

请求体

当你需要将数据从客户端（例如浏览器）发送给 API 时，你将其作为「请求体」发送。

请求体是客户端发送给 API 的数据。**响应体**是 API 发送给客户端的数据。

你的 API 几乎总是要发送**响应体**。但是客户端并不总是需要发送**请求体**。

我们使用 [Pydantic](#) 模型来声明**请求体**，并能够获得它们所具有的所有能力和优点。

!!! info 你不能使用 `GET` 操作（HTTP 方法）发送请求体。

要发送数据，你必须使用下列方法之一：``POST``（较常见）、``PUT``、``DELETE`` 或 ``PATCH``。

导入 Pydantic 的 `BaseModel`

首先，你需要从 `pydantic` 中导入 `BaseModel`：

```
{!../../../../../docs_src/body/tutorial001.py!}
```

创建数据模型

然后，将你的数据模型声明为继承自 `BaseModel` 的类。

使用标准的 Python 类型来声明所有属性：

```
{!../../../../../docs_src/body/tutorial001.py!}
```

和声明查询参数时一样，当一个模型属性具有默认值时，它不是必需的。否则它是一个必需属性。将默认值设为 `None` 可使其成为可选属性。

例如，上面的模型声明了一个这样的 JSON「`object`」（或 Python `dict`）：

```
{
  "name": "Foo",
  "description": "An optional description",
  "price": 45.2,
  "tax": 3.5
}
```

...由于 `description` 和 `tax` 是可选的（它们的默认值为 `None`），下面的 JSON「`object`」也将是有效的：

```
{
  "name": "Foo",
  "price": 45.2
}
```

声明为参数

使用与声明路径和查询参数的相同方式声明请求体，即可将其添加到「路径操作」中：

```
{!../../../../../docs_src/body/tutorial001.py!}
```

