

# آموزش جامع FastAPI

## فهرست

مقدمه: معرفی انواع داده در پایتون.....	۴
فصل ۱: اولین گام‌ها.....	۱۰
فصل ۲: پارامترهای مسیر.....	۱۸
فصل ۳: پارامترهای پرس‌وجو.....	۲۸
فصل ۴: بدنه‌ی درخواست.....	۳۳
فصل ۵: پارامترهای پرس‌وجو و اعتبارسنجی رشته‌ای.....	۴۱
فصل ۶: پارامترهای مسیر و اعتبارسنجی‌های عددی.....	۵۶
فصل ۷: مدل‌های پارامتر پرس‌وجو.....	۶۱
فصل ۸: بدنه - چندین پارامتر.....	۶۴
فصل ۹: بدنه - فیلدها.....	۷۰
فصل ۱۰: بدنه - مدل‌های تو در تو.....	۷۲
فصل ۱۱: اعلان نمونه داده‌های درخواست.....	۸۰
فصل ۱۲: انواع داده‌ای اضافی.....	۸۸
فصل ۱۳: پارامترهای کوکی.....	۹۱
فصل ۱۴: پارامترهای هدر.....	۹۳
فصل ۱۵: مدل‌های پارامتر کوکی.....	۹۶
فصل ۱۶: مدل‌های پارامتر هدر.....	۹۹
فصل ۱۷: مدل پاسخ - نوع بازگشتی.....	۱۰۳
فصل ۱۸: مدل‌های اضافی.....	۱۱۹
فصل ۱۹: کد وضعیت پاسخ.....	۱۲۶
فصل ۲۰: داده‌های فرم.....	۱۳۰
فصل ۲۱: مدل‌های فرم.....	۱۳۲

فصل ۲۲: فایل درخواست ..... ۱۳۵

فصل ۲۳: فرم‌ها و فایل‌های درخواست ..... ۱۴۱

فصل ۲۴: مدیریت خطاها ..... ۱۴۳

## مقدمه: معرفی انواع داده در پایتون

در اینجا بخش‌های مقدماتی و آموزش‌هایی برای یادگیری FastAPI آورده شده‌اند. می‌توانید این مجموعه را به‌عنوان یک کتاب، یک دوره آموزشی، و روش رسمی و توصیه‌شده برای یادگیری FastAPI در نظر بگیرید.

پایتون از «اعلان نوع» اختیاری (که «حاشیه‌نویسی نوع» نیز نامیده می‌شوند) پشتیبانی می‌کند. این «اعلان‌های نوع» یا حاشیه‌نویسی‌ها، سیستم‌کس خاصی هستند که اجازه می‌دهند نوع یک متغیر را اعلام کنیم. با اعلام نوع متغیرهایتان، ویرایشگرها و ابزارها می‌توانند پشتیبانی بهتری ارائه دهند.

این فقط یک آموزش سریع / یادآوری درباره اعلان‌های نوع پایتون است. تنها حداقل‌های لازم برای استفاده از آن‌ها با FastAPI را پوشش می‌دهد که در واقع بسیار ابتدایی و کم است. FastAPI کاملاً بر اساس این اعلان‌های نوع ساخته شده است، که به آن مزایای زیادی می‌دهد. اما حتی اگر قصد استفاده از FastAPI را ندارید، مقداری یادگیری درباره آن سودمند خواهد بود.

**نکته:** اگر متخصص پایتون هستید و همه چیز را درباره اعلان‌های نوع می‌دانید، به فصل بعدی بروید.

### انگیزه

بیایید با یک مثال ساده شروع کنیم:

```
def get_full_name(first_name, last_name):
    full_name = first_name.title() + " " + last_name.title()
    return full_name

print(get_full_name("john", "doe"))
```

اجرای این برنامه خروجی زیر را می‌دهد:

```
John Doe
```

این تابع موارد زیر را انجام می‌دهد:

- یک `first_name` و `last_name` می‌گیرد.
- حرف اول هر کدام را با `title()` به حرف بزرگ تبدیل می‌کند.
- آن‌ها را با یک فاصله در وسط به هم متصل می‌کند.

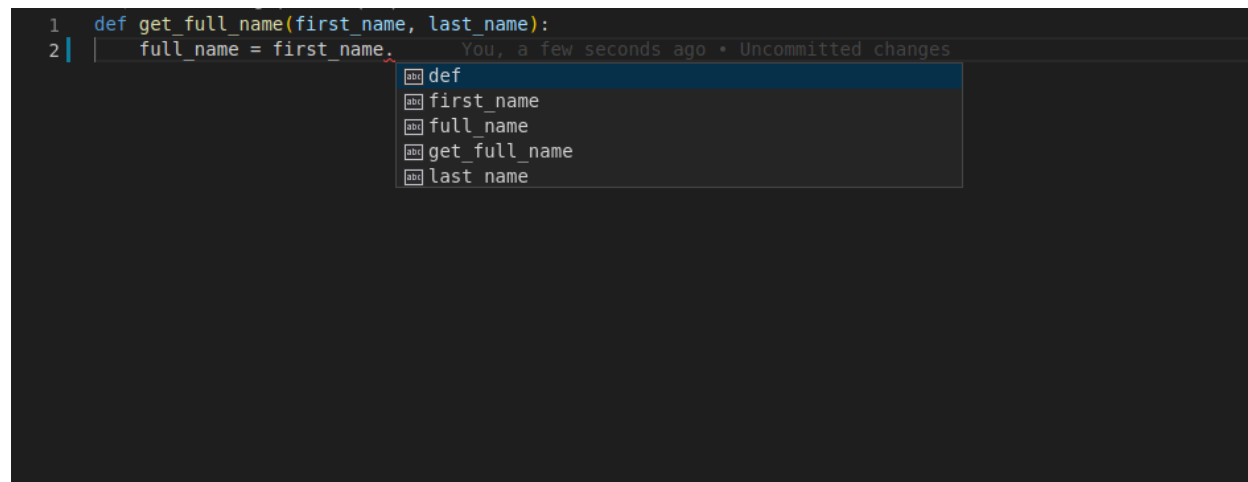
### ویرایش آن

این یک برنامه بسیار ساده است. اما حالا تصور کنید که آن را از ابتدا می‌نویسید. در یک لحظه تعریف تابع را شروع می‌کنید، پارامترها

آماده‌اند...

اما بعد باید آن «متدی که حرف اول را به بزرگ تبدیل می‌کند» را صدا بزنید. آیا نام آن upper بود؟ یا uppercase؟  
 ؟capitalize ؟first\_uppercase

سپس از دوست قدیمی برنامه‌نویسان، تکمیل خودکار ویرایشگر استفاده می‌کنید. اولین پارامتر تابع، first\_name را تایپ می‌کنید، سپس یک نقطه (.) و سپس Ctrl+Space را می‌زنید تا تکمیل انجام شود. اما متأسفانه، هیچ چیز مفیدی دریافت نمی‌کنید:



```

1 def get_full_name(first_name, last_name):
2 |     full_name = first_name.
    You, a few seconds ago • Uncommitted changes
    def
    first_name
    full_name
    get_full_name
    last_name
  
```

## افزودن انواع

بیاید فقط یک خط از نسخه قبلی را تغییر دهیم. دقیقاً این بخش، پارامترهای تابع، را از:

```
first_name, last_name
```

به:

```
first_name: str, last_name: str
```

تغییر می‌دهیم. همین. این‌ها همان «راهنماهای نوع» هستند:

```

def get_full_name(first_name: str, last_name: str):
    full_name = first_name.title() + " " + last_name.title()
    return full_name

print(get_full_name("john", "doe"))
  
```

این همانند اعلام مقدار پیش فرض نیست، مثل:

```
first_name="john", last_name="doe"
```

این چیز متفاوتی است. ما از دونقطه (:) استفاده می‌کنیم، نه علامت مساوی (=). و افزودن راهنماهای نوع معمولاً چیزی را در عملکرد تغییر نمی‌دهد. اما حالا، تصور کنید دوباره وسط نوشتن همان تابع هستید، اما با راهنماهای نوع. در همان لحظه، دوباره تکمیل خودکار را با Ctrl+Space فعال می‌کنید و می‌بینید:

```
1 def get_full_name(first_name: str, last_name: str):
2 |     full_name = first_name.
```

You, a few seconds ago • Uncommitted changes

- \* format
- \* join
- \* split
- \* encode
- capitalize
- casefold
- center
- count
- endswith
- expandtabs
- find
- format map

**str.format(self, \*args, \*\*kwargs)** ✕

S.format(args, \*kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

با آن، می‌توانید پیمایش کرده و گزینه‌ای که آشنا به نظر می‌رسد را پیدا کنید:

```
1 def get_full_name(first_name: str, last_name: str):
2 |     full_name = first_name.
```

You, a few seconds ago • Uncommitted changes

- rfind
- rindex
- rjust
- rpartition
- rsplit
- rstrip
- splitlines
- startswith
- strip
- swapcase
- title**
- translate

**str.title(self)** ✕

S.title() -> str

Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

## انگیزه بیشتر

این تابع را بررسی کنید، از قبل راهنماهای نوع دارد:

```
def get_name_with_age(name: str, age: int):
```

```
name_with_age = name + " is this old: " + age
return name_with_age
```

چون ویرایشگر نوع متغیرها را می‌داند، نه تنها تکمیل را دریافت می‌کنید، بلکه خطاها را هم شناسایی می‌کند:

```
1 def get_name_with_age(name: str, age: int):
2
3     [mypy] Unsupported operand types for + ("str" and "int")
4     [error]
5     name_with_age = name + " is this old: " + age
6     return name_with_age
7
```

حالا می‌دانید که باید آن را اصلاح کرده و age را با str(age) به رشته تبدیل کنید:

```
def get_name_with_age(name: str, age: int):
    name_with_age = name + " is this old: " + str(age)
    return name_with_age
```

## اعلام انواع

مکان اصلی اعلام راهنماهای نوع را همین الآن دیدید. در پارامترهای تابع. این مکان اصلی است که در FastAPI نیز از آن‌ها استفاده خواهید کرد.

## انواع ساده

می‌توانید تمام انواع استاندارد پایتون را اعلام کنید، نه فقط str. می‌توانید از موارد زیر نیز استفاده کنید:

- int •
- float •
- bool •
- bytes •

```
def get_items(item_a: str, item_b: int, item_c: float, item_d: bool, item_e:
    bytes):
    return item_a, item_b, item_c, item_d, item_d, item_e
```

## انواع عمومی با پارامترهای نوع

برخی ساختارهای داده می‌توانند مقادیر دیگر را در خود نگه دارند، مانند dict، list، set و tuple. و مقادیر داخلی آن‌ها

می‌توانند نوع خود را داشته باشند. این انواع که دارای نوع داخلی هستند، «انواع عمومی» (generic) نامیده می‌شوند. و می‌توان آن‌ها را با نوع داخلی‌شان اعلام کرد. برای اعلام آن‌ها و نوع داخلی‌شان، می‌توانید از ماژول استاندارد typing استفاده کنید. این ماژول مخصوص پشتیبانی از این راهنماهای نوع وجود دارد.

## نسخه‌های جدیدتر پایتون

نحو استفاده از typing با تمام نسخه‌ها، از پایتون ۳٫۶ به بالا، سازگار است. با پیشرفت پایتون، نسخه‌های جدیدتر پشتیبانی بهتری برای حاشیه‌نویسی نوع فراهم می‌کنند و اغلب حتی نیازی به وارد کردن و استفاده از ماژول typing نخواهید داشت. اگر می‌توانید نسخه جدیدتری از پایتون را برای پروژه‌تان انتخاب کنید، از آن سادگی اضافه بهره‌مند خواهید شد.

## لیست

برای مثال، بیایید یک متغیر تعریف کنیم که یک لیستی از رشته‌ها باشد. متغیر را با استفاده از همان نحو دو نقطه (:) تعریف می‌کنیم. به‌عنوان نوع، از list استفاده می‌کنیم. چون لیست، نوعی است که نوع‌های داخلی دارد (یعنی عناصری از نوع خاص درون خودش دارد)، آن نوع داخلی را داخل کروشه می‌نویسیم:

```
def process_items(items: list[str]):
    for item in items:
        print(item)
```

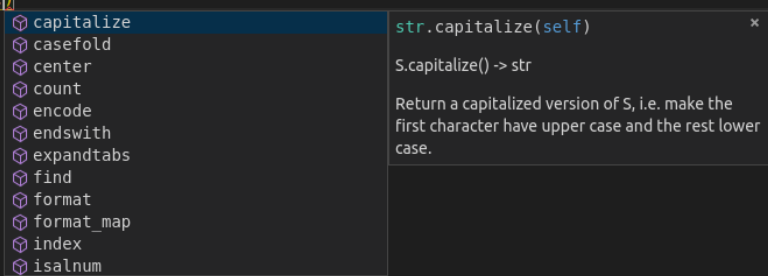
**اطلاعات:** این نوع‌های داخلی که در کروشه قرار می‌گیرند، «پارامترهای نوع» نام دارند.

در اینجا، str پارامتر نوعی است که به list داده شده. یعنی: متغیر items یک لیست است، و هر کدام از آیتم‌های درون آن از نوع str هستند.

**نکته:** اگر از پایتون ۳٫۹ یا بالاتر استفاده می‌کنید، نیازی نیست List را از typing ایمپورت کنید؛ می‌توانید از همان نوع list معمولی استفاده کنید.

با این کار، ویرایشگر شما حتی هنگام پردازش آیتم‌های درون لیست هم می‌تواند پشتیبانی ارائه دهد:

```
1 from typing import List
2
3 def process_items(items: List[str]):
4     for item in items:
5         print(item.capitalize())
6
```





بدون استفاده از راهنمای نوع، دستیابی به چنین قابلیت‌ای تقریباً غیرممکن است. دقت کنید که متغیر `item` یکی از عناصر موجود در لیست `items` است. و با این حال، ویرایشگر می‌داند که این متغیر از نوع `str` است، و امکانات مربوط به رشته‌ها را به شما پیشنهاد می‌دهد.

## Set و Tuple

برای تعریف `tuple` و `set` هم به همان روش عمل می‌کنید:

```
def process_items(items_t: tuple[int, int, str], items_s: set[bytes]):
    return items_t, items_s
```

این یعنی:

- متغیر `items_t` یک **تاپل شامل سه آیتِم** است: یک عدد صحیح، یک عدد صحیح دیگر، و یک رشته.
- متغیر `items_s` یک **مجموعه** است که هر کدام از عناصرش از نوع `bytes` هستند.

## دیکشنری (Dict)

برای تعریف یک دیکشنری، دو پارامتر نوع به آن می‌دهید که با کاما جدا شده‌اند.

پارامتر اول برای **کلیدهای** دیکشنری است.

پارامتر دوم برای **مقدارهای** دیکشنری است:

```
def process_items(prices: dict[str, float]):
    for item_name, item_price in prices.items():
        print(item_name)
        print(item_price)
```

یعنی:

متغیر `prices` یک دیکشنری است که:

- **کلیدهای آن از نوع `str` هستند** (مثلاً نام هر کالا)
- **مقدارهای آن از نوع `float` هستند** (مثلاً قیمت هر کالا)

## فصل ۱: اولین گام‌ها

ساده‌ترین فایل FastAPI می‌تواند به این شکل باشد:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

این کد را در فایلی با نام `main.py` ذخیره کنید. در خروجی، خطی مشابه زیر نمایش داده می‌شود:

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

این خط، آدرسی را نشان می‌دهد که اپلیکیشن شما در آن اجرا شده است.

### ۱-۱ بررسی آن

مرورگر خود را باز کرده و به آدرس <http://127.0.0.1:8000> بروید. در آنجا، یک پاسخ JSON خواهید دید:

```
{"message": "Hello World"}
```

### ۱-۲ مستندات تعاملی API

اکنون به آدرس <http://127.0.0.1:8000/docs> بروید. در اینجا، مستندات تعاملی API که توسط Swagger UI ارائه شده است، نمایش داده می‌شود.

Fast API 0.1.0 OAS3

/openapi.json

default

GET /items/{item\_id} Read Item Get

Parameters

Try it out

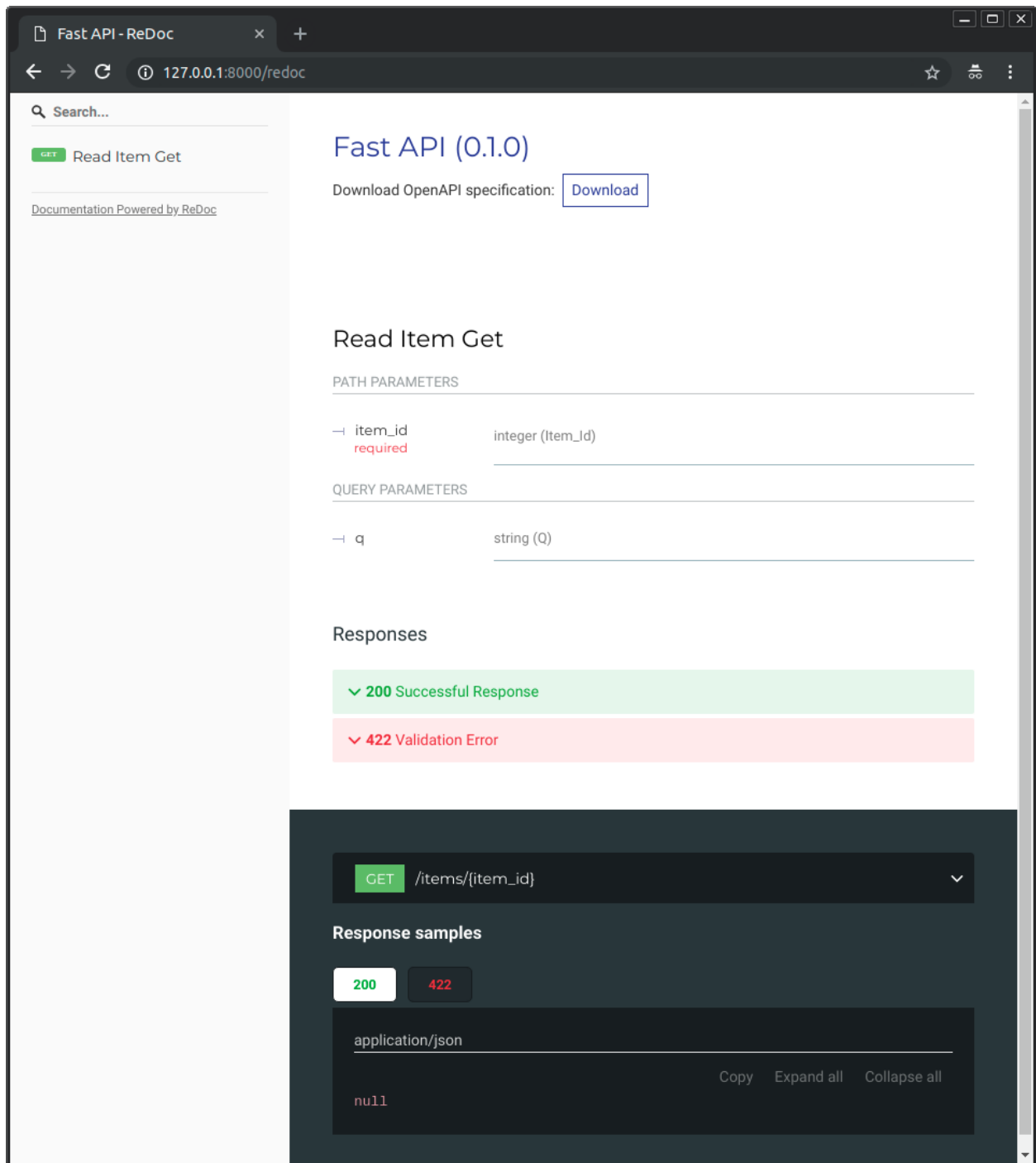
Name	Description
item_id <span style="color: red;">* required</span>	
integer	
(path)	
q	
string	
(query)	

Responses

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

### ۱-۳ مستندات جایگزین API

همچنین می‌توانید به آدرس <http://127.0.0.1:8000/redoc> مراجعه کنید. در این بخش، مستندات دیگری که توسط ReDoc ارائه شده، نمایش داده خواهد شد.



