请求体 - 嵌套模型

使用 **FastAPI**，你可以定义、校验、记录文档并使用任意深度嵌套的模型（归功于Pydantic）。

List 字段

你可以将一个属性定义为拥有子元素的类型。例如 Python `list`：

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

这将使 `tags` 成为一个由元素组成的列表。不过它没有声明每个元素的类型。

具有子类型的 List 字段

但是 Python 有一种特定的方法来声明具有子类型的列表：

从 typing 导入 `List`

首先，从 Python 的标准库 `typing` 模块中导入 `List`：

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

声明具有子类型的 List

要声明具有子类型的类型，例如 `list`、`dict`、`tuple`：

- 从 `typing` 模块导入它们
- 使用方括号 `[` 和 `]` 将子类型作为「类型参数」传入

```
from typing import List

my_list: List[str]
```

这完全是用于类型声明的标准 Python 语法。

对具有子类型的模型属性也使用相同的标准语法。

因此，在我们的示例中，我们可以将 `tags` 明确地指定为一个「字符串列表」：

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

Set 类型

但是随后我们考虑了一下，意识到标签不应该重复，它们很大可能会是唯一的字符串。

Python 具有一种特殊的数据类型来保存一组唯一的元素，即 `set`。

然后我们可以导入 `Set` 并将 `tag` 声明为一个由 `str` 组成的 `set`：

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

这样，即使你收到带有重复数据的请求，这些数据也会被转换为一组唯一项。

而且，每当你输出该数据时，即使源数据有重复，它们也将作为一组唯一项输出。

并且还会被相应地标注 / 记录文档。

嵌套模型

Pydantic 模型的每个属性都具有类型。

但是这个类型本身可以是另一个 Pydantic 模型。

因此，你可以声明拥有特定属性名称、类型和校验的深度嵌套的 JSON 对象。

上述这些都可以任意的嵌套。

定义子模型

例如，我们可以定义一个 `Image` 模型：

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

将子模型用作类型

然后我们可以将其用作一个属性的类型：

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

这意味着 **FastAPI** 将期望类似于以下内容的请求体：

```
{
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2,
    "tags": ["rock", "metal", "bar"],
    "image": {
        "url": "http://example.com/baz.jpg",
        "name": "The Foo live"
    }
}
```

再一次，仅仅进行这样的声明，你将通过 **FastAPI** 获得：

- 对被嵌入的模型也适用的编辑器支持（自动补全等）
- 数据转换
- 数据校验
- 自动生成文档

特殊的类型和校验

除了普通的单一值类型（如 `str`、`int`、`float` 等）外，你还可以使用从 `str` 继承的更复杂的单一值类型。

要了解所有的可用选项，请查看关于 来自 Pydantic 的外部类型 的文档。你将在下一章节中看到一些示例。

例如，在 `Image` 模型中我们有一个 `url` 字段，我们可以把它声明为 Pydantic 的 `HttpUrl`，而不是 `str`：

```
{!../../../docs_src/body_nested_models/tutorial005.py!}
```

该字符串将被检查是否为有效的 URL，并在 JSON Schema / OpenAPI 文档中进行记录。

带有一组子模型的属性

你还可以将 Pydantic 模型用作 `list`、`set` 等的子类型：

```
{!../../../docs_src/body_nested_models/tutorial006.py!}
```

这将期望（转换，校验，记录文档等）下面这样的 JSON 请求体：

```
{
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2,
    "tags": [
        "rock",
        "metal",
        "bar"
    ],
    "images": [
        {
            "url": "http://example.com/baz.jpg",
            "name": "The Foo live"
        },
        {
            "url": "http://example.com/dave.jpg",
            "name": "The Baz"
        }
    ]
}
```

!!! info 请注意 `images` 键现在具有一组 image 对象是如何发生的。

深度嵌套模型

你可以定义任意深度的嵌套模型：

```
{!../../../docs_src/body_nested_models/tutorial007.py!}
```

!!! info 请注意 `Offer` 拥有一组 `Item` 而反过来 `Item` 又是一个可选的 `Image` 列表是如何发生的。

纯列表请求体

如果你期望的 JSON 请求体的最外层是一个 JSON `array`（即 Python `list`），则可以在路径操作函数的参数中声明此类型，就像声明 Pydantic 模型一样：

```
images: List[Image]
```