...并且将它的类型声明为你创建的 `Item` 模型。

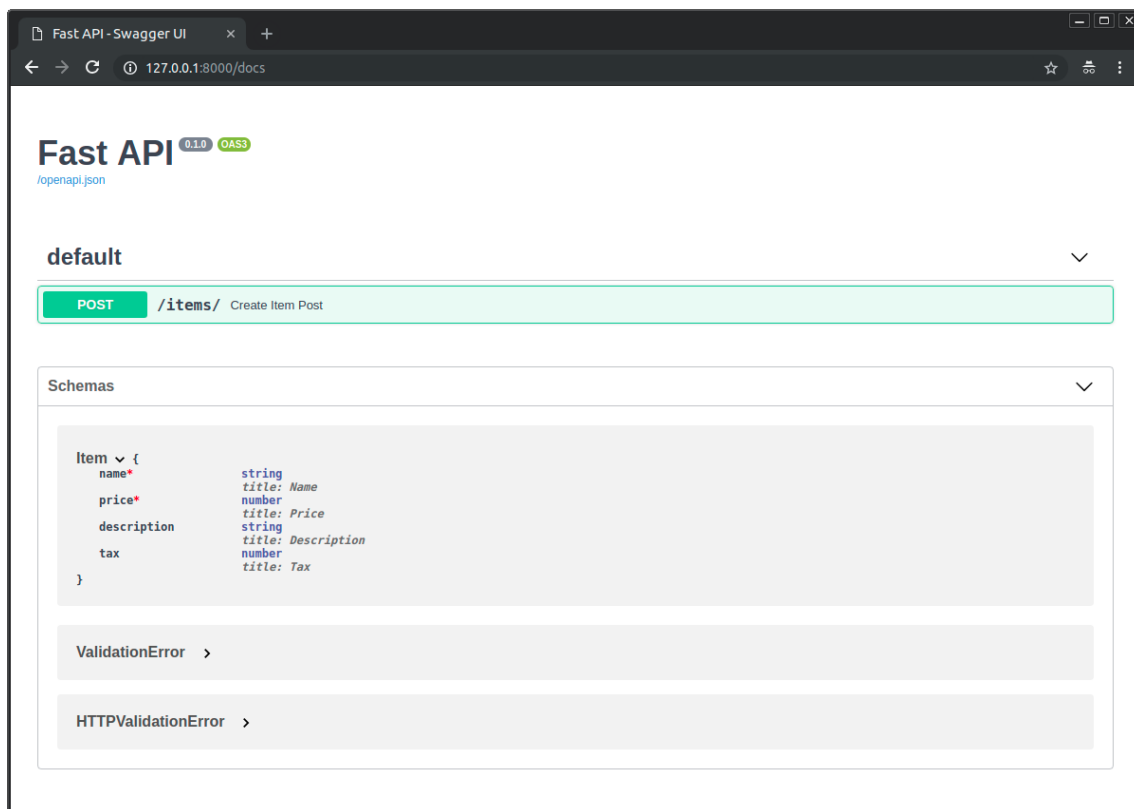
结果

仅仅使用了 Python 类型声明，**FastAPI** 将会：

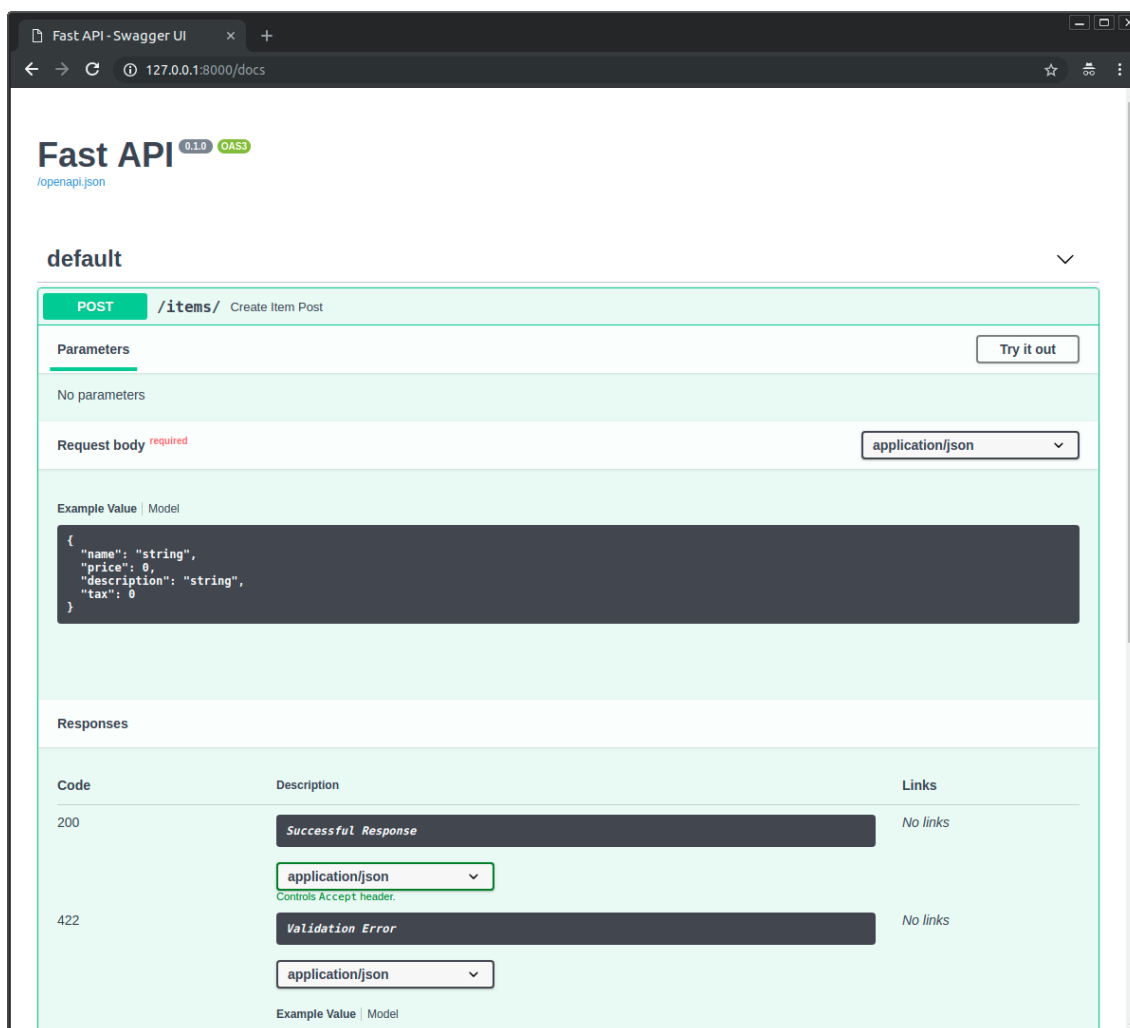
- 将请求体作为 JSON 读取。
- 转换为相应的类型（在需要时）。
- 校验数据。
 - 如果数据无效，将返回一条清晰易读的错误信息，指出不正确数据的确切位置和内容。
- 将接收的数据赋值到参数 `item` 中。
 - 由于你已经在函数中将它声明为 `Item` 类型，你还将获得对于所有属性及其类型的一切编辑器支持（代码补全等）。
- 为你的模型生成 [JSON 模式](#) 定义，你还可以在其他任何对你的项目有意义的地方使用它们。
- 这些模式将成为生成的 OpenAPI 模式的一部分，并且被自动化文档 [UI](#) 所使用。