## OpenAPI ۱-۴

FastAPI یک «الگوی» کامل از API شما را با استفاده از استاندارد OpenAPI تولید می کند.

### ۱-۴-۱ «الگو» چیست؟

یک «الگو»، توصیفی از یک چیزی است - نه کد اجرایی، بلکه یک تعریف انتزاعی.

## ۲-۴-۱ الگو در API

در اینجا، OpenAPI استاندارد است که نحوه‌ی تعریف الگوی API را مشخص می‌کند. این تعریف شامل مسیرهای API، پارامترهای ممکن و غیره است.

## ۳-۴-۱ الگو در داده‌ها

گاهی اوقات، واژه‌ی «الگو» به ساختار داده‌ها (مانند JSON) نیز اشاره دارد. در این حالت، الگو مشخص می‌کند که داده‌ها شامل چه ویژگی‌هایی هستند و چه نوعی دارند.

## ۴-۴-۱ OpenAPI و الگوی JSON

OpenAPI یک الگو برای API تعریف می‌کند. این الگو شامل تعاریفی برای داده‌هایی است که با API ارسال و دریافت می‌شوند، که از الگوی JSON پیروی می‌کند. اگر کنجکاو هستید که الگوی خام OpenAPI چگونه است، می‌توانید آن را مستقیماً مشاهده کنید:

```
http://127.0.0.1:8000/openapi.json
```

خروجی، یک JSON خواهد بود که با چیزی مشابه زیر شروع می‌شود:

```
{
  "openapi": "3.1.0",
  "info": {
    "title": "FastAPI",
    "version": "0.1.0"
  },
  "paths": {
    "/items/": {
      "get": {
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {"schema": {}}
            }
          }
        }
      }
    }
  }
}
```

## ۵-۴-۱ OpenAPI چه کاربردی دارد؟

الگوی OpenAPI موتور مستندات تعاملی در FastAPI را تأمین می‌کند. همچنین، ده‌ها ابزار دیگر مبتنی بر OpenAPI وجود دارند که می‌توان از آن‌ها در FastAPI استفاده کرد. حتی می‌توان با استفاده از این الگو، به‌صورت خودکار کدهای کلاینت را برای ارتباط با API تولید کرد، مانند اپلیکیشن‌های فرانت‌اند، موبایل یا اینترنت اشیا.

## ۵-۱ مرور گام به گام

### گام ۱: وارد کردن FastAPI

```
from fastapi import FastAPI
```

FastAPI یک کلاس در پایتون است که تمام قابلیت‌های API را فراهم می‌کند.

**جزئیات فنی:** FastAPI در اصل از Starlette ارث‌بری می‌کند، بنابراین تمامی امکانات Starlette در FastAPI نیز قابل استفاده هستند.

### گام ۲: ایجاد یک «نمونه» از FastAPI

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

متغیر app یک نمونه از کلاس FastAPI خواهد بود که نقطه‌ی اصلی برای ایجاد API محسوب می‌شود.

### گام ۳: تعریف یک عملیات مسیریابی

#### الف) مسیر چیست؟

«مسیر» (Path) در اینجا به بخش انتهایی یک URL گفته می‌شود که از / شروع می‌شود. مثلاً، در آدرس:

```
https://example.com/items/foo
```

مسیر برابر خواهد بود با:

```
/items/foo
```

**نکته:** مسیر را می‌توان نقطه‌پایانی (Endpoint) یا مسیر (Route) نیز نامید.

هنگام ساخت یک API، «مسیر» اصلی‌ترین روش برای تفکیک «عملیات» (Concerns) و «منابع» (Resources) محسوب می‌شود.

#### ب) عملیات چیست؟

«عملیات» به یکی از متدهای HTTP اشاره دارد، مانند:

- POST
- GET
- PUT
- DELETE

و متدهای کمتر رایج مانند:

- OPTIONS
- HEAD
- PATCH
- TRACE

در پروتکل HTTP، می‌توانید با هر «مسیر» از طریق یک (یا چند) از این «متدها» ارتباط برقرار کنید. در توسعه API، هر متد HTTP معمولاً برای عملی خاص استفاده می‌شود:

- POST: ایجاد داده
- GET: خواندن داده
- PUT: به‌روزرسانی داده
- DELETE: حذف داده

در OpenAPI، هر یک از متدهای HTTP به‌عنوان یک «عملیات» در نظر گرفته می‌شود.

### ج) تعریف یک دکوراتور برای عملیات مسیر

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

دکوراتور `@app.get("/")` به FastAPI می‌گوید که تابع زیر آن، مسئول پردازش درخواست‌هایی است که:

- به مسیر / ارسال شده
- از متد GET استفاده می‌کنند.

**جزئیات دکوراتور:** آن دستور `@something` در پایتون، «دکوراتور» (Decorator) نامیده می‌شود.

شما آن را بالای یک تابع قرار می‌دهید؛ مثل یک کلاه تزئینی زیبا (احتمالاً نام آن هم از همین ایده گرفته شده است).

«دکوراتور» تابعی را که در زیر آن قرار دارد، گرفته و کاری روی آن انجام می‌دهد.

در مثال ما، این دکوراتور به FastAPI اعلام می‌کند که تابع زیر مربوط به مسیر / و با عملیات GET است.

این همان چیزی است که به آن «دکوراتور عملیات مسیر» (path operation decorator) گفته می‌شود.

علاوه بر `@app.get()`، می‌توان از متدهای دیگر استفاده کرد:

```
@app.post()
@app.put()
@app.delete()
```

و متدهای کمتر رایج مانند:

```
@app.options()
@app.head()
@app.patch()
@app.trace()
```

**نکته:** شما آزاد هستید که از هر عملیات (متد HTTP) به دلخواه خود استفاده کنید. FastAPI هیچ معنا یا کاربرد خاصی را تحمیل نمی‌کند. اطلاعات ارائه‌شده در اینجا صرفاً به عنوان یک راهنما مطرح شده‌اند، نه یک الزام. برای مثال، هنگام استفاده از GraphQL، معمولاً تمام عملیات را فقط با استفاده از متد POST انجام می‌دهید.

## گام ۴: تعریف تابع عملیات مسیر

این «تابع عملیات مسیر» ما است:

- مسیر: /
- عملیات: GET
- تابع: `root()` که در زیر دکوراتور `@app.get("/")` قرار دارد

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

این یک تابع پایتونی است که FastAPI هنگام دریافت درخواست GET به / آن را اجرا می‌کند. تابع همزمان یا غیرهمزمان؟ در اینجا از `async def` برای تعریف تابع استفاده شده است، اما می‌توان از `def` نیز استفاده کرد:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    return {"message": "Hello World"}
```



## گام ۵: بازگردانی محتوا

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

می‌توانید یک دیکشنری، لیست، یا مقادیر منفرد مانند `str`، `int` و غیره را بازگردانید. همچنین می‌توانید مدل‌های Pydantic را نیز بازگردانید (در ادامه بیشتر درباره آن خواهید دید). اشیاء و مدل‌های بسیار دیگری نیز وجود دارند که به صورت خودکار به JSON تبدیل می‌شوند (از جمله ORMها و غیره). می‌توانید از موارد دلخواه خود استفاده کنید؛ به احتمال زیاد، قبلاً پشتیبانی می‌شوند.

### خلاصه

- FastAPI را وارد کنید.
- یک نمونه از FastAPI ایجاد کنید.
- یک دکوراتور مسیر مانند `@app.get("/")` اضافه کنید.
- یک تابع عملیات مسیر تعریف کنید.
- سرور توسعه را اجرا کنید: `fastapi dev`

## فصل ۲: پارامترهای مسیر

پارامترهای مسیر (Path Parameters) بخش‌هایی از URL هستند که متغیر محسوب می‌شوند و معمولاً برای شناسایی منابع خاص در یک API استفاده می‌شوند. می‌توانید «پارامترها» یا «متغیرهای» مسیر را با همان نحوی که در رشته‌های قالب‌بندی پایتون استفاده می‌شود، تعریف کنید:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id):
    return {"item_id": item_id}
```

مقدار پارامتر مسیر `item_id` به عنوان آرگومان `item_id` به تابع شما ارسال خواهد شد. پس اگر این مثال را اجرا کنید و به آدرس <http://127.0.0.1:8000/items/foo> بروید، پاسخ زیر را خواهید دید:

```
{"item_id": "foo"}
```

### ۲-۱ پارامترهای مسیر با اعلان نوع

می‌توانید نوع پارامتر مسیر را در تابع خود با استفاده از حاشیه‌نویسی استاندارد پایتون مشخص کنید:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

در این حالت، `item_id` به عنوان یک `int` تعریف شده است.

**نکته:** این کار به شما امکان می‌دهد که درون تابع خود از پشتیبانی ویرایشگر، بررسی خطاها، تکمیل خودکار و غیره بهره‌مند شوید.

### ۲-۲ تبدیل داده‌ها

اگر این مثال را اجرا کنید و در مرورگر به آدرس <http://127.0.0.1:8000/items/3> بروید، پاسخ زیر را خواهید دید:

```
{"item_id": 3}
```

**نکته:** توجه کنید که مقدار دریافت شده (و بازگردانده شده) توسط تابع شما 3 به عنوان یک `int` پایتون است، نه یک رشته "3"! بنابراین، با این اعلان نوع، FastAPI به طور خودکار درخواست را «تجزیه» می کند.

### ۳-۲ اعتبارسنجی داده ها

اما اگر به آدرس <http://127.0.0.1:8000/items/foo> بروید، یک خطای HTTP مناسب مشاهده خواهید کرد:

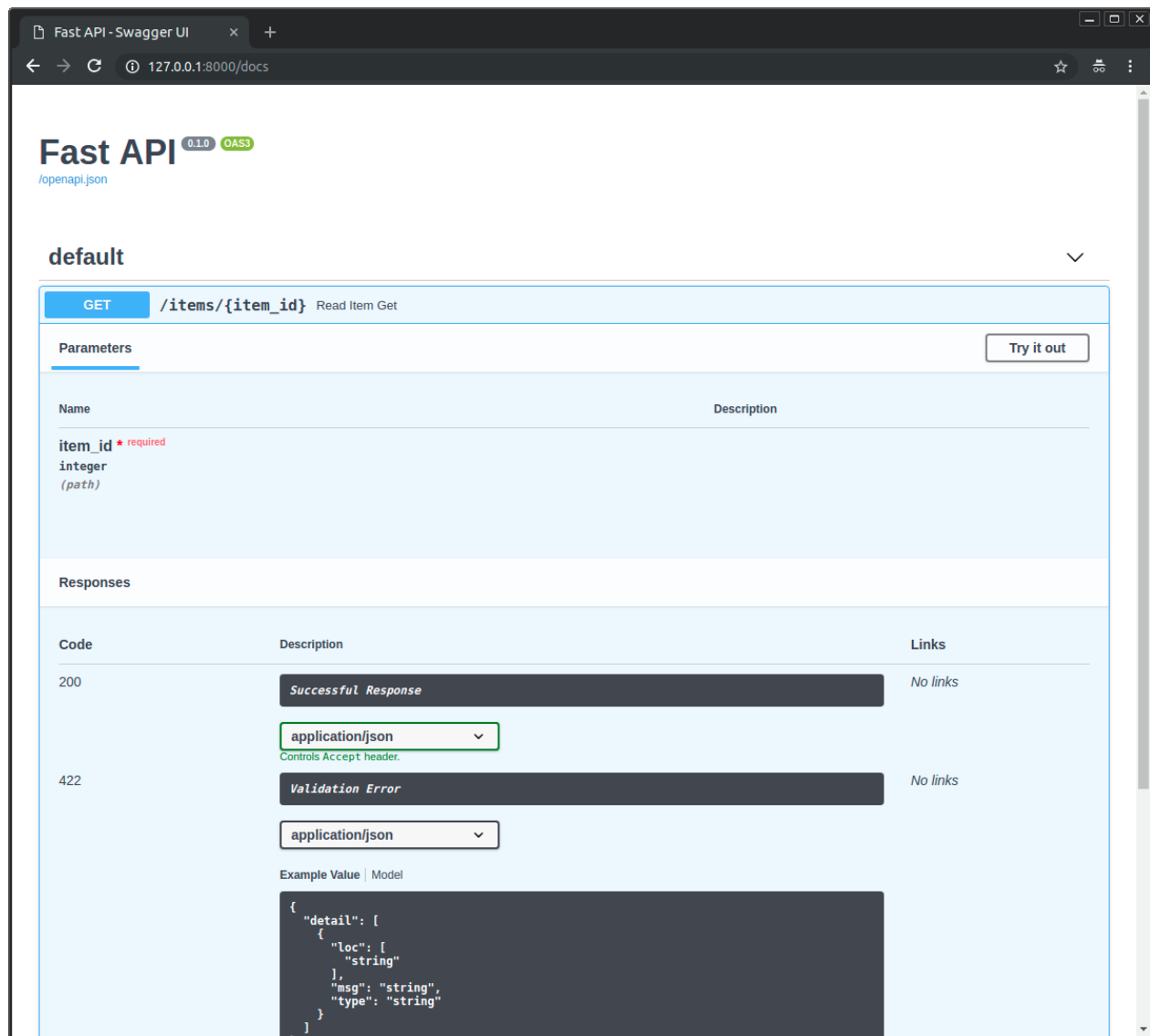
```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": [
        "path",
        "item_id"
      ],
      "msg": "Input should be a valid integer, unable to parse string as an integer",
      "input": "foo",
      "url": "https://errors.pydantic.dev/2.1/v/int_parsing"
    }
  ]
}
```

چون مقدار پارامتر مسیر `item_id` برابر "foo" است که `int` نیست. همین خطا در صورت استفاده از `float` به جای `int` نیز ظاهر می شود، مثلاً در آدرس <http://127.0.0.1:8000/items/4.2>.

**نکته:** به این ترتیب، با همان اعلان نوع پایتون، FastAPI به شما اعتبارسنجی داده ها را ارائه می دهد. همچنین، خطا دقیقاً مشخص می کند که اعتبارسنجی در چه نقطه ای رد شده است. این امر در هنگام توسعه و اشکال زدایی کدهای مربوط به API بسیار مفید است.

### ۴-۲ مستندسازی

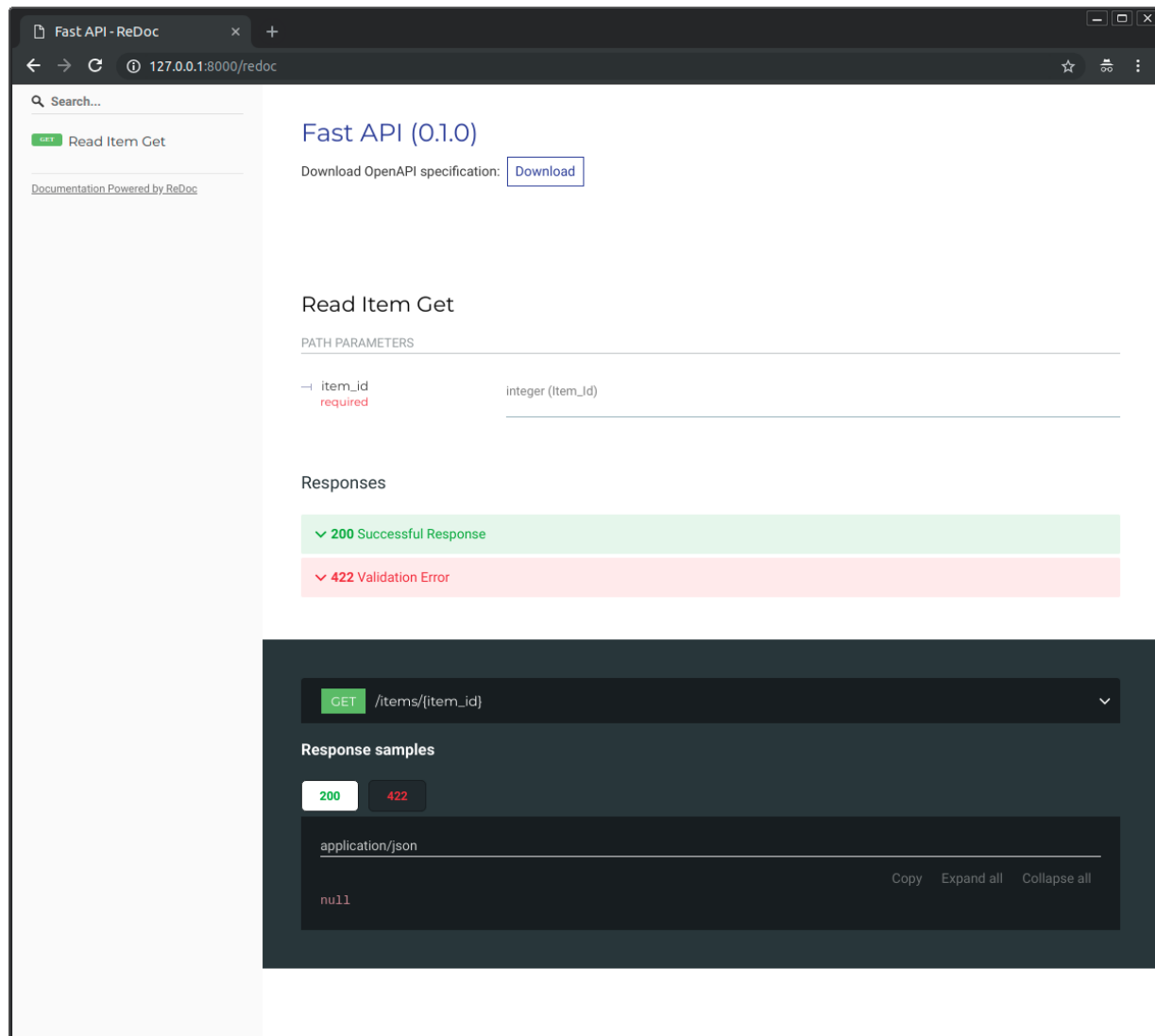
هنگامی که مرورگر خود را به آدرس <http://127.0.0.1:8000/docs> باز کنید، مستندات API به صورت خودکار و تعاملی را مشاهده خواهید کرد که شامل:



**نکته:** با همان اعلان نوع پایتون، FastAPI مستندات خودکار و تعاملی را ارائه می‌دهد که با Swagger UI یکپارچه شده است. توجه کنید که پارامتر مسیر به عنوان یک عدد صحیح تعریف شده است.