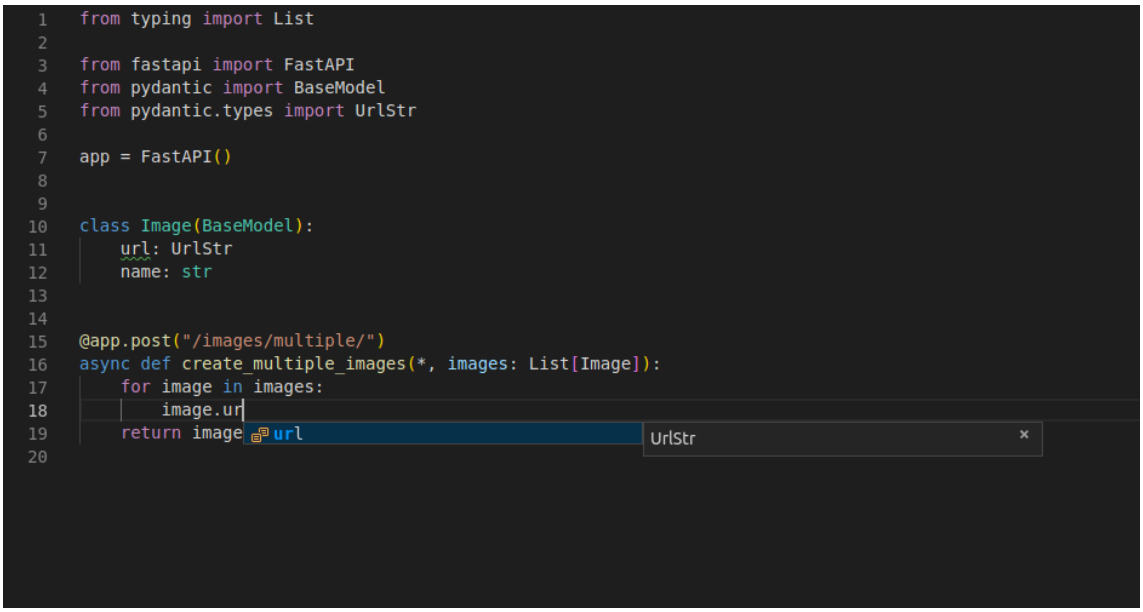
例如：

```
{!../../../docs_src/body_nested_models/tutorial008.py!}
```

无处不在的编辑器支持

你可以随处获得编辑器支持。

即使是列表中的元素：



如果你直接使用 `dict` 而不是 Pydantic 模型，那你将无法获得这种编辑器支持。

但是你根本不必担心这两者，传入的字典会自动被转换，你的输出也会自动被转换为 JSON。

任意 `dict` 构成的请求体

你也可以将请求体声明为使用某类型的键和其他类型值的 `dict`。

无需事先知道有效的字段/属性（在使用 Pydantic 模型的场景）名称是什么。

如果你想接收一些尚且未知的键，这将很有用。

其他有用的场景是当你想要接收其他类型的键时，例如 `int`。

这也是我们在接下来将看到的。

在下面的例子中，你将接受任意键为 `int` 类型并且值为 `float` 类型的 `dict`：

```
{!../../../docs_src/body_nested_models/tutorial009.py!}
```

!!! tip 请记住 JSON 仅支持将 `str` 作为键。

但是 `Pydantic` 具有自动转换数据的功能。

这意味着，即使你的 `API` 客户端只能将字符串作为键发送，只要这些字符串内容仅包含整数，`Pydantic` 就会对其进行转换并校验。

然后你接收的名为 `weights` 的 `dict` 实际上将具有 `int` 类型的键和 `float` 类型的值。

总结

使用 **FastAPI** 你可以拥有 Pydantic 模型提供的极高灵活性，同时保持代码的简单、简短和优雅。

而且还具有下列好处：

- 编辑器支持（处处皆可自动补全！）
- 数据转换（也被称为解析/序列化）
- 数据校验
- 模式文档
- 自动生成的文档