自动化文档

你所定义模型的 JSON 模式将成为生成的 OpenAPI 模式的一部分，并且在交互式 API 文档中展示：



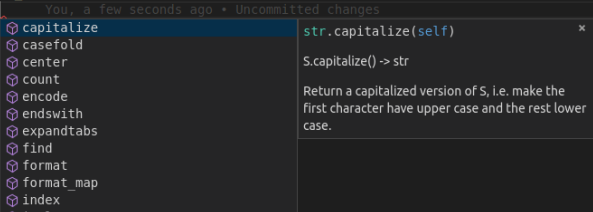
而且还将在每一个需要它们的路径操作的 API 文档中使用：



编辑器支持

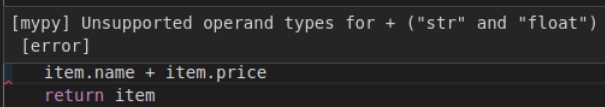
在你的编辑器中，你会在函数内部的任意地方得到类型提示和代码补全（如果你接收的是一个 `dict` 而不是 Pydantic 模型，则不会发生这种情况）：

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4
5 class Item(BaseModel):
6     name: str
7     description: str = None
8     price: float
9     tax: float = None
10
11
12 app = FastAPI()
13
14
15 @app.post("/items/")
16 async def create_item(item: Item):
17     item.name =
18     return item
19
```



你还会获得对不正确的类型操作的错误检查：

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4
5 class Item(BaseModel):
6     name: str
7     description: str = None
8     price: float
9     tax: float = None
10
11
12 app = FastAPI()
13
14
15 @app.post("/items/")
16 async def create_item(item: Item):
17
18     [mypy] Unsupported operand types for + ("str" and "float")
19     [error]
20     item.name + item.price
21     return item
22
```



这并非偶然，整个框架都是围绕该设计而构建。

并且在进行任何实现之前，已经在设计阶段经过了全面测试，以确保它可以在所有的编辑器中生效。

Pydantic 本身甚至也进行了一些更改以支持此功能。

上面的截图取自 [Visual Studio Code](#)。

但是在 [PyCharm](#) 和绝大多数其他 Python 编辑器中你也会获得同样的编辑器支持：

```

1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4
5 class Item(BaseModel):
6     name: str
7     description: str = None
8     price: float
9     tax: float = None
10
11
12 app = FastAPI()
13
14
15 @app.post("/items/")
16 async def create_item(item: Item):
17     item.name = item.name.capitalize()
18     return item
19

```

使用模型

在函数内部，你可以直接访问模型对象的所有属性：

```
{!../../../../../docs_src/body/tutorial002.py!}
```

请求体 + 路径参数

你可以同时声明路径参数和请求体。

FastAPI 将识别出与路径参数匹配的函数参数应**从路径中获取**，而声明为 Pydantic 模型的函数参数应**从请求体中获取**。

```
{!../../../../../docs_src/body/tutorial003.py!}
```

请求体 + 路径参数 + 查询参数

你还可以同时声明**请求体**、**路径参数**和**查询参数**。

FastAPI 会识别它们中的每一个，并从正确的位置获取数据。

```
{!../../../../../docs_src/body/tutorial004.py!}
```

函数参数将依次按如下规则进行识别：

- 如果在**路径**中也声明了该参数，它将被用作路径参数。
- 如果参数属于**单一类型**（比如 `int`、`float`、`str`、`bool` 等）它将被解释为**查询参数**。
- 如果参数的类型被声明为一个 **Pydantic 模型**，它将被解释为**请求体**。

不使用 Pydantic

如果你不想使用 Pydantic 模型，你还可以使用 **Body** 参数。请参阅文档 [请求体 - 多个参数: 请求体中的单一值](#) {internal-link target=_blank}。