

特性

FastAPI 特性

FastAPI 提供了以下内容：

基于开放标准

- 用于创建 API 的 [OpenAPI](#) 包含了路径操作，请求参数，请求体，安全性等的声明。
- 使用 [JSON Schema](#) (因为 OpenAPI 本身就是基于 JSON Schema 的)自动生成数据模型文档。
- 经过了缜密的研究后围绕这些标准而设计。并非狗尾续貂。
- 这也允许了在很多语言中自动**生成客户端代码**。

自动生成文档

交互式 API 文档以及具探索性 web 界面。因为该框架是基于 OpenAPI，所以有很多可选项，FastAPI 默认自带两个交互式 API 文档。

- [Swagger UI](#)，可交互式操作，能在浏览器中直接调用和测试你的 API 。

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

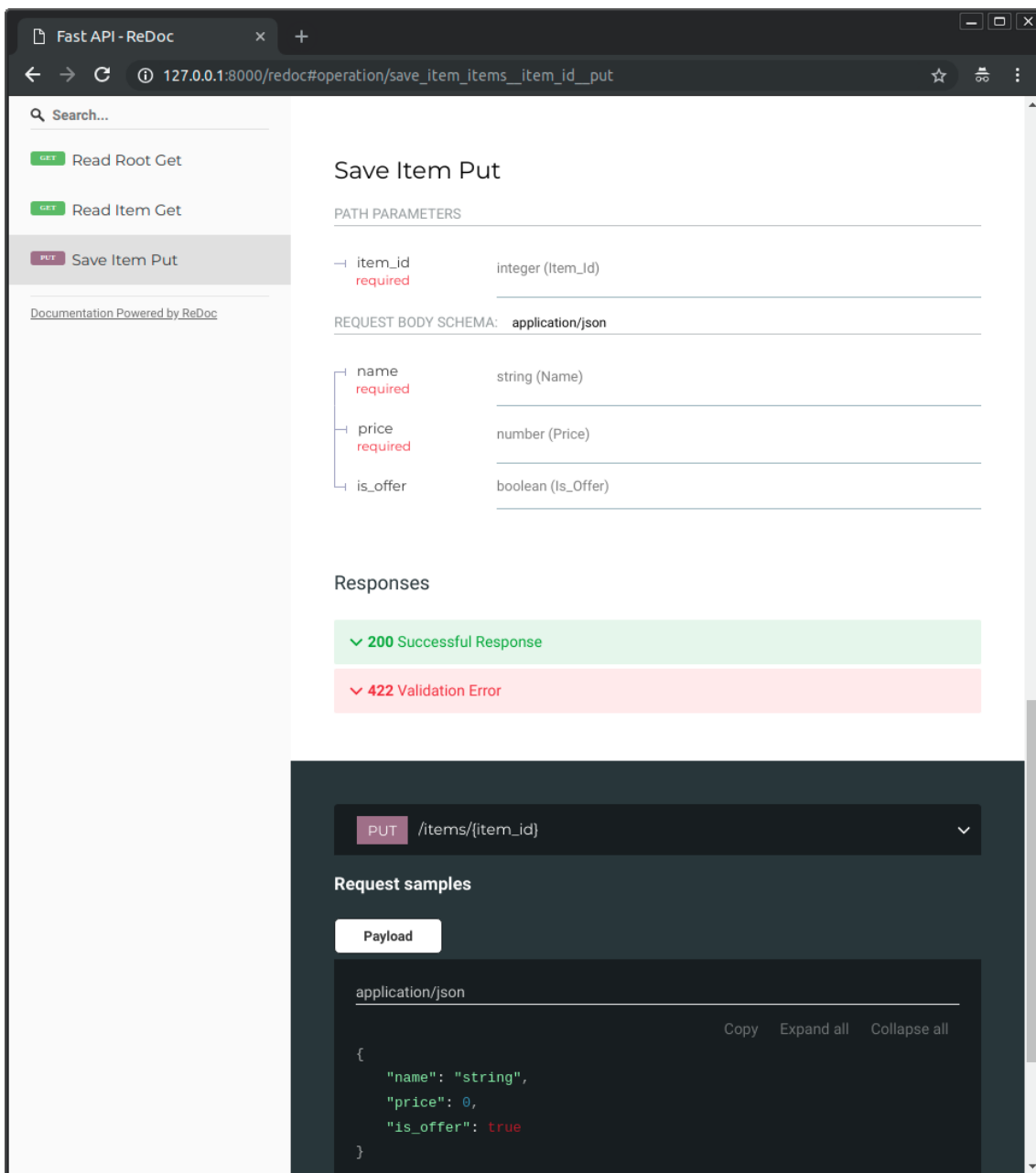
Example Value | Schema

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

Responses

Code	Description	Links
200	Successful Response	No links

- 另外的 API 文档: [ReDoc](#)