## ۵-۲ مزایای مبتنی بر استانداردها و مستندسازی جایگزین

از آنجا که طرح ایجاد شده بر اساس استاندارد OpenAPI است، ابزارهای سازگار متعددی وجود دارند. به همین دلیل، FastAPI مستندات جایگزینی برای API (با استفاده از ReDoc) ارائه می‌دهد که می‌توانید از طریق <http://127.0.0.1:8000/redoc> به آن دسترسی داشته باشید.



به همین ترتیب، ابزارهای سازگار زیادی وجود دارند، از جمله ابزارهای تولید کد برای بسیاری از زبان‌ها.

## ۲-۶ Pydantic

تمام اعتبارسنجی‌های داده‌ای در پشت صحنه توسط Pydantic انجام می‌شوند، بنابراین شما از تمام مزایای آن بهره‌مند می‌شوید. می‌توانید از همان اعلان‌های نوع با `bool`، `float`، `str` و بسیاری از انواع داده‌های پیچیده دیگر استفاده کنید. چندین مورد از این قابلیت‌ها در فصل‌های بعدی این آموزش بررسی خواهند شد.

## ۲-۷ ترتیب مسیرها مهم است

در هنگام ایجاد عملیات مسیر، ممکن است شرایطی پیش بیاید که یک مسیر ثابت داشته باشید، مانند: `/users/me` (مثلاً برای دریافت اطلاعات کاربر فعلی). سپس می‌توانید مسیری مانند `/users/{user_id}` داشته باشید که اطلاعات یک کاربر خاص را بر اساس `user_id` دریافت کند. از آنجا که عملیات مسیر به ترتیب ارزیابی می‌شوند، باید مطمئن شوید که مسیر `/users/me` قبل از

/users/{user\_id} تعریف شده است:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/me")
async def read_user_me():
    return {"user_id": "the current user"}

@app.get("/users/{user_id}")
async def read_user(user_id: str):
    return {"user_id": user_id}
```

در غیر این صورت، مسیر /users/{user\_id} شامل /users/me نیز خواهد شد و مقدار "me" را به عنوان user\_id دریافت خواهد کرد. به طور مشابه، نمی‌توانید یک عملیات مسیر را مجدداً تعریف کنید:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users")
async def read_users():
    return ["Rick", "Morty"]

@app.get("/users")
async def read_users2():
    return ["Bean", "Elfo"]
```

اولین مورد همیشه استفاده خواهد شد زیرا مسیر آن ابتدا مطابقت پیدا می‌کند.

## ۸-۲ مقادیر از پیش تعیین شده

اگر عملیاتی دارید که پارامتر مسیر دریافت می‌کند اما می‌خواهید مقادیر مجاز از پیش تعیین شده باشند، می‌توانید از یک Enum استاندارد پایتون استفاده کنید.

### ۸-۲-۱ ایجاد کلاس Enum

ماژول Enum را وارد کرده و یک زیرکلاس ایجاد کنید که از str و Enum ارث‌بری می‌کند. با ارث‌بری از str، مستندات API قادر خواهند بود تشخیص دهند که مقادیر باید از نوع رشته‌ای باشند و آن‌ها را به‌درستی نمایش دهند. سپس، ویژگی‌های کلاس را با مقادیر ثابت تعریف کنید؛ این مقادیر، همان مقادیر معتبر و مجاز در دسترس خواهند بود.

```
from enum import Enum

from fastapi import FastAPI

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"
```

```

app = FastAPI()

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}

    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}

    return {"model_name": model_name, "message": "Have some residuals"}

```

**نکته:** اگر کنجکاو هستید، باید بگوییم که AlexNet، ResNet و LeNet فقط نام مدل‌های یادگیری ماشینی هستند.

### ۲-۸-۲ اعلان یک پارامتر مسیر

سپس با استفاده از کلاس `enum` که ایجاد کرده‌اید (`ModelName`)، یک پارامتر مسیر با حاشیه‌نویسی نوع، تعریف کنید:

```

from enum import Enum

from fastapi import FastAPI

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"

app = FastAPI()

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}

    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}

    return {"model_name": model_name, "message": "Have some residuals"}

```

### ۲-۸-۳ بررسی مستندات

از آنجایی که مقادیر مجاز برای پارامتر مسیر از پیش تعیین شده‌اند، مستندات تعاملی می‌تواند آن‌ها را به خوبی نمایش دهند. اگر اپلیکیشن FastAPI را اجرا کنید و مرورگر را به این آدرس ببرید: <http://127.0.0.1:8000/docs>

مستندات تعاملی Swagger UI باز خواهد شد. در اینجا، می‌توانید مسیر `/models/{model_name}` را مشاهده کنید و لیستی از مقادیر مجاز (`lenet` و `resnet`، `alexnet`) را ببینید.

FastAPI 0.1.0 OAS3

/openapi.json

default

GET /models/{model\_name} Get Model

Parameters

Cancel

Name	Description
model_name * required	
string (path)	

alexnet  
resnet  
lenet

Execute

Responses

Code	Description	Links
200	Successful Response	No links
	Media type: application/json	
	Controls Accept header.	
	Example Value   Schema	
	(no example available)	
422	Validation Error	No links
	Media type	

#### ۴-۸-۲ کار با Enumeration های پایتون

مقدار پارامتر مسیر یک عضو از کلاس Enumeration خواهد بود.

#### مقایسه اعضای Enumeration

می‌توانید آن را با یکی از اعضای Enum تعریف شده‌تان در کلاس ModelName مقایسه کنید.



```

from enum import Enum
from fastapi import FastAPI

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"

app = FastAPI()

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}

    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}

    return {"model_name": model_name, "message": "Have some residuals"}

```

### دریافت مقدار Enumeration

می‌توانید مقدار واقعی یک عضو از Enum را دریافت کنید (که در این مورد یک str است) با استفاده از `model_name.value` یا به‌طور کلی `your_enum_member.value`.

```

from enum import Enum
from fastapi import FastAPI

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"

app = FastAPI()

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}

    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}

    return {"model_name": model_name, "message": "Have some residuals"}

```

**نکته:** همچنین می‌توانید مقدار خاصی را به‌صورت مستقیم دریافت کنید، مانند: `ModelName.lenet.value`.

### برگرداندن اعضای Enumeration

می‌توانید اعضای Enum را از مسیرهای خود بازگردانید، حتی اگر درون یک بدنه JSON (مثلاً یک دیکشنری) قرار داشته باشند.

FastAPI به طور خودکار اعضای Enum را قبل از ارسال به کلاینت به مقدار متنی (رشته‌ای) آن‌ها تبدیل می‌کند.

```
from enum import Enum
from fastapi import FastAPI

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"

app = FastAPI()

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}

    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}

    return {"model_name": model_name, "message": "Have some residuals"}
```

در سمت کلاینت، پاسخی به صورت JSON دریافت خواهید کرد مانند:

```
{
  "model_name": "alexnet",
  "message": "Deep Learning FTW!"
}
```

## ۹-۲ پارامترهای مسیری که شامل مسیرهای فایل هستند

فرض کنید یک مسیر به صورت `/files/{file_path}` دارید. اما مقدار `file_path` خودش شامل یک مسیر باشد، مثلاً: `.home/johndoe/myfile.txt`. در این صورت، URL مربوطه می‌تواند به این شکل باشد:

```
/files/home/johndoe/myfile.txt
```

### ۹-۲-۱ مشکل در OpenAPI

OpenAPI به طور پیش فرض روشی برای تعریف پارامترهای مسیر که خودشان شامل مسیرهای دیگر هستند ندارد. زیرا چنین قابلیت‌ای می‌تواند منجر به سناریوهایی شود که تست و مدیریت آن‌ها دشوار باشد. با این حال، در FastAPI می‌توان این قابلیت را با یکی از ابزارهای داخلی Starlette پیاده‌سازی کرد. مستندات API همچنان کار خواهد کرد، اما توضیح خاصی درباره اینکه پارامتر باید شامل مسیر باشد ارائه نمی‌شود.

### ۹-۲-۲ تبدیل‌کننده مسیر (Path Converter)

با استفاده از یکی از قابلیت‌های Starlette، می‌توان یک پارامتر مسیر را طوری تعریف کرد که شامل یک مسیر باشد. فرمت URL به این صورت خواهد بود:

```
/files/{file_path:path}
```

در اینجا نام پارامتر `file_path` است و قسمت `path` در انتهای آن مشخص می‌کند که مقدار این پارامتر باید با هر مسیری مطابقت داشته باشد.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/files/{file_path:path}")
async def read_file(file_path: str):
    return {"file_path": file_path}
```

**نکته:** ممکن است به پارامتری نیاز داشته باشید که حاوی مسیر `/home/johndoe/myfile.txt` باشد - با اسلش ابتدایی (/). در این حالت، URL به این شکل خواهد بود:

```
/files//home/johndoe/myfile.txt
```

که در آن بین `files` و `home` دو اسلش پشت سر هم (//) وجود دارد.

### جمع‌بندی

با `FastAPI`، با استفاده از اعلان‌های کوتاه، شهودی و استاندارد پایتون، ویژگی‌های زیر را دریافت می‌کنید:

- پشتیبانی ویرایشگر (بررسی خطاها، تکمیل خودکار و غیره)
- «تجزیه» داده‌ها
- اعتبارسنجی داده‌ها
- مستندسازی خودکار API

و همه این‌ها فقط با یک بار تعریف کردن آن‌ها. این احتمالاً یکی از مهم‌ترین مزایای `FastAPI` نسبت به فریمورک‌های جایگزین است (جدا از عملکرد بالا).

## فصل ۳: پارامترهای پرس و جو

وقتی پارامترهای دیگری را در توابع تعریف می‌کنید که بخشی از پارامترهای مسیر نیستند، این پارامترها به صورت خودکار به عنوان «پارامترهای پرس و جو یا کوئری» (Query Parameters) تفسیر می‌شوند.

```
from fastapi import FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"},
                  {"item_name": "Bar"},
                  {"item_name": "Baz"}]

@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]
```

پارامترهای پرس و جو مجموعه‌ای از جفت‌های کلید-مقدار است که بعد از علامت «؟» در URL می‌آید و با علامت «&» از هم جدا می‌شوند. برای مثال، در URL زیر:

```
http://127.0.0.1:8000/items/?skip=0&limit=10
```

پارامترهای پرس و جو عبارت‌اند از:

- Skip: با مقدار ۰
- Limit: با مقدار ۱۰

از آنجایی که این مقادیر بخشی از URL هستند، «به‌طور طبیعی» از نوع رشته هستند. اما وقتی آن‌ها را با نوع‌های پایتون (مثل مثال بالا، به صورت int) تعریف می‌کنید، به آن نوع تبدیل و بر اساس آن اعتبارسنجی می‌شوند. تمام مراحل که برای پارامترهای مسیر اعمال می‌شدند، برای پارامترهای پرس و جو هم اعمال می‌شوند:

- پشتیبانی و پیشنهاد در ویرایشگر (بدیهی است)
- «تجزیه» داده‌ها
- اعتبارسنجی داده‌ها
- مستندسازی خودکار

### ۳-۱ مقادیر پیش‌فرض

از آنجایی که پارامترهای پرس و جو بخشی ثابت از مسیر نیستند، می‌توانند اختیاری باشند و مقادیر پیش‌فرض داشته باشند. در مثال بالا، مقادیر پیش‌فرض skip=0 و limit=10 هستند. بنابراین، رفتن به URL زیر:

```
http://127.0.0.1:8000/items/
```

معادل است با رفتن به:

```
http://127.0.0.1:8000/items/?skip=0&limit=10
```

اما اگر به آدرس زیر بروید:

```
http://127.0.0.1:8000/items/?skip=20
```

مقادیر پارامترها در تابع به صورت زیر خواهند بود:

- `skip=20`: چون در URL تعیین شده
- `limit=10`: چون مقدار پیش فرض آن است

## ۳-۲ پارامترهای اختیاری

به همان روش، می‌توانید پارامترهای پرس‌وجو اختیاری تعریف کنید، با مقدار پیش فرض `None`:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: str, q: str | None = None):
    if q:
        return {"item_id": item_id, "q": q}
    return {"item_id": item_id}
```

در این حالت، پارامتر `q` اختیاری است و به صورت پیش فرض `None` خواهد بود.

**نکته:** همچنین دقت کنید که `FastAPI` به اندازه کافی هوشمند است که تشخیص دهد `item_id` یک پارامتر مسیر است و `q` پارامتر پرس‌وجو است.

## ۳-۳ تبدیل نوع پارامتر پرس‌وجو

می‌توانید نوع `bool` نیز تعریف کنید و `FastAPI` آن را تبدیل خواهد کرد:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: str, q: str | None = None, short: bool = False):
    item = {"item_id": item_id}
    if q:
```

```

        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long
             description"}
        )
    return item

```

در این حالت، اگر به URL های زیر بروید:

```

http://127.0.0.1:8000/items/foo?short=1
http://127.0.0.1:8000/items/foo?short=True
http://127.0.0.1:8000/items/foo?short=true
http://127.0.0.1:8000/items/foo?short=on
http://127.0.0.1:8000/items/foo?short=yes

```

یا هر حالت دیگری از حروف بزرگ و کوچک، تابع شما پارامتر short را به عنوان مقدار بولی True درک خواهد کرد. در غیر این صورت مقدار آن False خواهد بود.

### ۴-۳ چند پارامتر مسیر و پرس و جو

می‌توانید به صورت هم‌زمان چند پارامتر مسیر و پارامتر پرس و جو تعریف کنید. FastAPI می‌داند کدام یک چیست و نیازی نیست آن‌ها را به ترتیب خاصی تعریف کنید. FastAPI آن‌ها را بر اساس نامشان تشخیص می‌دهد:

```

from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{user_id}/items/{item_id}")
async def read_user_item(
    user_id: int, item_id: str, q: str | None = None, short: bool = False):
    item = {"item_id": item_id, "owner_id": user_id}
    if q:
        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long
             description"}
        )
    return item

```

### ۵-۳ پارامترهای پرس و جو الزامی

وقتی برای پارامترهایی که پارامتر مسیر نیستند (در اینجا، پارامترهای پرس و جو) مقدار پیش فرض مشخص می‌کنید، آن پارامتر الزامی نیست. اگر نمی‌خواهید مقدار خاصی بدهید اما می‌خواهید پارامتر اختیاری باشد، مقدار پیش فرض را None قرار دهید. اما اگر می‌خواهید پارامتر پرس و جو الزامی باشد، کافی است هیچ مقدار پیش فرضی تعیین نکنید:

```

from fastapi import FastAPI

```

```
app = FastAPI()

@app.get("/items/{item_id}")
async def read_user_item(item_id: str, needy: str):
    item = {"item_id": item_id, "needy": needy}
    return item
```

در اینجا پارامتر `needy` یک پارامتر پرس و جو الزامی از نوع رشته است. اگر URL زیر را باز کنید:

```
http://127.0.0.1:8000/items/foo-item
```

و پارامتر الزامی `needy` را اضافه نکنید، با خطای زیر مواجه خواهید شد:

```
{
  "detail": [
    {
      "type": "missing",
      "loc": [
        "query",
        "needy"
      ],
      "msg": "Field required",
      "input": null,
      "url": "https://errors.pydantic.dev/2.1/v/missing"
    }
  ]
}
```

چون `needy` پارامتر الزامی است، باید آن را در URL مشخص کنید:

```
http://127.0.0.1:8000/items/foo-item?needy=sooooneedy
```

این درخواست با موفقیت اجرا خواهد شد:

```
{
  "item_id": "foo-item",
  "needy": "sooooneedy"
}
```

و البته، می‌توانید بعضی از پارامترها را الزامی تعریف کنید، بعضی را با مقدار پیش فرض و بعضی را کاملاً اختیاری:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_user_item(
    item_id: str, needy: str, skip: int = 0, limit: int | None = None
):
    item = {"item_id": item_id, "needy": needy, "skip": skip, "limit": limit}
```

```
return item
```

در این حالت، سه پارامتر پرس و جو داریم:

- `needy`: از نوع `str` و الزامی
- `skip`: از نوع `int` با مقدار پیش فرض صفر
- `limit`: از نوع `int` و اختیاری

**نکته:** شما همچنین می‌توانید از `Enum`ها به همان روشی که با پارامترهای مسیر استفاده می‌کنید، استفاده کنید.



## فصل ۴: بدنه‌ی درخواست

زمانی که نیاز دارید داده‌ای را از سمت کلاینت (مثلاً مرورگر) به API خود ارسال کنید، آن را به صورت بدنه‌ی درخواست (Request Body) ارسال می‌کنید. بدنه درخواست داده‌ای است که از سوی کلاینت به API شما ارسال می‌شود. بدنه‌ی پاسخ داده‌ای است که API شما به کلاینت بازمی‌گرداند.

در اغلب موارد، API شما باید بدنه پاسخ ارائه دهد. اما کلاینت‌ها الزاماً همیشه نیازی به ارسال بدنه درخواست ندارند، گاهی فقط یک مسیر را درخواست می‌کنند، شاید با چند پارامتر پرس‌وجو، اما بدون ارسال بدنه‌ای از داده. برای اعلان یک بدنه درخواست، شما از مدل‌های Pydantic با تمام قدرت و مزایای آن استفاده می‌کنید.

**نکته:** برای ارسال داده، باید از یکی از متدهای: POST (متداول‌ترین)، PUT، DELETE یا PATCH استفاده شود. ارسال بدنه با یک درخواست GET رفتاری تعریف نشده در استانداردها دارد. با این حال، FastAPI از آن پشتیبانی می‌کند، فقط برای موارد بسیار پیچیده یا خاص. از آنجا که این کار توصیه نمی‌شود، مستندات تعاملی با Swagger UI مستندی برای بدنه درخواست هنگام استفاده از GET نشان نمی‌دهند، و پراکسی‌های میانی نیز ممکن است از آن پشتیبانی نکنند.

