - 需要 `SessionMiddleware` 支持时安装。
- [pyyaml](#) - 使用 Starlette 提供的 `SchemaGenerator` 时安装（有 FastAPI 你可能并不需要它）。
- [graphene](#) - 需要 `GraphQLApp` 支持时安装。
- [ujson](#) - 使用 `UJSONResponse` 时安装。

用于 FastAPI / Starlette:

- [uvicorn](#) - 用于加载和运行你的应用程序的服务器。
- [orjson](#) - 使用 `ORJSONResponse` 时安装。

你可以通过 `pip install fastapi[all]` 命令来安装以上所有依赖。

许可协议

该项目遵循 MIT 许可协议。