更主流的 Python

全部都基于标准的 **Python 3.6 类型声明**（感谢 Pydantic）。没有新的语法需要学习。只需要标准的 Python。

如果你需要2分钟来学习如何使用 Python 类型（即使你不使用 FastAPI），看看这个简短的教程：[Python Types](#)。{internal-link target=_blank}。

编写带有类型标注的标准 Python：

```
from datetime import date

from pydantic import BaseModel
```

```
# Declare a variable as a str
# and get editor support inside the function
def main(user_id: str):
    return user_id

# A Pydantic model
class User(BaseModel):
    id: int
    name: str
    joined: date
```

可以像这样来使用：

```
my_user: User = User(id=3, name="John Doe", joined="2018-07-19")

second_user_data = {
    "id": 4,
    "name": "Mary",
    "joined": "2018-11-30",
}

my_second_user: User = User(**second_user_data)
```

!!! info `**second_user_data` 意思是:

直接将`second_user_data`字典的键和值直接作为key-value参数传递，等同于：`User(id=4, name="Mary", joined="2018-11-30")`

编辑器支持

整个框架都被设计得易于使用且直观，所有的决定都在开发之前就在多个编辑器上进行了测试，来确保最佳的开发体验。

在最近的 Python 开发者调查中，我们能看到 [被使用最多的功能是“自动补全”](#)。

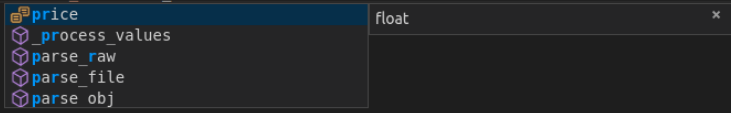
整个 **FastAPI** 框架就是基于这一点的。任何地方都可以进行自动补全。

你几乎不需要经常回来看文档。

在这里，你的编辑器可能会这样帮助你：

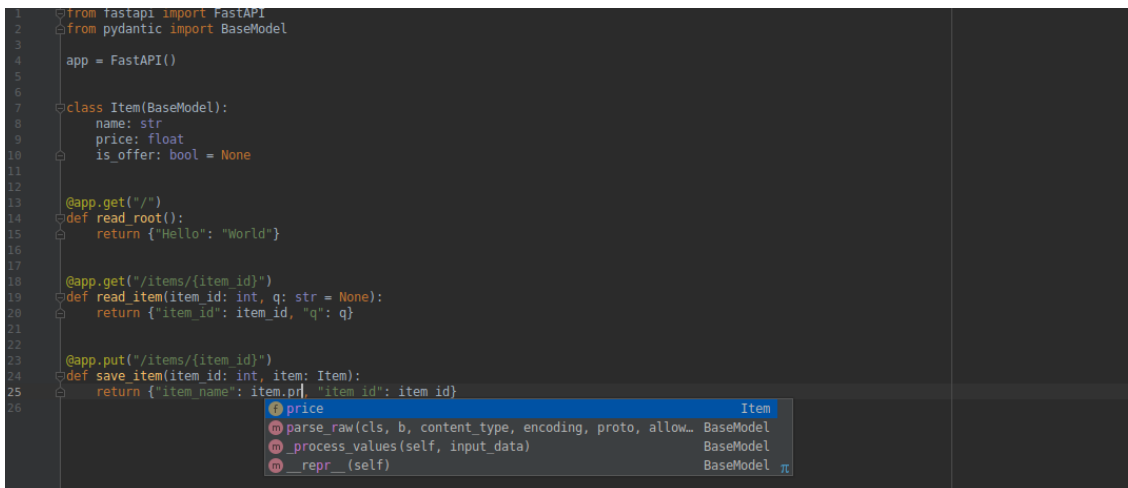
- [Visual Studio Code](#) 中:

```
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
```



The image shows a PyCharm IDE with a code editor and an autocomplete popup. The code defines a FastAPI application with an Item model and two endpoints. The autocomplete popup is triggered on the 'price' attribute in the Item model, showing options like 'price', '_process_values', 'parse_raw', 'parse_file', and 'parse_obj'. The 'price' option is selected, and its type 'float' is shown in a separate window.

- [PyCharm](#) 中:



The image shows a PyCharm IDE with a code editor. The code is the same as the previous image. The autocomplete popup is triggered on the 'price' attribute in the Item model, showing options like 'price', '_process_values', 'parse_raw', 'parse_file', and 'parse_obj'. The 'price' option is selected, and its type 'float' is shown in a separate window.