### ۱-۴ وارد کردن BaseModel از Pydantic

ابتدا باید BaseModel را از pydantic وارد کنید:

```
from pydantic import BaseModel
```

### ۲-۴ ساخت مدل داده

سپس مدل داده‌ی خود را به صورت یک کلاس که از BaseModel ارث‌بری می‌کند، تعریف می‌کنید. از انواع استاندارد پایتون برای تمام ویژگی‌ها استفاده کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    return item
```

همانند تعریف پارامترهای پرس وجو، زمانی که ویژگی‌های مدل مقدار پیش فرض دارند، اجباری نیستند. در غیر این صورت، الزامی هستند. برای اختیاری کردن ویژگی‌ها، از None به عنوان مقدار پیش فرض استفاده کنید. برای نمونه، مدل بالا یک شیء JSON (یا dict در پایتون) مشابه این را تعریف می‌کند:

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

و چون ویژگی‌های description و tax اختیاری هستند (با مقدار پیش فرض None)، این شیء JSON نیز معتبر خواهد بود:

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

### ۳-۴ اعلان به عنوان یک پارامتر

برای افزودن آن به مسیر عملیات خود (path operation)، دقیقاً مانند پارامترهای مسیر یا پرس وجو عمل کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    return item
```

...و نوع آن را مدل تعریف شده (اینجا Item) اعلام کنید.

### ۴-۴ نتایج

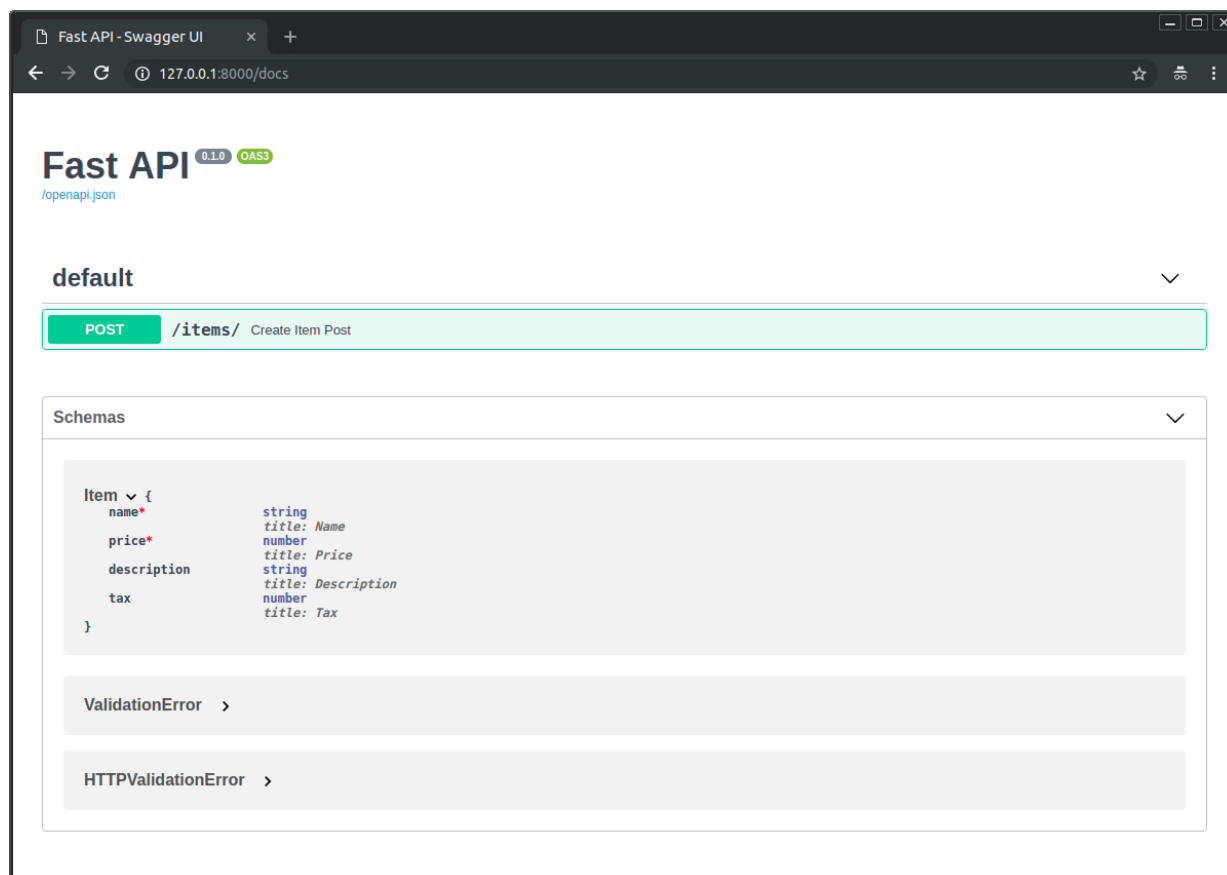
با همین اعلان ساده‌ی نوع در پایتون، FastAPI به صورت خودکار:

- بدنه‌ی درخواست را به صورت JSON می‌خواند.
- انواع داده‌ها را در صورت نیاز تبدیل می‌کند.
- داده را اعتبارسنجی می‌کند.

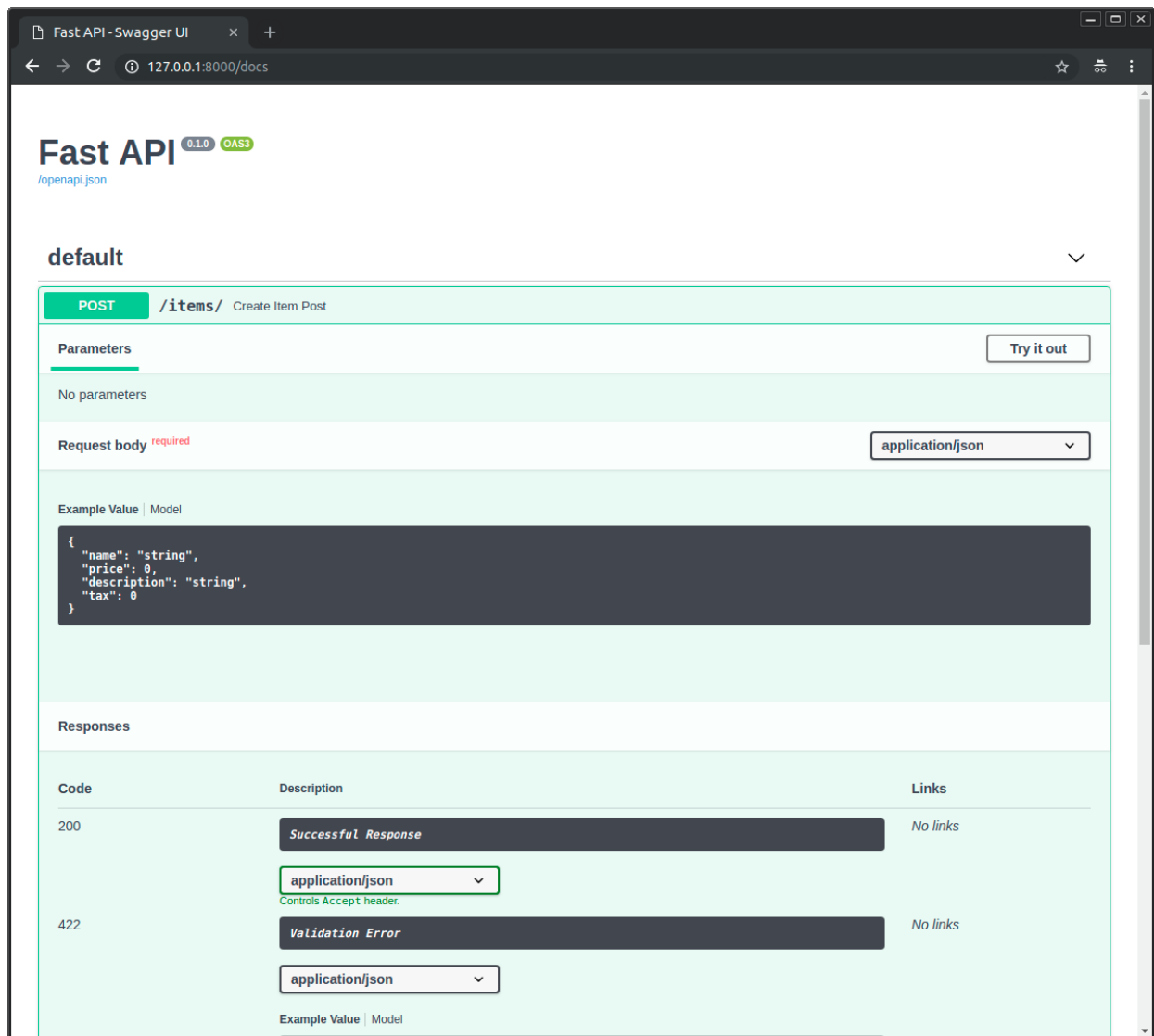
- در صورت نامعتبر بودن داده، یک خطای واضح و دقیق باز می گرداند که مشخص می کند مشکل دقیقاً کجاست.
- داده‌ی دریافت شده را در پارامتر `item` در اختیار شما قرار می دهد.
- چون در تابع نوع آن را `Item` اعلام کرده اید، از پشتیبانی کامل ویرایشگر برای تکمیل خودکار و بررسی نوع‌ها بهره‌مند می شوید.
- تعریف اسکیمای `JSON` برای مدل شما تولید می شود که می توانید در جاهای دیگر نیز از آن استفاده کنید.
- این اسکیمای `OpenAPI` از اسکیمای `OpenAPI` تولید شده خواهند بود و در مستندات خودکار به کار می روند.

## ۵-۴ مستندات خودکار

اسکیمای `JSON` مدل‌های شما بخشی از اسکیمای `OpenAPI` تولید شده خواهند بود و در مستندات `API` تعاملی نمایش داده می شوند:



و همچنین در مستندات `API` داخل هر عملیات مسیر که به آنها نیاز است، استفاده خواهد شد:



## ۶-۴ پشتیبانی ویرایشگر

در ویرایشگر خود، داخل تابع، اعلان نوع‌ها و تکمیل خودکار خواهید داشت (این ویژگی در صورتی که فقط یک dict دریافت می‌کردید فعال نمی‌بود):

```

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

```

You, a few seconds ago • Uncommitted changes

capitalize  
casefold  
center  
count  
encode  
endswith  
expandtabs  
find  
format  
format\_map  
index  
isalnum

str.capitalize(self) ×  
S.capitalize() -> str  
Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

همچنین بررسی خطا برای عملیات‌هایی که نوع نادرستی دارند نیز دریافت می‌کنید:

```

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

```

این موضوع تصادفی نیست. تمام فریمورک FastAPI با این طراحی ساخته شده است. در مرحله‌ی طراحی، پیش از پیاده‌سازی، آزمایش‌های دقیقی برای اطمینان از عملکرد آن با ویرایشگرها انجام شده است. حتی تغییراتی در خود Pydantic برای پشتیبانی از این ویژگی‌ها اعمال شده است.

تصاویر بالا از Visual Studio Code گرفته شده‌اند، اما با PyCharm و بیشتر ویرایشگرهای پایتون نیز همین پشتیبانی را دریافت خواهید کرد:

```

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  @app.post("/items/")
15  async def create_item(item: Item):
16      item.name =
17      return item.name.capitalize()
18
19

```

**نکته:** اگر از PyCharm استفاده می‌کنید، می‌توانید افزونه‌ی Pydantic PyCharm Plugin را نصب کنید. این افزونه پشتیبانی ویرایشگر از مدل‌های Pydantic را بهبود می‌دهد، با ویژگی‌هایی مانند:

- تکمیل خودکار
- بررسی نوع
- بازسازی کد
- جست‌وجو
- بازرسی‌ها

## ۷-۴ استفاده از مدل

درون تابع می‌توانید مستقیماً به تمام ویژگی‌های شیء مدل دسترسی داشته باشید:

```

from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

app = FastAPI()

```

```
@app.post("/items/")
async def create_item(item: Item):
    item_dict = item.dict()
    if item.tax is not None:
        price_with_tax = item.price + item.tax
        item_dict.update({"price_with_tax": price_with_tax})
    return item_dict
```

#### ۸-۴ بدنه‌ی درخواست به همراه پارامترهای مسیر

شما می‌توانید به‌طور هم‌زمان پارامترهای مسیر و بدنه‌ی درخواست را اعلام کنید. FastAPI تشخیص خواهد داد که پارامترهایی که با پارامترهای مسیر هم‌نام هستند باید از مسیر گرفته شوند، و پارامترهایی که از نوع مدل‌های Pydantic هستند باید از بدنه‌ی درخواست گرفته شوند.

```
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

app = FastAPI()

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

#### ۹-۴ بدنه‌ی درخواست + مسیر + پارامترهای پرس‌وجو

شما همچنین می‌توانید هم‌زمان بدنه، مسیر و پارامترهای پرس‌وجو را اعلام کنید. FastAPI هر کدام را شناسایی کرده و داده را از محل مناسب استخراج می‌کند.

```
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

app = FastAPI()

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item, q: str | None = None):
    result = {"item_id": item_id, **item.dict()}
    if q:
        result.update({"q": q})
    return result
```

پارامترهای تابع به این صورت شناسایی می‌شوند:

- اگر پارامتر در مسیر نیز اعلام شده باشد، به عنوان پارامتر مسیر استفاده می‌شود.
- اگر پارامتر از نوع‌های ساده (مثل `int`، `float`، `str`، `bool` و ...) باشد، به عنوان پارامتر پرس‌وجو تفسیر می‌شود.
- اگر پارامتر از نوع مدل `Pydantic` باشد، به عنوان بدنه‌ی درخواست در نظر گرفته می‌شود.

**نکته:** `FastAPI` متوجه می‌شود که مقدار `q` الزامی نیست چون مقدار پیش‌فرض آن `None` است.

عبارت `str | None` (در پایتون ۳٫۱۰+) یا `Union[str, None]` (در پایتون ۳٫۸+) توسط `FastAPI` برای تعیین ضروری یا اختیاری بودن استفاده نمی‌شود، بلکه وجود مقدار پیش‌فرض `None` = تعیین‌کننده است.

با این حال، نوشتن نوع‌ها کمک می‌کند ویرایشگر شما پشتیبانی بهتری ارائه دهد و خطاها را تشخیص دهد.

## ۱۰-۴ بدون `Pydantic`

اگر نمی‌خواهید از مدل‌های `Pydantic` استفاده کنید، می‌توانید از پارامترهای `Body` نیز استفاده کنید که در فصل ۸ بررسی خواهد شد.



## فصل ۵: پارامترهای پرس و جو و اعتبارسنجی رشته‌ای

FastAPI به شما اجازه می‌دهد اطلاعات اضافی و اعتبارسنجی‌هایی برای پارامترهای تان تعریف کنید. به عنوان مثال اجازه دهید این اپلیکیشن را بررسی کنیم:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
async def read_items(q: str | None = None):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

پارامتر پرس و جو `q` از نوع `str | None` است؛ یعنی می‌تواند مقدار رشته‌ای یا `None` داشته باشد. در واقع، مقدار پیش فرض آن `None` است، بنابراین FastAPI متوجه می‌شود که این پارامتر الزامی نیست.

**نکته:** FastAPI به دلیل مقدار پیش فرض برابر با `None` متوجه می‌شود که مقدار `q` اجباری نیست.

داشتن `str | None` باعث می‌شود ویرایشگر شما پشتیبانی بهتری ارائه دهد و خطاها را تشخیص دهد.

### ۵-۱ اعتبارسنجی اضافی

ما قصد داریم این محدودیت را اعمال کنیم که اگرچه پارامتر `q` اختیاری است، اما هر زمان که ارائه شد، طول آن نباید از ۵۰ کاراکتر تجاوز کند. برای رسیدن به این هدف، ابتدا باید این موارد را ایمپورت کنید:

- `fastapi` از `Query`
- `typing` از `Annotated`

```
from typing import Annotated
from fastapi import FastAPI, Query
```

**اطلاعات:** FastAPI از نسخه 0.95.0 پشتیبانی از `Annotated` را اضافه کرده و استفاده از آن را توصیه می‌کند. اگر نسخه قدیمی‌تری دارید، در استفاده از `Annotated` دچار خطا خواهید شد. مطمئن شوید که FastAPI را به حداقل نسخه 0.95.1 به روزرسانی کرده‌اید.

### ۵-۲ استفاده از `Annotated` برای پارامتر `q`

یادتان هست که در «مقدمه‌ای بر نوع‌های پایتون» گفته بودم می‌توانید از Annotated برای افزودن فراداده به پارامترهای تان استفاده کنید؟ حالا وقت آن رسیده که آن را در FastAPI به کار ببرید. ما قبلاً این نوع تایپ را داشتیم:

```
q: str | None = None
```

حالا آن را با Annotated بسته‌بندی می‌کنیم تا به این شکل درآید:

```
q: Annotated[str | None] = None
```

هر دو نسخه یک معنی دارند: پارامتر q می‌تواند مقدار str یا None داشته باشد و مقدار پیش‌فرض آن None است.

### ۳-۵ افزودن Query به Annotated در پارامتر q

حالا که Annotated را داریم و می‌توانیم اطلاعات بیشتری داخل آن بگذاریم (در این مورد اعتبارسنجی بیشتر)، پرس‌وجو را درون Annotated قرار می‌دهیم و پارامتر max\_length را روی ۵۰ تنظیم می‌کنیم:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: Annotated[str | None, Query(max_length=50)] = None):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

توجه کنید که مقدار پیش‌فرض همچنان None است، پس پارامتر همچنان اختیاری باقی می‌ماند. اما حالا که از Query(max\_length=50) در داخل Annotated استفاده کرده‌ایم، به FastAPI گفته‌ایم که برای این مقدار اعتبارسنجی بیشتری انجام بدهد، مثلاً بیشینه طول ۵۰ کاراکتر.

**نکته:** در اینجا از Query() استفاده کرده‌ایم چون این پارامتر از نوع پارامتر پرس‌وجو است. در آینده با گزینه‌هایی مثل Path()، Body()، Header() و Cookie() آشنا خواهید شد که همه پارامترهایی مشابه با Query() را می‌پذیرند.

FastAPI اکنون:

- داده را اعتبارسنجی می‌کند تا حداکثر طول آن ۵۰ کاراکتر باشد
- خطای واضحی به کاربر در صورت نامعتبر بودن داده نشان می‌دهد
- پارامتر را در مستندات OpenAPI (رابط خودکار) ثبت می‌کند

#### ۴-۵ جایگزین (قدیمی): استفاده از Query به عنوان مقدار پیش فرض

نسخه‌های قبلی FastAPI (پیش از 0.95.0) شما را مجبور می‌کردند که به جای Annotated از Query به عنوان مقدار پیش فرض پارامتر استفاده کنید. احتمال دارد هنوز کدی با این ساختار ببینید، پس آن را هم توضیح می‌دهم.

**نکته:** برای کدهای جدید خود و هر جا که ممکن است، از Annotated استفاده کنید. دلایل و مزایای زیادی دارد (که در ادامه گفته شده) و هیچ عیبی ندارد.

این نحوه استفاده از Query() به عنوان مقدار پیش فرض پارامتر تابع است:

```
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: str | None = Query(default=None, max_length=50)):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

در این حالت (بدون Annotated)، به جای None باید مقدار پیش فرض را با Query() جایگزین کنیم و به آن مقدار default=None بدهیم. هدف از این کار تعریف مقدار پیش فرض برای FastAPI است. بنابراین:

```
q: str | None = Query(default=None)
```

همانند:

```
q: str | None = None
```

اما نسخه‌ی Query آن را به صراحت به عنوان پارامتر پرس و جو اعلام می‌کند. سپس می‌توانیم پارامترهای بیشتری به Query بدهیم، مثل max\_length که برای رشته‌ها کاربرد دارد:

```
q: str | None = Query(default=None, max_length=50)
```

این مورد باعث اعتبارسنجی داده‌ها، اعلام خطای واضح در صورت نامعتبر بودن داده، و مستندسازی در OpenAPI می‌شود.

#### ۴-۵-۱ استفاده از Query به عنوان مقدار پیش فرض یا در Annotated

دقت کنید وقتی از Query داخل Annotated استفاده می‌کنید، نمی‌توانید پارامتر default را برای Query بنویسید. بلکه باید مقدار پیش فرض را به صورت مستقیم برای پارامتر تابع قرار دهید. در غیر این صورت ناسازگاری ایجاد می‌شود. مثلاً این کار نادرست است:

```
q: Annotated[str, Query(default="rick")] = "morty"
```

چون مشخص نیست مقدار پیش فرض باید "rick" باشد یا "morty". در عوض از این ساختار استفاده کنید:

```
q: Annotated[str, Query()] = "rick"
```

یا در کدهای قدیمی ممکن است ببینید:

```
q: str = Query(default="rick")
```

## ۲-۴-۵ مزایای Annotated

استفاده از Annotated به جای مقدار پیش فرض تابع توصیه می شود چون دلایل زیادی برای برتری آن وجود دارد:

- مقدار پیش فرض تابع همان مقدار واقعی در پایتون است، که در کل با رفتار پایتون سازگارتر و شهودی تر است
- می توانید همان تابع را در جاهای دیگر بدون FastAPI هم صدا بزنید و همانطور که انتظار دارید عمل کند
- اگر پارامتر الزامی باشد، ویرایشگر به شما هشدار می دهد، و پایتون هم هنگام اجرا خطا خواهد داد
- در روش قدیمی (بدون Annotated)، اگر تابع را خارج از FastAPI صدا بزنید و پارامتر ندهید، خطایی دریافت نمی کنید اما عملکردی که انتظار دارید اتفاق نمی افتد (مثلاً به جای str، شیء QueryInfo دریافت می کنید)
- Annotated می تواند چند فراداده را در خود نگه دارد، پس می توان همان تابع را در ابزارهای دیگر مثل Typer هم استفاده کرد

## ۵-۵ افزودن اعتبارسنجی های بیشتر

می توانید پارامتر min\_length را هم اضافه کنید:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Annotated[str | None, Query(min_length=3, max_length=50)] = None,
):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

## ۶-۵ افزودن عبارات منظم (Regex)

می توانید الگوی عبارت منظم مشخص کنید تا مقدار پارامتر با آن مطابقت داشته باشد:

```

from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Annotated[
        str | None, Query(min_length=3, max_length=50, pattern="^fixedquery$")
    ] = None,):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

```

این الگوی عبارت منظم بررسی می‌کند که مقدار دریافت‌شده:

- $\wedge$ : با این کاراکترها شروع شود، یعنی چیزی قبلش نباشد
- `fixedquery`: دقیقاً این مقدار را داشته باشد
- $\$$ : همین‌جا پایان یابد، یعنی چیزی بعدش نباشد

اگر با «عبارات منظم» آشنایی ندارید، نگران نباشید. برای خیلی‌ها موضوع سختی است. بدون آن هم می‌توانید کارهای زیادی انجام دهید. فقط بدانید هر وقت لازم شد، در FastAPI می‌توانید از آن استفاده کنید.

### ۱-۶-۵ نسخه v1 از Pydantic: regex به جای pattern

پیش از نسخه دوم Pydantic و قبل از FastAPI 0.100.0، پارامتر `pattern` با نام `regex` شناخته می‌شد، اما اکنون منسوخ شده است. ممکن است هنوز کدی با آن ببینید:

```

from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Annotated[
        str | None, Query(min_length=3, max_length=50, regex="^fixedquery$")
    ] = None,):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

```

اما بدانید که این روش منسوخ شده و باید از پارامتر جدید `pattern` استفاده شود.

### ۷-۵ مقادیر پیش‌فرض

البته می‌توانید از مقادیر پیش‌فرضی به‌جز None هم استفاده کنید. فرض کنیم می‌خواهید پارامتر پرس‌وجو q حداقل ۳ کاراکتر داشته باشد و مقدار پیش‌فرض آن هم "fixedquery" باشد:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: Annotated[str, Query(min_length=3)] = "fixedquery"):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

**نکته:** داشتن مقدار پیش‌فرض از هر نوعی (از جمله None) باعث می‌شود پارامتر اختیاری (غیرالزامی) در نظر گرفته شود.

## ۵-۸ پارامترهای الزامی

وقتی نیازی به اعتبارسنجی یا فراداده اضافی نیست، می‌توانید پارامتر q را فقط با حذف مقدار پیش‌فرض، الزامی کنید:

```
q: str
```

به‌جای:

```
q: str | None = None
```

اما وقتی از Query استفاده می‌کنید، مثلاً به این شکل:

```
q: Annotated[str, Query(min_length=3)]
```

این یعنی پارامتر الزامی است چون هیچ مقدار پیش‌فرضی تعریف نشده است.

### ۵-۸-۱ الزامی ولی قابل None

می‌توانید پارامتری را تعریف کنید که مقدار None را بپذیرد ولی همچنان الزامی باشد (یعنی کاربر باید مقدار ارسال کند حتی اگر None باشد):

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: Annotated[str | None, Query(min_length=3)]):
```

```
results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
if q:
    results.update({"q": q})
return results
```

## ۹-۵ لیست پارامترهای پرس و جو / مقادیر چندگانه

وقتی پارامتر پرس و جو را با Query تعریف می کنید، می توانید آن را به صورت لیستی تعریف کنید تا چند مقدار دریافت کند. برای مثال:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: Annotated[list[str] | None, Query()] = None):
    query_items = {"q": q}
    return query_items
```

سپس با آدرسی مثل:

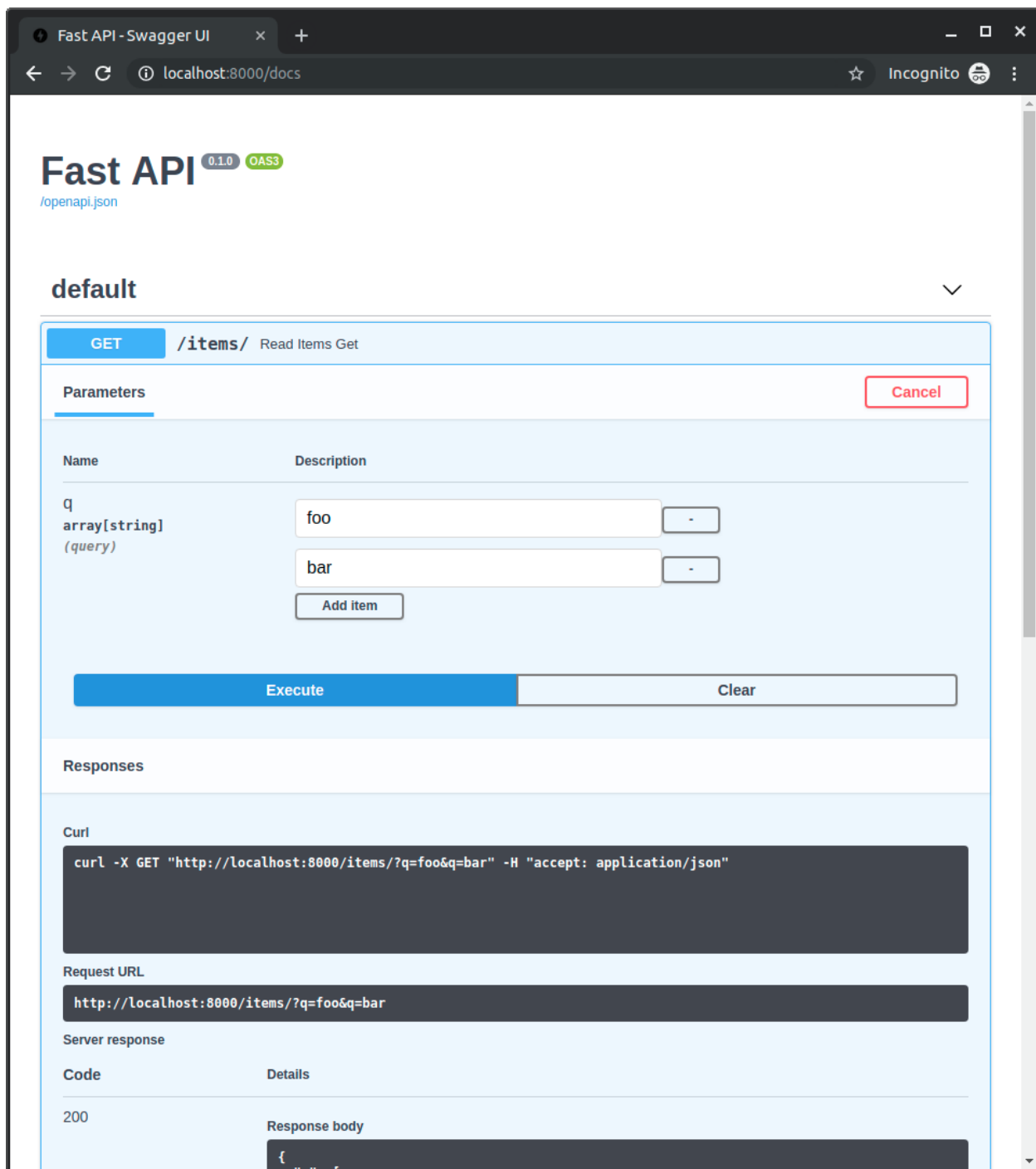
```
http://localhost:8000/items/?q=foo&q=bar
```

شما مقادیر چندگانه پارامتر پرس و جو q (مانند foo و bar) را به صورت یک لیست پایتونی در داخل تابع عملیات مسیر خود و در پارامتر تابع q دریافت خواهید کرد. پاسخ به این URL به شکل زیر خواهد بود:

```
{
  "q": [
    "foo",
    "bar"
  ]
}
```

**نکته:** برای اینکه پارامتر از نوع لیست باشد، باید صراحتاً از Query استفاده کنید. در غیر این صورت، FastAPI آن را به عنوان بدنه‌ی درخواست تفسیر خواهد کرد.

مستندات تعاملی API نیز به طور متناسب به روزرسانی خواهند شد تا امکان وارد کردن مقادیر چندگانه را فراهم کنند:



۱-۵ لیست پارامترهای پرس و جو / مقادیر چندگانه، با مقادیر پیش فرض

شما همچنین می‌توانید یک لیست پیش فرض از مقادیر را تعریف کنید، در صورتی که هیچ مقداری ارائه نشود:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()
```



```
@app.get("/items/")
async def read_items(q: Annotated[list[str], Query()] = ["foo", "bar"]):
    query_items = {"q": q}
    return query_items
```

اگر به این آدرس بروید:

```
http://localhost:8000/items/
```

مقدار پیش فرض `q` برابر با `["foo", "bar"]` خواهد بود و پاسخ به صورت زیر خواهد بود:

```
{
  "q": [
    "foo",
    "bar"
  ]
}
```

### استفاده فقط از `list`

شما می توانید به جای `list[str]`، مستقیماً از `list` استفاده کنید:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: Annotated[list, Query()] = []):
    query_items = {"q": q}
    return query_items
```

**نکته:** به این نکته توجه داشته باشید که در این حالت، `FastAPI` محتوای لیست را بررسی نخواهد کرد. برای مثال، `list[int]` بررسی می کند (و مستند می سازد) که محتوای لیست از نوع عدد صحیح است، اما `list` به تنهایی این کار را انجام نمی دهد.

## ۵-۱۰ تعریف بیشتر فرا-داده (Metadata)

شما می توانید اطلاعات بیشتری درباره پارامتر تعریف کنید. این اطلاعات در `OpenAPI` تولید شده گنجانده می شود و توسط رابط های مستندات و ابزارهای خارجی استفاده می گردد.

**نکته:** به این نکته توجه داشته باشید که ابزارهای مختلف ممکن است از `OpenAPI` در سطوح مختلفی پشتیبانی کنند. برخی از آنها ممکن است هنوز همه اطلاعات اضافی را نمایش ندهند، هرچند که در بیشتر موارد، ویژگی موردنظر در برنامه توسعه قرار دارد.

می‌توانید یک عنوان (title) اضافه کنید:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Annotated[str | None, Query("Query string", min_length=3)] = None,
):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

و یک توضیح (description):

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Annotated[
        str | None,
        Query(
            title="Query string",
            description="Query string for the items to search in the database
                        that have a good match",
            min_length=3,
        ),
    ] = None,
):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

## ۱۱-۵ پارامترهای با نام مستعار (Alias)

فرض کنید می‌خواهید پارامتر شما به شکل item-query باشد. مانند:

```
http://127.0.0.1:8000/items/?item-query=foobaritems
```

اما item-query یک نام معتبر برای متغیر پایتون نیست. نزدیک‌ترین معادل آن item\_query می‌شود، ولی شما همچنان نیاز دارید که دقیقاً همان item-query را داشته باشید...

در این صورت، می‌توانید یک نام مستعار تعریف کنید. این نام مستعار همان چیزی خواهد بود که برای یافتن مقدار پارامتر استفاده می‌شود:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: Annotated[str | None, Query(alias="item-query")] = None):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

## ۱۲-۵ منسوخ کردن پارامترها

حالا فرض کنید دیگر از این پارامتر خوششان نمی‌آید. شما باید آن را برای مدتی نگه دارید، چون برخی کلاینت‌ها از آن استفاده می‌کنند، اما می‌خواهید مستندات آن را به‌وضوح به‌عنوان «منسوخ‌شده» نمایش دهند.

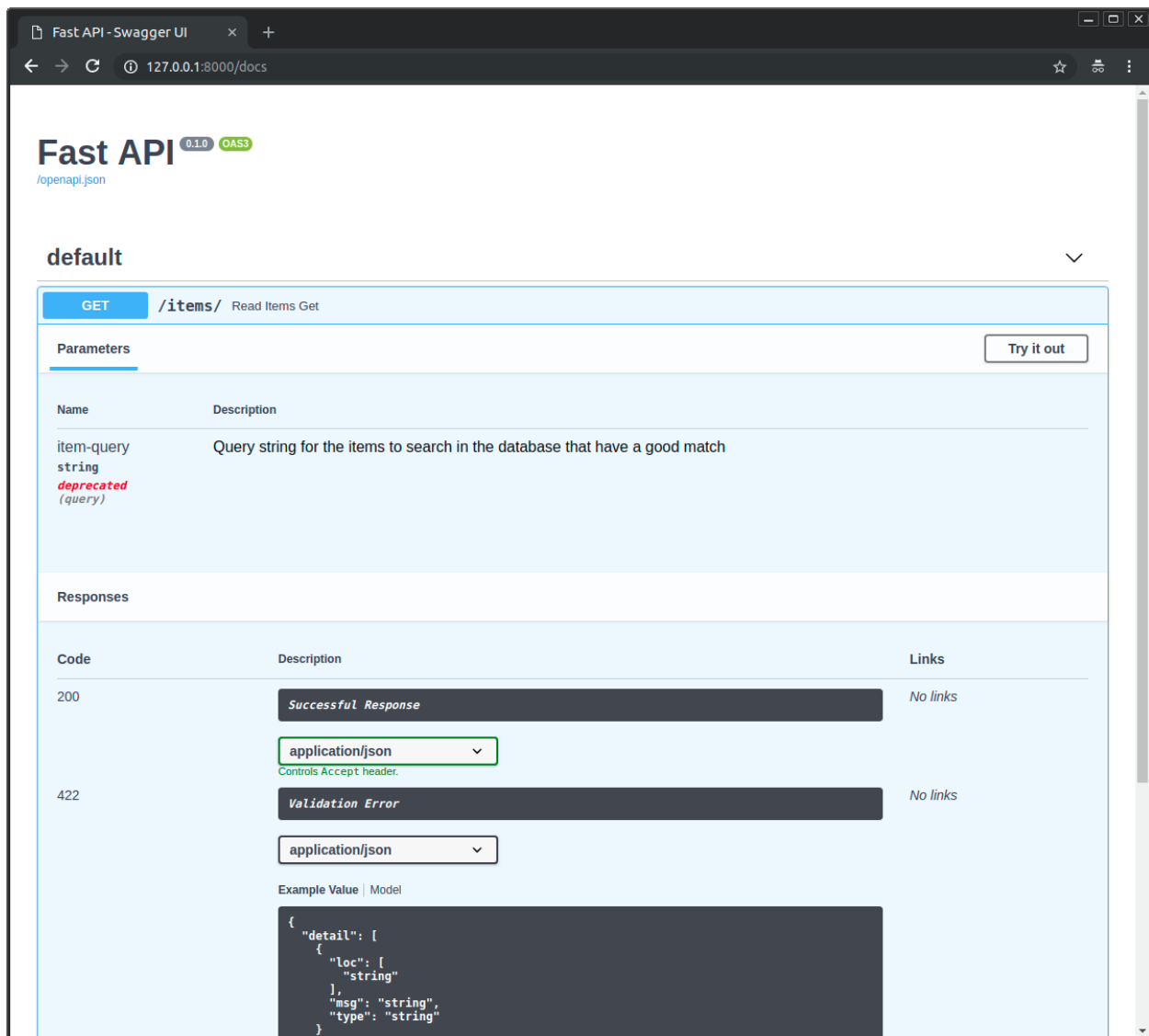
در این صورت، پارامتر `deprecated=True` را به `Query` بدهید:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Annotated[
        str | None,
        Query(
            alias="item-query",
            title="Query string",
            description="Query string for the items to search in the database that have a good match",
            min_length=3,
            max_length=50,
            pattern="^fixedquery$",
            deprecated=True,
        ),
    ] = None,
):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results
```

مستندات آن را به این صورت نمایش خواهند داد:



### ۱۳-۵ حذف پارامتر از OpenAPI

برای اینکه یک پارامتر Query در شمای OpenAPI تولید شده (و در نتیجه، در سیستم مستندسازی خودکار) لحاظ نشود، باید `include_in_schema=False` را به Query بدهید:

```
from typing import Annotated
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    hidden_query: Annotated[str | None, Query(include_in_schema=False)] =
        None,
):
    if hidden_query:
```

```

        return {"hidden_query": hidden_query}
    else:
        return {"hidden_query": "Not found"}

```

#### ۱۴-۵ اعتبارسنجی سفارشی

گاهی ممکن است نیاز به انجام اعتبارسنجی‌هایی داشته باشید که با پارامترهای قبلی امکان‌پذیر نیست. در این موارد، می‌توانید از تابع اعتبارسنجی سفارشی استفاده کنید که پس از اعتبارسنجی معمول (مثلاً بررسی اینکه مقدار از نوع str است) اعمال می‌شود. این کار را می‌توانید با استفاده از AfterValidator از کتابخانه‌ی Pydantic و درون Annotated انجام دهید.

**نکته:** کتابخانه‌ی Pydantic همچنین BeforeValidator و گزینه‌های دیگری نیز دارد.

برای مثال، اعتبارسنجی زیر بررسی می‌کند که شناسه‌ی آیتم با isbn- برای شماره کتاب یا با imdb- برای شناسه‌ی فیلم IMDB شروع شود:

```

import random
from typing import Annotated

from fastapi import FastAPI
from pydantic import AfterValidator

app = FastAPI()

data = {
    "isbn-9781529046137": "The Hitchhiker's Guide to the Galaxy",
    "imdb-tt0371724": "The Hitchhiker's Guide to the Galaxy",
    "isbn-9781439512982": "Isaac Asimov: The Complete Stories, Vol. 2",
}

def check_valid_id(id: str):
    if not id.startswith(("isbn-", "imdb-")):
        raise ValueError('Invalid ID format, it must start with "isbn-" or "imdb-")
    return id

@app.get("/items/")
async def read_items(
    id: Annotated[str | None, AfterValidator(check_valid_id)] = None,
):
    if id:
        item = data.get(id)
    else:
        id, item = random.choice(list(data.items()))
    return {"id": id, "name": item}

```

**اطلاعات:** این قابلیت در نسخه ۲ یا بالاتر از Pydantic در دسترس است.

**نکته:** اگر نیاز دارید اعتبارسنجی‌ای انجام دهید که به ارتباط با مؤلفه‌های خارجی (مثل پایگاه داده یا API دیگر) نیاز دارد، باید از وابستگی‌های FastAPI استفاده کنید. بعداً درباره آن‌ها خواهیم آموخت. این اعتبارسنجی‌های سفارشی فقط برای بررسی‌هایی هستند که می‌توان صرفاً با استفاده از داده‌های ارسال‌شده در درخواست انجام داد.

### ۵-۱۴-۱ درک این کد

نکته‌ی مهم استفاده از `AfterValidator` همراه با یک تابع درون `Annotated` است. می‌توانید از این بخش صرف‌نظر کنید. اما اگر درباره‌ی این مثال خاص کنجکاو هستید، در اینجا چند توضیح اضافه آمده:

### ۵-۱۴-۲ رشته با `value.startswith()`

متوجه شدید؟ در رشته، می‌توان از `value.startswith()` همراه با یک تاپل استفاده کرد، و این تابع، هر مقدار موجود در تاپل را بررسی می‌کند:

```
def check_valid_id(id: str):
    if not id.startswith(("isbn-", "imdb-")):
        raise ValueError('Invalid ID format, it must start with "isbn-" or "imdb-")
    return id
```

### ۵-۱۴-۳ یک آیتیم تصادفی

با `data.items()` یک شیء قابل تکرار از تاپل‌هایی دریافت می‌کنیم که شامل کلید و مقدار هر عضو دیکشنری هستند. ما این شیء را با `list(data.items())` به فهرستی کامل تبدیل می‌کنیم. سپس با `random.choice()` یکی از آیتیم‌های این فهرست را انتخاب می‌کنیم. در نتیجه، یک تاپل به صورت `(id, name)` دریافت می‌کنیم. چیزی شبیه:

```
("imdb-tt0371724", "The Hitchhiker's Guide to the Galaxy")
```

سپس این دو مقدار تاپل را به متغیرهای `id` و `name` اختصاص می‌دهیم. بنابراین اگر کاربر شناسه‌ی آیتیمی ارائه نکند، همچنان یک پیشنهاد تصادفی دریافت خواهد کرد. و همه‌ی این کارها را تنها در یک خط ساده انجام می‌دهیم.

```
@app.get("/items/")
async def read_items(
    id: Annotated[str | None, AfterValidator(check_valid_id)] = None,
):
    if id:
        item = data.get(id)
    else:
        id, item = random.choice(list(data.items()))
    return {"id": id, "name": item}
```

## خلاصه

شما می‌توانید اعتبارسنجی‌های اضافی و فرا-داده برای پارامترهای خود تعریف کنید.

اعتبارسنجی‌های عمومی و فرا-داده:

- alias •
- title •
- description •
- deprecated •

اعتبارسنجی‌های خاص برای رشته‌ها:

- min\_length •
- max\_length •
- pattern •

اعتبارسنجی سفارشی با استفاده از `AfterValidator`

در این مثال‌ها، دیدید که چگونه اعتبارسنجی‌هایی برای مقادیر `str` تعریف کنیم. برای آشنایی با اعتبارسنجی انواع دیگر مثل اعداد، به فصل‌های بعدی مراجعه کنید.

## فصل ۶: پارامترهای مسیر و اعتبارسنجی‌های عددی

همان‌طور که می‌توانید با استفاده از Query اعتبارسنجی‌ها و فراداده‌های بیشتری برای پارامترهای پرس‌وجو تعریف کنید، می‌توانید همان نوع اعتبارسنجی و فراداده را با استفاده از Path برای پارامترهای مسیر نیز تعریف کنید.

### ۶-۱ وارد کردن Path

ابتدا باید Path را از fastapi و همچنین Annotated را وارد کنید:

```
from typing import Annotated
from fastapi import Path
```

**نکته:** FastAPI از نسخه 0.95.0 پشتیبانی از Annotated را اضافه کرد (و استفاده از آن را توصیه می‌کند). اگر از نسخه‌های قدیمی‌تر استفاده می‌کنید، هنگام استفاده از Annotated با خطا مواجه خواهید شد. اطمینان حاصل کنید که نسخه‌ی FastAPI شما حداقل 0.95.1 باشد تا بتوانید از Annotated استفاده کنید.

### ۶-۲ تعریف فراداده (Metadata)

می‌توانید دقیقاً همان پارامترهایی که برای Query قابل تعریف هستند را برای Path نیز تعریف کنید. برای مثال، برای افزودن عنوان به پارامتر مسیر item\_id می‌توانید بنویسید:

```
from typing import Annotated
from fastapi import FastAPI, Path, Query

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    item_id: Annotated[int, Path(title="The ID of the item to get")],
    q: Annotated[str | None, Query(alias="item-query")] = None,
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

**نکته:** پارامترهای مسیر همیشه اجباری هستند، زیرا باید بخشی از مسیر URL باشند. حتی اگر مقدار پیش‌فرض برای آن‌ها تعریف کنید یا None تعیین کنید، باز هم اجباری خواهند بود.

### ۶-۳ ترتیب پارامترها بر اساس نیاز



**نکته:** اگر از Annotated استفاده کنید، احتمالاً نیازی به این بخش نخواهید داشت.

فرض کنید می‌خواهید پارامتر `q` را به عنوان یک رشته‌ی اجباری تعریف کنید. نیازی به تعریف خاصی برای آن با Query ندارید، ولی برای `item_id` باید از Path استفاده کنید. و به هر دلیلی نمی‌خواهید از Annotated استفاده کنید. در این صورت، اگر پارامتری که مقدار پیش‌فرض دارد (مثل `item_id`) را قبل از پارامتر بدون پیش‌فرض (مثل `q`) قرار دهید، پایتون به شما اخطار خواهد داد. اما می‌توانید ترتیب آن‌ها را تغییر دهید و ابتدا پارامتر بدون پیش‌فرض را قرار دهید.

FastAPI هیچ مشکلی با ترتیب ندارد. پارامترها را بر اساس نام، نوع و تعریفشان (مثل Path, Query و غیره) تشخیص می‌دهد و به ترتیب آن‌ها اهمیت نمی‌دهد.

**نکته:** در صورت امکان، استفاده از Annotated توصیه می‌شود.

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    q: str, item_id: int = Path(title="The ID of the item to get")):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

اما اگر از Annotated استفاده کنید، چنین مشکلی نخواهید داشت، چون مقدار پیش‌فرض برای پارامترها تعیین نمی‌کند:

```
from typing import Annotated
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    q: str, item_id: Annotated[int, Path(title="The ID of the item to get")]
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

## ۴-۶ ترتیب پارامترها: ترفندها

**نکته:** اگر از Annotated استفاده می‌کنید، احتمالاً به این ترفند نیازی نخواهید داشت.

اگر بخواهید:

- پارامتر پرس و جوی `q` را بدون استفاده از `Query` و بدون مقدار پیش فرض تعریف کنید
- پارامتر مسیر `item_id` را با استفاده از `Path` تعریف کنید
- ترتیب آن‌ها را دلخواه تنظیم کنید
- از `Annotated` استفاده نکنید

پایتون یک نحو خاص برای این وضعیت دارد:

با قراردادن `*` به عنوان اولین پارامتر تابع، به پایتون می‌فهمانید که تمام پارامترهای بعدی باید به صورت آرگومان‌های کلیدی (جفت‌های کلید-مقدار) فراخوانی شوند؛ که به آن‌ها `kwargs` نیز گفته می‌شود — حتی اگر آن پارامترها مقدار پیش فرض نداشته باشند.

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(*, item_id: int = Path(title="The ID of the item to get"), q: str):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

### ۱-۴-۶ بهتر با `Annotated`

اگر از `Annotated` استفاده می‌کنید، نیازی به مقدار پیش فرض در پارامترها ندارید، بنابراین این مشکل وجود نخواهد داشت و نیازی به `*` نیز نخواهید داشت:

```
from typing import Annotated
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    item_id: Annotated[int, Path(title="The ID of the item to get")], q: str
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

### ۵-۶ اعتبارسنجی عددی: بزرگ‌تر یا مساوی (`ge`)

با استفاده از Query و Path (و سایر مواردی که بعداً خواهید دید)، می‌توانید محدودیت‌های عددی تعریف کنید. در مثال زیر با `ge=1`، مقدار `item_id` باید عددی صحیح و بزرگ‌تر یا مساوی ۱ باشد:

```
from typing import Annotated
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    item_id: Annotated[int, Path(title="The ID of the item to get", ge=1)],
    q: str
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

## ۶-۶ اعتبارسنجی عددی: بزرگ‌تر (gt) و کوچک‌تر یا مساوی (le)

همین مفهوم برای موارد زیر نیز صدق می‌کند:

- `gt`: بزرگ‌تر از
- `le`: کوچک‌تر یا مساوی

```
from typing import Annotated
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    item_id: Annotated[int, Path(title="The ID of the item to get", gt=0,
                                le=1000)],
    q: str,
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

## ۶-۷ اعتبارسنجی عددی برای اعداد اعشاری، بزرگ‌تر و کوچک‌تر

اعتبارسنجی عددی برای اعداد اعشاری نیز کاربرد دارد. در اینجا اهمیت استفاده از `gt` (و نه فقط `ge`) مشخص می‌شود؛ زیرا می‌توانید الزام کنید که مقدار باید بزرگ‌تر از صفر باشد، حتی اگر کمتر از ۱ باشد. مثلاً مقدار ۰٫۵ مجاز است، ولی ۰ یا ۰٫۰۰ مجاز نیستند. همین امر برای `lt` نیز صادق است.

```
from typing import Annotated
```

```

from fastapi import FastAPI, Path, Query

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    *,
    item_id: Annotated[int, Path(title="The ID of the item to get", ge=0,
                                le=1000)],
    q: str,
    size: Annotated[float, Query(gt=0, lt=10.5)],
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    if size:
        results.update({"size": size})
    return results

```

### جمع‌بندی

با Path، Query (و سایر ابزارهایی که بعداً خواهید دید)، می‌توانید همانند اعتبارسنجی رشته‌ها، فراداده و اعتبارسنجی عددی نیز تعریف کنید:

- gt: بزرگ‌تر از
- ge: بزرگ‌تر یا مساوی
- lt: کوچک‌تر از
- le: کوچک‌تر یا مساوی

**نکته:** کلاس‌های Path، Query و دیگر موارد مشابه، همگی زیرکلاس‌هایی از کلاس مشترکی به نام Param هستند. تمام این کلاس‌ها از همان پارامترها برای اعتبارسنجی و فراداده‌ای که تاکنون دیده‌اید پشتیبانی می‌کنند.

**جزئیات فنی:** هنگامی که Path، Query و سایر موارد را از fastapi وارد می‌کنید، در واقع توابعی را وارد می‌کنید. این توابع زمانی که فراخوانی می‌شوند، نمونه‌ای از کلاس‌هایی با همان نام را بازمی‌گردانند. مثلاً شما Query را وارد می‌کنید که یک تابع است، و زمانی که آن را فراخوانی می‌کنید، یک نمونه از کلاسی به نام Query برمی‌گرداند. علت استفاده از توابع (به جای استفاده مستقیم از کلاس‌ها) این است که ویرایشگر شما درباره نوع آن‌ها هشدار خطا ندهد. به این ترتیب، می‌توانید از ویرایشگرها و ابزارهای توسعه‌ی عادی استفاده کنید بدون اینکه نیاز باشد تنظیمات خاصی برای نادیده گرفتن این خطاها انجام دهید.

## فصل ۷: مدل‌های پارامتر پرس‌وجو

اگر گروهی از پارامترهای پرس‌وجو دارید که به هم مرتبط هستند، می‌توانید برای آن‌ها یک مدل Pydantic تعریف کنید. این کار به شما اجازه می‌دهد تا آن مدل را در بخش‌های مختلفی از برنامه‌تان دوباره استفاده کرده و همچنین اعتبارسنجی‌ها (validation) و فراداده‌های (metadata) مربوط به تمام پارامترها را به صورت یک‌جا تعریف کنید.

**نکته:** این ویژگی در FastAPI از نسخه ۰.۱۱۵.۰ به بالا پشتیبانی می‌شود.

### ۷-۱ پارامترهای پرس‌وجو با یک مدل Pydantic

پارامترهای پرس‌وجو مورد نیاز را در یک مدل Pydantic تعریف کنید، سپس پارامتر را به عنوان Query اعلان کنید:

```
from typing import Annotated, Literal
from fastapi import FastAPI, Query
from pydantic import BaseModel, Field

app = FastAPI()

class FilterParams(BaseModel):
    limit: int = Field(100, gt=0, le=100)
    offset: int = Field(0, ge=0)
    order_by: Literal["created_at", "updated_at"] = "created_at"
    tags: list[str] = []

@app.get("/items/")
async def read_items(filter_query: Annotated[FilterParams, Query()]):
    return filter_query
```

FastAPI داده‌های مربوط به هر فیلد را از پارامترهای پرس‌وجو موجود در درخواست استخراج می‌کند و مدل Pydantic تعریف شده را به شما تحویل می‌دهد.

### ۷-۲ بررسی در مستندات

شما می‌توانید پارامترهای پرس‌وجو را در رابط کاربری مستندات موجود در مسیر `/docs` مشاهده کنید:

default

GET /items/ Read Items

Parameters

Name	Description
limit integer (query)	100 maximum: 100
offset integer (query)	0 minimum: 0
order_by string (query)	created_at
tags array[string] (query)	Add string item

Servers

### ۳-۷ محدود کردن پارامترهای اضافی پرس و جو

در برخی کاربردهای خاص (که احتمالاً چندان رایج نیستند)، ممکن است بخواهید پارامترهای پرس و جو ورودی را محدود به مواردی خاص کنید. می‌توانید با استفاده از تنظیمات مدل در Pydantic از دریافت فیلدهای اضافی جلوگیری کنید:

```
from typing import Annotated, Literal
from fastapi import FastAPI, Query
from pydantic import BaseModel, Field

app = FastAPI()

class FilterParams(BaseModel):
    model_config = {"extra": "forbid"}

    limit: int = Field(100, gt=0, le=100)
    offset: int = Field(0, ge=0)
    order_by: Literal["created_at", "updated_at"] = "created_at"
    tags: list[str] = []

@app.get("/items/")
async def read_items(filter_query: Annotated[FilterParams, Query()]):
    return filter_query
```

اگر کلاینت تلاش کند تا پارامتر اضافی‌ای در پرس و جو ارسال کند، با پاسخ خطا مواجه خواهد شد. برای مثال، اگر کلاینت پارامتر tool را با مقدار plumbus ارسال کند، مانند زیر:

```
https://example.com/items/?limit=10&tool=plumbus
```

پاسخی حاوی خطا دریافت خواهد کرد که به او می‌گوید پارامتر tool مجاز نیست:

```
{
  "detail": [
    {
      "type": "extra_forbidden",
      "loc": ["query", "tool"],
      "msg": "Extra inputs are not permitted",
      "input": "plumbus"
    }
  ]
}
```

### خلاصه

می‌توانید برای تعریف پارامترهای پرس‌وجو در FastAPI از مدل‌های Pydantic استفاده کنید.

**نکته:** شما می‌توانید از مدل‌های Pydantic برای تعریف کوکی‌ها (cookies) و هدرها (headers) نیز استفاده کنید، اما در ادامه‌ی این آموزش درباره‌ی آنها خواهید خواند.

## فصل ۸: بدنه – چندین پارامتر

اکنون که دیدیم چطور می‌توان از Path و Query استفاده کرد، بیاید استفاده‌های پیشرفته‌تری از تعریف بدنه‌ی درخواست را بررسی کنیم.