你将来能进行代码补全，这是在之前你可能曾认为不可能的事。例如，在来自请求 JSON 体（可能是嵌套的）中的键 `price`。

不会再输错键名，来回翻看文档，或者来回滚动寻找你最后使用的 `username` 或者 `user_name`。

简洁

任何类型都有合理的默认值，任何地方都有可选配置。所有的参数被微调，来满足你的需求，定义成你需要的 API。

但是默认情况下，一切都能“顺利工作”。

验证

- 校验大部分（甚至所有？）的 Python **数据类型**，包括：

- JSON 对象 (`dict`)。
- JSON 数组 (`list`) 定义成员类型。
- 字符串 (`str`) 字段, 定义最小或最大长度。
- 数字 (`int` , `float`) 有最大值和最小值，等等。

- 校验外来类型，比如：

- URL。
- Email。
- UUID。
- ...及其他。

所有的校验都由完善且强大的 **Pydantic** 处理。

安全性及身份验证

集成了安全性和身份认证。杜绝数据库或者数据模型的渗透风险。

OpenAPI 中定义的安全模式，包括：

- HTTP 基本认证。
- **OAuth2** (也使用 **JWT tokens**)。在 [OAuth2 with JWT](#) (internal-link target=_blank)查看教程。
- API 密钥，在：
 - 请求头。
 - 查询参数。
 - Cookies, 等等。

加上来自 Starlette（包括 **session cookie**）的所有安全特性。

所有的这些都是可复用的工具和组件，可以轻松与你的系统，数据仓库，关系型以及 NoSQL 数据库等等集成。

依赖注入

FastAPI 有一个使用非常简单，但是非常强大的**依赖注入**系统。

- 甚至依赖也可以有依赖，创建一个层级或者“**图**”**依赖**。
- 所有**自动化处理**都由框架完成。
- 所有的依赖关系都可以从请求中获取数据，并且**增加了路径操作**约束和自动文档生成。
- 即使在依赖项中被定义的**路径操作**也会**自动验证**。
- 支持复杂的用户身份认证系统，**数据库连接**等等。
- **不依赖**数据库，前端等。但是和它们集成很简单。

无限制“插件”

或者说，导入并使用你需要的代码，而不需要它们。

任何集成都被设计得被易于使用（用依赖关系），你可以用和**路径操作**相同的结构和语法，在两行代码中为你的应用创建一个“插件”。

测试

- 100% **测试覆盖**。
- 代码库100% **类型注释**。
- 用于生产应用。

Starlette 特性

FastAPI 和 **Starlette** 完全兼容(并基于)。所以，你有的其他的 **Starlette** 代码也能正常工作。**FastAPI** 实际上是 **Starlette** 的一个子类。所以，如果你已经知道或者使用 **Starlette**，大部分的功能会以相同的方式工作。

通过 **FastAPI** 你可以获得所有 **Starlette** 的特性 (**FastAPI** 就像加强版的 **Starlette**)：

- 令人惊叹的性能。它是 [Python 可用的最快的框架之一](#)，和 **NodeJS** 及 **Go** 相当。
- 支持 **WebSocket**。
- 支持 **GraphQL**。
- 后台任务处理。
- Startup 和 shutdown 事件。
- 测试客户端基于 `requests`。
- **CORS**, **GZip**, 静态文件, 流响应。
- 支持 **Session** 和 **Cookie**。
- 100% 测试覆盖率。
- 代码库 100% 类型注释。

Pydantic 特性

FastAPI 和 **Pydantic** 完全兼容(并基于)。所以，你有的其他的 **Pydantic** 代码也能正常工作。

兼容包括基于 **Pydantic** 的外部库，例如用与数据库的 **ORMs**, **ODMs**。

这也意味着在很多情况下，你可以将从请求中获得的相同对象**直接传到数据库**，因为所有的验证都是自动的。

反之亦然，在很多情况下，你也可以将从数据库中获取的对象**直接传到客户端**。

通过 **FastAPI** 你可以获得所有 **Pydantic** (**FastAPI** 基于 **Pydantic** 做了所有的数据处理)：

- **更简单：**
 - 没有新的模式定义 **micro-language** 需要学习。
 - 如果你知道 **Python types**，你就知道如何使用 **Pydantic**。
- 和你 **IDE/linter/brain** 适配:
 - 因为 **pydantic** 数据结构仅仅是你定义的类的实例；自动补全，**linting**，**mypy** 以及你的直觉应该可以和你验证的数据一起正常工作。
- **更快：**
 - 在 [基准测试](#) 中，**Pydantic** 比其他被测试的库都要快。
- 验证**复杂结构**:
 - 使用分层的 **Pydantic** 模型, **Python typing** 的 **List** 和 **Dict** 等等。
 - 验证器使我们能够简单清楚的将复杂的数据模式定义、检查并记录为 **JSON Schema**。
 - 你可以拥有深度**嵌套的 JSON** 对象并对它们进行验证和注释。
- **可扩展：**
 - **Pydantic** 允许定义自定义数据类型或者你可以用验证器装饰器对被装饰的模型上的方法扩展验证。
- 100% 测试覆盖率。