### ۸-۱ ترکیب پارامترهای Path، Query و بدنه

اول از همه، می‌توانید پارامترهای Path، Query و بدنه‌ی درخواست را به‌طور آزادانه ترکیب کنید و FastAPI تشخیص خواهد داد که چه کاری باید انجام دهد. همچنین می‌توانید پارامترهای بدنه را به‌صورت اختیاری تعریف کنید، با قرار دادن مقدار پیش‌فرض برابر با None:

```
from typing import Annotated
from fastapi import FastAPI, Path
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

@app.put("/items/{item_id}")
async def update_item(
    item_id: Annotated[int, Path(title="The ID of the item to get", ge=0,
                                  le=1000)],
    q: str | None = None,
    item: Item | None = None,
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    if item:
        results.update({"item": item})
    return results
```

**نکته:** توجه داشته باشید که در این حالت، آیتمی که از بدنه دریافت می‌شود اختیاری است، زیرا مقدار پیش‌فرض آن None است.

### ۸-۲ پارامترهای چندگانه‌ی بدنه

در مثال قبلی، عملیات مسیر انتظار دارد که یک بدنه‌ی JSON با ویژگی‌های آیتم به شکل زیر دریافت کند:

```
{
  "name": "Foo",
  "description": "The pretender",
```



```

    "price": 42.0,
    "tax": 3.2
}

```

اما می‌توانید چندین پارامتر بدنه تعریف کنید، مثلاً `item` و `user`:

```

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

class User(BaseModel):
    username: str
    full_name: str | None = None

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item, user: User):
    results = {"item_id": item_id, "item": item, "user": user}
    return results

```

در این حالت، FastAPI متوجه خواهد شد که بیش از یک پارامتر بدنه وجود دارد (دو پارامتر که مدل Pydantic هستند). بنابراین از نام پارامترها به عنوان کلید در بدنه استفاده خواهد کرد و بدنه‌ای مشابه زیر را انتظار دارد:

```

{
  "item": {
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2
  },
  "user": {
    "username": "dave",
    "full_name": "Dave Grohl"
  }
}

```

**نکته:** توجه داشته باشید که حتی اگر `item` دقیقاً مانند قبل تعریف شده باشد، اکنون انتظار می‌رود در داخل کلیدی به نام `item` در بدنه قرار گیرد.

FastAPI به‌طور خودکار تبدیل‌های لازم را انجام می‌دهد تا محتوای مورد نظر را به پارامتر `item` تخصیص دهد و همین‌طور برای `user`. همچنین اعتبارسنجی داده‌های ترکیبی را انجام می‌دهد و آن را برای مستندات OpenAPI به همین صورت نمایش می‌دهد.

### ۳-۸ مقادیر تکی در بدنه

همان‌طور که برای تعریف اطلاعات اضافی در پارامترهای پرس‌وجو و مسیر از Query و Path استفاده می‌کنید، FastAPI معادل آن را برای بدنه با Body فراهم می‌کند. برای مثال، فرض کنید می‌خواهید یک کلید دیگر به نام importance نیز در همان بدنه، کنار item و user قرار دهید. اگر آن را بدون چیزی تعریف کنید، چون مقدار تکی است، FastAPI آن را به‌صورت پیش‌فرض به‌عنوان پارامتر پرس‌وجو در نظر می‌گیرد. اما می‌توانید FastAPI را مجبور کنید که آن را به‌عنوان کلیدی در بدنه در نظر بگیرد، با استفاده از Body:

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

class User(BaseModel):
    username: str
    full_name: str | None = None

@app.put("/items/{item_id}")
async def update_item(
    item_id: int, item: Item, user: User, importance: Annotated[int, Body()]
):
    results = {"item_id": item_id, "item": item, "user": user, "importance":
        importance}
    return results
```

در این حالت FastAPI چنین بدنه‌ای را انتظار دارد:

```
{
  "item": {
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2
  },
  "user": {
    "username": "dave",
    "full_name": "Dave Grohl"
  },
  "importance": 5
}
```

دوباره، تبدیل نوع داده، اعتبارسنجی، مستندسازی و غیره انجام خواهد شد.

#### ۴-۸ پارامترهای بدنه‌ی چندگانه و پرس‌وجو

البته، هر زمان که بخواهید می‌توانید علاوه بر پارامترهای بدنه، پارامترهای پرس‌وجو نیز تعریف کنید. از آن‌جا که به‌طور پیش‌فرض مقادیر تکی به‌عنوان پارامتر پرس‌وجو تفسیر می‌شوند، لازم نیست حتماً از Query استفاده کنید. می‌توانید به‌سادگی بنویسید:

```
q: Union[str, None] = None
```

یا در پایتون ۳.۱۰ و بالاتر:

```
q: str | None = None
```

برای مثال:

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

class User(BaseModel):
    username: str
    full_name: str | None = None

@app.put("/items/{item_id}")
async def update_item(
    *,
    item_id: int,
    item: Item,
    user: User,
    importance: Annotated[int, Body(gt=0)],
    q: str | None = None,
):
    results = {"item_id": item_id, "item": item, "user": user, "importance":
        importance}
    if q:
        results.update({"q": q})
    return results
```

**اطلاعات:** Body نیز مانند Query و Path و سایر موارد، تمام پارامترهای اعتبارسنجی و فراداده‌ی اضافی را پشتیبانی می‌کند.

#### ۵-۸ قراردادن پارامتر تکی در کلید بدنه

فرض کنیم فقط یک پارامتر بدنه به نام `item` دارید که یک مدل `Pydantic` است. به طور پیش فرض، `FastAPI` مستقیماً انتظار دارد محتوای آن مدل در بدنه قرار بگیرد. اما اگر بخواهید یک `JSON` داشته باشید که کلیدی به نام `item` داشته باشد و محتوای مدل درون آن قرار گیرد، همان طور که در زمانی که پارامترهای بدنه ی بیشتری تعریف کرده اید اتفاق می افتد، می توانید از پارامتر ویژه `embed` استفاده کنید:

```
item: Item = Body(embed=True)
```

مثلاً:

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Annotated[Item, Body(embed=True)]):
    results = {"item_id": item_id, "item": item}
    return results
```

در این حالت، `FastAPI` چنین بدنه ای را انتظار دارد:

```
{
  "item": {
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2
  }
}
```

در مقابل این ساختار پیش فرض:

```
{
  "name": "Foo",
  "description": "The pretender",
  "price": 42.0,
  "tax": 3.2
}
```

جمع بندی

شما می‌توانید چندین پارامتر بدنه به تابع عملیات مسیر خود اضافه کنید، حتی اگر فقط یک بدنه‌ی درخواست مجاز باشد. اما FastAPI آن را مدیریت کرده و داده‌ها را به‌درستی به شما خواهد داد، اعتبارسنجی می‌کند و ساختار درست را در مستندات مسیر ارائه می‌دهد. همچنین می‌توانید مقادیر تکی را نیز به‌عنوان بخشی از بدنه دریافت کنید. و می‌توانید به FastAPI بگویید که حتی در حالتی که فقط یک پارامتر تعریف شده، آن را درون کلیدی در بدنه قرار دهد.

## فصل ۹: بدنه – فیلدها

همان‌طور که می‌توانید اعتبارسنجی‌های اضافی و فراداده‌ها را در پارامترهای تابع عملیات مسیر با استفاده از Query، Path و Body اعلام کنید، می‌توانید همین کار را برای ویژگی‌های مدل‌های Pydantic با استفاده از Field در خود Pydantic انجام دهید.

### ۹-۱ وارد کردن Field

ابتدا باید آن را وارد کنید:

```
from pydantic import Field
```

**هشدار:** توجه داشته باشید که Field مستقیماً از پکیج pydantic وارد می‌شود، نه از fastapi (برخلاف مواردی مثل Query، Body، Path و غیره که از fastapi وارد می‌شوند).

### ۹-۲ اعلام ویژگی‌های مدل

سپس می‌توانید از Field برای ویژگی‌های مدل استفاده کنید:

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = Field(
        default=None, title="The description of the item", max_length=300
    )
    price: float = Field(gt=0, description="The price must be greater than zero")
    tax: float | None = None

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Annotated[Item, Body(embed=True)]):
    results = {"item_id": item_id, "item": item}
    return results
```

Field دقیقاً مشابه Query، Path و Body عمل می‌کند و همه پارامترهای مشابه را پشتیبانی می‌کند.

**جزئیات فنی:** در واقع، Query، Path و سایر مواردی که در ادامه می‌بینید، شی‌هایی از کلاس‌های فرعی یک کلاس مشترک به نام Param هستند که خودش زیر کلاسی از FieldInfo در Pydantic است. همچنین، تابع Field در Pydantic نیز نمونه‌ای از کلاس FieldInfo را برمی‌گرداند. Body نیز اشیائی از یک زیر کلاس FieldInfo باز می‌گرداند. و دیگر مواردی که بعداً

می‌بینید نیز زیر کلاس‌هایی از کلاس Body هستند. به یاد داشته باشید وقتی که Path، Query و سایر موارد را از fastapi وارد می‌کنید، در واقع آن‌ها توابعی هستند که کلاس‌های خاصی را باز می‌گردانند.

**نکته:** به ساختار هر ویژگی مدل دقت کنید؛ هر ویژگی با نوع داده، مقدار پیش‌فرض و Field دقیقاً همان ساختاری را دارد که در پارامترهای توابع عملیات مسیر دیده‌اید، فقط اینجا از Field به جای Path، Query یا Body استفاده شده است.

### ۳-۹ افزودن اطلاعات اضافی

شما می‌توانید اطلاعات اضافه‌ای را با استفاده از پارامترهای کلیدی در Field، Query، Body و غیره تعریف کنید. این اطلاعات در طرح اسکیمای JSON تولید شده لحاظ خواهد شد. در ادامه مستندات، وقتی نحوه اعلام نمونه‌ها را یاد بگیرید، اطلاعات بیشتری درباره افزودن اطلاعات اضافی خواهید آموخت.

**هشدار:** کلیدهای اضافی که به Field ارسال می‌کنید، در طرح نهایی OpenAPI اپلیکیشن شما نیز ظاهر خواهند شد. از آنجایی که این کلیدها ممکن است جزو استاندارد OpenAPI نباشند، برخی ابزارهای OpenAPI مانند OpenAPI validator ممکن است با طرح شما سازگار نباشند.

### جمع‌بندی

می‌توانید از Field در Pydantic برای افزودن اعتبارسنجی‌های اضافی و فراداده‌ها به ویژگی‌های مدل استفاده کنید. همچنین می‌توانید با استفاده از آرگومان‌های کلیدی اضافی، اطلاعات بیشتری برای اسکیمای JSON فراهم کنید.

## فصل ۱۰: بدنه – مدل‌های تو در تو

با FastAPI می‌توانید مدل‌هایی با هر سطح از تو در تو بودن را تعریف، اعتبارسنجی، مستندسازی و استفاده کنید (به لطف Pydantic).

### ۱۰-۱ فیلدهای لیست

می‌توانید یک ویژگی را به عنوان یک زیرنوع تعریف کنید. مثلاً یک لیست پایتونی:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list = []

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

این کار، tags را به یک لیست تبدیل می‌کند، اگرچه نوع عناصر لیست را مشخص نکرده‌ایم.

### ۱۰-۲ فیلدهای لیست با پارامتر نوع

اما پایتون راه خاصی برای تعریف لیست‌هایی با نوع داخلی (type parameter) دارد:

#### ۱۰-۲-۱ typing List از کردن

در پایتون ۳٫۹ و بالاتر می‌توانید از لیست استاندارد (list) برای تعیین حاشیه‌نویسی نوع استفاده کنید، همان‌طور که در ادامه خواهیم دید. اما در نسخه‌های پایتون قبل از ۳٫۹ (از جمله ۳٫۶ تا ۳٫۸)، باید ابتدا List را از ماژول typing وارد کنید:

```
from typing import List, Union
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
```



```
tags: List[str] = []

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

## ۲-۱۰ تعریف لیست با پارامتر نوع

برای تعریف نوع‌هایی که پارامتر نوع دارند (مثل list، dict، tuple):

- اگر از نسخه‌ای قبل از پایتون ۳٫۹ استفاده می‌کنید، از معادل آن در typing استفاده کنید.
- نوع داخلی را درون کروشه [ ] قرار دهید.

مثلاً در پایتون ۳٫۹ به بعد:

```
my_list: list[str]
```

در نسخه‌های پایین‌تر:

```
from typing import List

my_list: List[str]
```

این دستور زبان استاندارد پایتون برای تعریف نوع‌هاست. در نتیجه در مثال ما می‌توانیم tags را به‌طور خاص به‌عنوان «لیستی از رشته‌ها» تعریف کنیم:

```
class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list[str] = []
```

## ۳-۱۰ مجموعه‌ها (Set)

بعد به این فکر می‌کنیم که شاید تگ‌ها نباید تکراری باشند، بلکه باید رشته‌های منحصر به فرد باشند. پایتون برای مجموعه‌ای از آیتم‌های یکتا، نوع داده‌ای خاص به نام set دارد. در این صورت می‌توان tags را به‌صورت مجموعه‌ای از رشته‌ها تعریف کرد:

```
class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()
```

با این کار، حتی اگر در درخواست داده‌های تکراری ارسال شود، آن‌ها به مجموعه‌ای از آیتم‌های یکتا تبدیل می‌شوند. و وقتی آن داده‌ها را برمی‌گردانید، حتی اگر منبع تکرار داشته باشد، به صورت مجموعه‌ای از آیتم‌های یکتا نمایش داده می‌شوند. همچنین مستندسازی آن نیز به درستی انجام می‌شود.

#### ۴-۱۰ مدل‌های تو در تو

هر ویژگی یک مدل Pydantic یک نوع دارد. اما آن نوع می‌تواند خودش یک مدل Pydantic دیگر باشد. بنابراین می‌توانید ساختارهای JSON تو در تویی با نام ویژگی‌های مشخص، نوع‌ها و اعتبارسنجی‌های دقیق تعریف کنید. و این تو در تو بودن، بدون محدودیت در عمق است.

##### ۴-۱۰-۱ تعریف یک زیرمدل

مثلاً می‌توانیم یک مدل Image تعریف کنیم:

```
class Image(BaseModel):
    url: str
    name: str
```

##### ۴-۱۰-۲ استفاده از زیرمدل به عنوان نوع

سپس می‌توانیم از آن به عنوان نوع یک ویژگی استفاده کنیم:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Image(BaseModel):
    url: str
    name: str

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()
    image: Image | None = None

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

این یعنی FastAPI بدنه‌ای مشابه این انتظار دارد:

```
{
  "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"
  }
}
```

و با همین تعریف ساده، FastAPI به شما موارد زیر را ارائه می‌دهد:

- پشتیبانی کامل و هوشمند ویرایشگر (تکمیل خودکار و غیره)، حتی برای مدل‌های تو در تو
- تبدیل نوع داده‌ها
- اعتبارسنجی داده‌ها
- مستندسازی خودکار برای OpenAPI

## ۵-۱۰ انواع ویژه و اعتبارسنجی

علاوه بر انواع معمول و تکی مانند `str`، `int`، `float` و ... می‌توانید از انواع پیچیده‌تری که از `str` ارث‌بری می‌کنند نیز استفاده کنید. برای دیدن تمام گزینه‌هایی که در اختیار دارید، به بخش «نمای کلی نوع‌ها» در مستندات Pydantic مراجعه کنید. برخی نمونه‌ها را در فصل بعد خواهید دید. برای مثال، همان‌طور که در مدل `Image` یک فیلد `url` داریم، می‌توانیم آن را به جای یک `str` به صورت یک نمونه از `HttpUrl` در Pydantic تعریف کنیم:

```
from fastapi import FastAPI
from pydantic import BaseModel, HttpUrl

app = FastAPI()

class Image(BaseModel):
    url: HttpUrl
    name: str

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()
    image: Image | None = None

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

رشته بررسی می‌شود که آیا یک URL معتبر است یا نه، و در اسکیمای JSON/OpenAPI نیز به صورت یک URL مستند خواهد شد.

## ۶-۱۰ ویژگی‌هایی با لیستی از زیرمدل‌ها

می‌توانید مدل‌های Pydantic را به عنوان انواع فرعی برای list، set و ... استفاده کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel, HttpUrl

app = FastAPI()

class Image(BaseModel):
    url: HttpUrl
    name: str

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()
    images: list[Image] | None = None

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

در این حالت، یک بدنه JSON مانند زیر انتظار می‌رود (تبدیل، اعتبارسنجی، مستندسازی و ... انجام می‌شود):

```
{
  "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"
    }
  ]
}
```

**نکته:** توجه کنید که کلید images اکنون یک لیست از آبجکت‌های تصویر است.

## ۷-۱۰ مدل‌های تودرتو به صورت عمیق

می‌توانید مدل‌هایی با تودرتوی عمیق دلخواه تعریف کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel, HttpUrl

app = FastAPI()

class Image(BaseModel):
    url: HttpUrl
    name: str

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()
    images: list[Image] | None = None

class Offer(BaseModel):
    name: str
    description: str | None = None
    price: float
    items: list[Item]

@app.post("/offers/")
async def create_offer(offer: Offer):
    return offer
```

**نکته:** توجه کنید که Offer یک لیست از Itemها دارد که خودشان نیز می‌توانند یک لیست اختیاری از Imageها داشته باشند.

## ۸-۱۰ بدنه‌هایی که صرفاً یک لیست هستند

اگر مقدار سطح بالای بدنه JSON شما یک آرایه JSON (یک لیست پایتون) باشد، می‌توانید نوع آن را مانند مدل‌های Pydantic تعریف کنید:

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

یا در پایتون نسخه ۳٫۹ و بالاتر:

```
images: list[Image]
```

برای مثال:

```
from fastapi import FastAPI
from pydantic import BaseModel, HttpUrl
```

```

app = FastAPI()

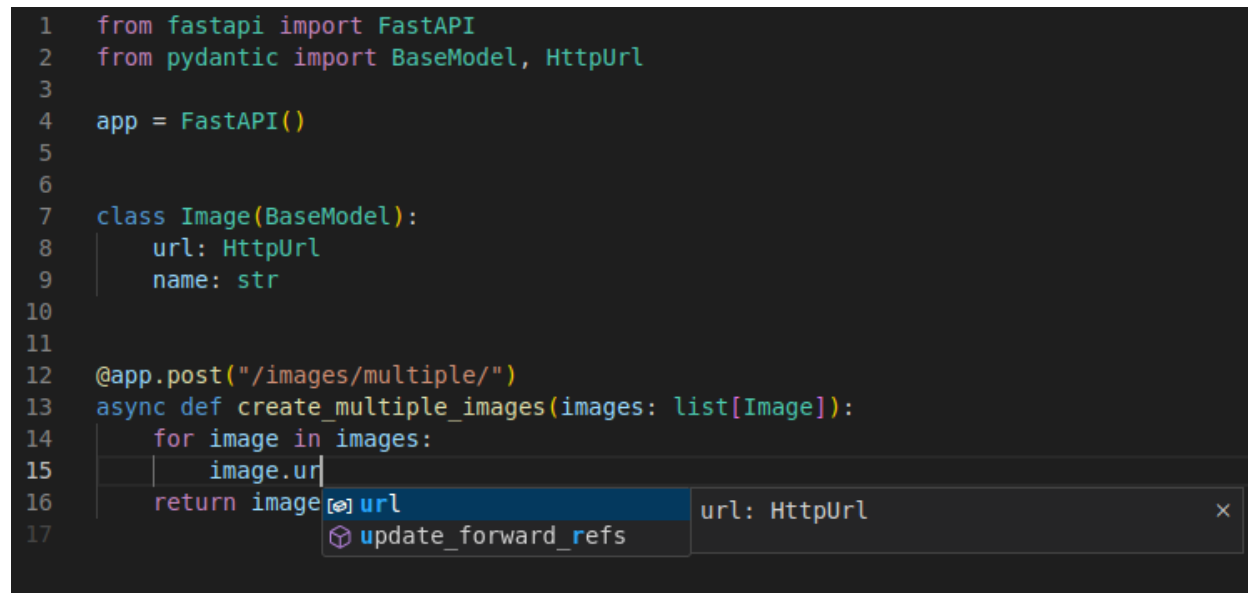
class Image(BaseModel):
    url: HttpUrl
    name: str

@app.post("/images/multiple/")
async def create_multiple_images(images: list[Image]):
    return images

```

### ۱۰-۱۰ پشتیبانی کامل از ویرایشگر کد

در این حالت، حتی برای آیتم‌های داخل لیست نیز پشتیبانی کامل از تکمیل خودکار و بررسی نوع خواهید داشت.



```

1  from fastapi import FastAPI
2  from pydantic import BaseModel, HttpUrl
3
4  app = FastAPI()
5
6
7  class Image(BaseModel):
8      url: HttpUrl
9      name: str
10
11
12 @app.post("/images/multiple/")
13 async def create_multiple_images(images: list[Image]):
14     for image in images:
15         image.url
16     return image
17

```

The screenshot shows an IDE completion popup for the `image.url` attribute. The popup lists `url: HttpUrl` and `update_forward_refs` as options.

اگر به جای مدل‌های Pydantic مستقیماً با دیکشنری کار می‌کردید، چنین پشتیبانی‌ای از ویرایشگر دریافت نمی‌کردید. همچنین لازم نیست نگران تبدیل داده باشید؛ دیکشنری‌های ورودی به صورت خودکار تبدیل می‌شوند و خروجی نیز به صورت خودکار به JSON برمی‌گردد.

### ۱۰-۱۱ بدنه‌هایی که دیکشنری دلخواه هستند

می‌توانید یک بدنه را به صورت دیکشنری تعریف کنید با کلیدهایی از یک نوع و مقادیری از نوع دیگر. در این حالت، نیازی نیست از قبل بدانید که نام کلیدهای معتبر چه چیزهایی هستند (برخلاف مدل‌های Pydantic). این روش زمانی مفید است که می‌خواهید کلیدهایی را دریافت کنید که از قبل آن‌ها را نمی‌شناسید. یا زمانی که می‌خواهید کلیدها نوعی غیر از `str` داشته باشند (مثلاً `int`). در این مثال، می‌خواهیم یک دیکشنری را بپذیریم که کلیدهای آن `int` و مقادیر آن `float` باشند:

```

from fastapi import FastAPI

app = FastAPI()

```

```
@app.post("/index-weights/")
async def create_index_weights(weights: dict[int, float]):
    return weights
```

**نکته:** به یاد داشته باشید که JSON فقط `str` را به عنوان کلید پشتیبانی می کند. اما Pydantic تبدیل خودکار نوع داده را انجام می دهد. یعنی حتی اگر کلاینت های شما فقط بتوانند کلیدهایی به شکل رشته بفرستند، مادامی که آن رشته ها اعداد صحیح (`int`) باشند، Pydantic آن ها را تبدیل و اعتبارسنجی می کند. و دیکشنری ای که شما در پارامتر `weights` دریافت می کنید، در واقع کلیدهای `int` و مقادیر `float` خواهد داشت.

### جمع بندی

با FastAPI بیشترین انعطاف پذیری ممکن از مدل های Pydantic را دارید، در حالی که کد شما ساده، کوتاه و زیبا باقی می ماند. و در کنار آن تمام مزایا را دارید:

- پشتیبانی کامل از ویرایشگر (تکمیل خودکار در همه جا!)
- تبدیل داده (یا همان `parse / serialization`)
- اعتبارسنجی داده
- مستندسازی اسکیمای
- مستندات خودکار

## فصل ۱۱: اعلان نمونه داده‌های درخواست

شما می‌توانید نمونه‌هایی از داده‌هایی را که اپلیکیشن‌تان می‌پذیرد، تعریف کنید. در اینجا چند روش مختلف برای انجام این کار وجود دارد.

### ۱۱-۱ داده‌های اضافی اسکیمای JSON در مدل‌های Pydantic

می‌توانید برای یک مدل Pydantic نمونه‌هایی تعریف کنید که به اسکیمای JSON تولیدشده افزوده شوند:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

model_config = {
    "json_schema_extra": {
        "examples": [
            {
                "name": "Foo",
                "description": "A very nice Item",
                "price": 35.4,
                "tax": 3.2,
            }
        ]
    }
}

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

این اطلاعات اضافه به صورت مستقیم به اسکیمای JSON افزوده می‌شوند و در مستندات API نمایش داده خواهند شد. در Pydantic نسخه ۲ از ویژگی `model_config` استفاده می‌شود که دیکشنری‌ای برای پیکربندی مدل است. شما می‌توانید کلید `"json_schema_extra"` را با دیکشنری‌ای حاوی داده‌های اضافی موردنظرتان از جمله `examples` تنظیم کنید.

**نکته:** می‌توانید با همین تکنیک اطلاعات دلخواه دیگری نیز به اسکیمای JSON اضافه کنید، مثلاً برای رابط کاربری فرانت‌اند.



**اطلاعات:** OpenAPI نسخه ۳٫۱٫۰ (از FastAPI 0.99.0 به بعد) از کلید examples پشتیبانی می‌کند که جزئی از استاندارد اسکیمای JSON است. قبل از آن تنها کلید example پشتیبانی می‌شد که تنها یک مثال را قبول می‌کرد و اکنون منسوخ شده است. بنابراین بهتر است به examples مهاجرت کنید.

## ۲-۱۱ آرگومان‌های اضافی در Field

زمانی که از Field() در مدل‌های Pydantic استفاده می‌کنید، می‌توانید نمونه‌هایی نیز تعریف کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):
    name: str = Field(examples=["Foo"])
    description: str | None = Field(default=None, examples=["A very nice Item"])
    price: float = Field(examples=[35.4])
    tax: float | None = Field(default=None, examples=[3.2])

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

## ۳-۱۱ نمونه‌ها در OpenAPI – اسکیمای JSON

هنگامی که از هر یک از موارد زیر استفاده می‌کنید:

- Path()
- Query()
- Header()
- Cookie()
- Body()
- Form()
- File()

می‌توانید مجموعه‌ای از نمونه‌ها را همراه با اطلاعات اضافی اعلام کنید که به اسکیمای JSON آن‌ها در داخل OpenAPI افزوده می‌شود.

### ۱-۳-۱۱ بدنه با یک نمونه

در اینجا، نمونه‌ای را ارسال می‌کنیم که شامل یک نمونه از داده‌ای است که در Body() انتظار می‌رود.

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel
```

```

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

@app.put("/items/{item_id}")
async def update_item(
    item_id: int,
    item: Annotated[
        Item,
        Body(
            examples=[
                {
                    "name": "Foo",
                    "description": "A very nice Item",
                    "price": 35.4,
                    "tax": 3.2,
                }
            ],
        ),
    ],
):
    results = {"item_id": item_id, "item": item}
    return results

```

## ۲-۳-۱۱ نمونه در رابط کاربری مستندات

هنگام استفاده از هر یک از روش‌های بالا، این گونه در مسیر `/docs` نمایش داده خواهد شد.

Fast API 0.1.0 OAS3

/openapi.json

default

**PUT** /items/{item\_id} Update Item Put

**Parameters** Try it out

Name	Description
<b>item_id</b> * required integer (path)	

**Request body** required application/json

**Example Value** | Schema

```
{
  "name": "Foo",
  "description": "A very nice Item",
  "price": 35.4,
  "tax": 3.2
}
```

**Responses**

Code	Description	Links
200	<b>Successful Response</b>	No links
422	<b>Validation Error</b>	No links

۳-۳-۱۱ بدنه با چند نمونه

شما البته می‌توانید چندین نمونه را نیز ارسال کنید.

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel
```

```

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

@app.put("/items/{item_id}")
async def update_item(
    *,
    item_id: int,
    item: Annotated[
        Item,
        Body(
            examples=[
                {
                    "name": "Foo",
                    "description": "A very nice Item",
                    "price": 35.4,
                    "tax": 3.2,
                },
                {
                    "name": "Bar",
                    "price": "35.4",
                },
                {
                    "name": "Baz",
                    "price": "thirty five point four",
                },
            ],
        ),
    ],
):
    results = {"item_id": item_id, "item": item}
    return results

```

وقتی این کار را انجام می‌دهید، نمونه‌ها بخشی از اسکیمای داخلی JSON برای آن داده‌ی بدنه خواهند بود. با این حال، در زمان نگارش این متن، Swagger UI هنوز از نمایش چندین نمونه برای داده‌های موجود در اسکیمای JSON پشتیبانی نمی‌کند. اما در ادامه یک راه‌حل برای این موضوع آورده شده است.

### ۴-۳-۱۱ نمونه‌های مخصوص OpenAPI

از زمانی پیش از آن که اسکیمای JSON از examples پشتیبانی کند، OpenAPI پشتیبانی از فیلدی متفاوت با همین نام یعنی examples را فراهم کرده بود. این فیلد examples مخصوص OpenAPI در بخش متفاوتی از مشخصات OpenAPI قرار می‌گیرد. این فیلد در جزئیات مربوط به هر عملیات مسیر (path operation) تعریف می‌شود، نه درون هر اسکیمای JSON.

Swagger UI نیز مدتی است که از این فیلد خاص examples پشتیبانی می‌کند. بنابراین می‌توانید از آن برای نمایش نمونه‌های مختلف در رابط مستندات استفاده کنید. ساختار این فیلد خاص OpenAPI، یک دیکشنری شامل چندین نمونه است (به جای یک

لیست). هر نمونه دارای اطلاعات اضافی ای است که در مستندات OpenAPI نیز نمایش داده خواهد شد. این فیلد داخل هیچ یک از اسکیماهای JSON موجود در OpenAPI قرار نمی گیرد، بلکه مستقیماً در تعریف عملیات مسیر (path operation) قرار دارد.

### ۵-۳-۱۱ استفاده از پارامتر openapi\_examples

می توانید نمونه های خاص OpenAPI را در FastAPI با استفاده از پارامتر openapi\_examples تعریف کنید برای موارد زیر:

- Path()
- Query()
- Header()
- Cookie()
- Body()
- Form()
- File()

کلیدهای دیکشنری openapi\_examples معرف هر نمونه هستند و مقدار هر کلید، خود یک دیکشنری دیگر است. هر دیکشنری مربوط به یک نمونه می تواند شامل موارد زیر باشد:

- summary: توضیح کوتاهی برای نمونه.
- description: توضیحی طولانی تر که می تواند شامل متن Markdown باشد.
- value: نمونه واقعی داده که نشان داده خواهد شد، مثلاً یک دیکشنری.
- externalValue: جایگزینی برای value، آدرسی (URL) که به نمونه اشاره دارد. گرچه ممکن است توسط همه ابزارها مانند value پشتیبانی نشود.

می توانید از آن به این شکل استفاده کنید:

```
from typing import Annotated
from fastapi import Body, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None

@app.put("/items/{item_id}")
async def update_item(
    *,
    item_id: int,
    item: Annotated[
        Item,
        Body(
            openapi_examples={
```

```

        "normal": {
            "summary": "A normal example",
            "description": "A **normal** item works correctly.",
            "value": {
                "name": "Foo",
                "description": "A very nice Item",
                "price": 35.4,
                "tax": 3.2,
            },
        },
        "converted": {
            "summary": "An example with converted data",
            "description": "FastAPI can convert price `strings` to actual `numbers` automatically",
            "value": {
                "name": "Bar",
                "price": "35.4",
            },
        },
        "invalid": {
            "summary": "Invalid data is rejected with an error",
            "value": {
                "name": "Baz",
                "price": "thirty five point four",
            },
        },
    ],
),
):
    results = {"item_id": item_id, "item": item}
    return results

```

### ۶-۳-۱۱ نمونه‌هایی از OpenAPI در رابط کاربری مستندات

با افزودن `openapi_examples` به `Body()`، رابط `/docs` به این صورت نمایش داده می‌شود:

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/update\_item\_items\_\_item\_id\_\_put

Incognito

FastAPI0.1.0OAS3

/openapi.json

default

PUT

/items/{item\_id}

Update Item

Parameters

Try it out

Name	Description
<b>item_id</b> <small>★ required</small>	
integer	item_id
(path)	

Request body required

application/json

Examples:

A normal example

A normal example

An example with converted data

Invalid data is rejected with an error

```
{  "name": "Foo",  "description": "A very nice Item",  "price": 35.4,  "tax": 3.2}
```

Example Description

A normal item works correctly.

Responses

Code	Description	Links
200	Successful Response	No links

## فصل ۱۲: انواع داده‌ای اضافی

تا اینجا، شما از انواع داده‌ای رایج مانند موارد زیر استفاده کرده‌اید:

- `int`
- `float`
- `str`
- `bool`

اما می‌توانید از انواع داده‌ای پیچیده‌تری نیز استفاده کنید و همچنان از همان قابلیت‌هایی که تاکنون دیده‌اید بهره‌مند خواهید شد:

- پشتیبانی عالی و هوشمندانه ویرایشگر.
- تبدیل داده‌ها از درخواست‌های ورودی.
- تبدیل داده‌ها برای پاسخ‌ها.
- اعتبارسنجی داده‌ها.
- مستندسازی و تعریف خودکار نوع‌ها.

### ۱۲-۱ سایر انواع داده‌ای

در ادامه برخی از انواع داده‌ای اضافی که می‌توانید استفاده کنید آمده است:

- **:UUID**  
یک شناسه استاندارد جهانی (Universally Unique Identifier) که در بسیاری از پایگاه‌های داده و سامانه‌ها به عنوان شناسه استفاده می‌شود. در درخواست‌ها و پاسخ‌ها به صورت رشته (`str`) نمایش داده می‌شود.
- **:datetime.datetime**  
یک شیء `datetime.datetime` پایتون.  
در درخواست‌ها و پاسخ‌ها به صورت رشته‌ای با فرمت ISO 8601 نمایش داده می‌شود، مانند:  
`2008-09-15T15:53:00+05:00`
- **:datetime.date**  
شیء `datetime.date` پایتون.  
در درخواست‌ها و پاسخ‌ها به صورت رشته با فرمت ISO 8601، مانند: `2008-09-15`
- **:datetime.time**  
شیء `datetime.time` پایتون.  
در درخواست‌ها و پاسخ‌ها به صورت رشته با فرمت ISO 8601، مانند: `14:23:55.003`
- **:datetime.timedelta**  
شیء `datetime.timedelta` پایتون.  
در درخواست‌ها و پاسخ‌ها به صورت عدد اعشاری نمایش داده می‌شود که معادل تعداد ثانیه‌های کل است.



Pydantic همچنین اجازه می‌دهد این مقدار به صورت «رمزگذاری فاصله زمانی ISO 8601» نمایش داده شود. برای اطلاعات بیشتر به مستندات مراجعه کنید.

• **frozenset:**

در درخواست‌ها و پاسخ‌ها همانند مجموعه (set) رفتار می‌کند:

- در درخواست‌ها، یک لیست خوانده می‌شود، مقادیر تکراری حذف شده و به set تبدیل می‌گردد.
- در پاسخ‌ها، set به لیست تبدیل می‌شود.
- در اسکیمای تولید شده ذکر می‌شود که مقادیر مجموعه باید یکتا باشند (با استفاده از ویژگی uniqueItems در اسکیمای JSON).

• **bytes:**

نوع استاندارد bytes پایتون.

در درخواست‌ها و پاسخ‌ها به صورت str پردازش می‌شود.

اسکیمای تولید شده مشخص خواهد کرد که این مقدار، رشته‌ای با فرمت باینری است.

• **Decimal:**

نوع استاندارد Decimal در پایتون.

در درخواست‌ها و پاسخ‌ها مانند float پردازش می‌شود.

برای مشاهده تمام انواع داده‌ای معتبر در Pydantic به مستندات مراجعه کنید.

## ۲-۱۲ مثال

در اینجا مثالی از یک مسیر عملیات (path operation) آورده شده که از برخی از انواع بالا استفاده می‌کند:

```
from datetime import datetime, time, timedelta
from typing import Annotated
from uuid import UUID
from fastapi import Body, FastAPI

app = FastAPI()

@app.put("/items/{item_id}")
async def read_items(
    item_id: UUID,
    start_datetime: Annotated[datetime, Body()],
    end_datetime: Annotated[datetime, Body()],
    process_after: Annotated[timedelta, Body()],
    repeat_at: Annotated[time | None, Body()] = None,
):
    start_process = start_datetime + process_after
    duration = end_datetime - start_process
    return {
        "item_id": item_id,
        "start_datetime": start_datetime,
        "end_datetime": end_datetime,
        "process_after": process_after,
```

```

        "repeat_at": repeat_at,
        "start_process": start_process,
        "duration": duration,
    }

```

توجه داشته باشید که پارامترهای داخل تابع از نوع‌های طبیعی (native) خود استفاده می‌کنند و شما می‌توانید به راحتی عملیات‌های معمول تاریخ را انجام دهید، مثلاً:

```

from datetime import datetime, time, timedelta
from typing import Annotated
from uuid import UUID
from fastapi import Body, FastAPI

app = FastAPI()

@app.put("/items/{item_id}")
async def read_items(
    item_id: UUID,
    start_datetime: Annotated[datetime, Body()],
    end_datetime: Annotated[datetime, Body()],
    process_after: Annotated[timedelta, Body()],
    repeat_at: Annotated[time | None, Body()] = None,
):
    start_process = start_datetime + process_after
    duration = end_datetime - start_process
    return {
        "item_id": item_id,
        "start_datetime": start_datetime,
        "end_datetime": end_datetime,
        "process_after": process_after,
        "repeat_at": repeat_at,
        "start_process": start_process,
        "duration": duration,
    }

```

با استفاده از این انواع داده‌ای اضافی، برنامه‌های شما می‌توانند داده‌های غنی‌تری را پردازش و مستند کنند، و همچنان از مزایای کامل FastAPI و اعتبارسنجی داده‌ها بهره‌مند شوند.

## فصل ۱۳: پارامترهای کوکی

کوکی (Cookie) یک قطعه داده است که از سمت سرور در مرورگر کاربر ذخیره می‌شود و با هر درخواست بعدی به همان سرور، به صورت خودکار در سرصفحه‌ی درخواست (HTTP headers) ارسال می‌شود. کوکی‌ها معمولاً برای شناسایی کاربران، نگه‌داری وضعیت ورود، یا ذخیره اطلاعات شخصی سازی شده استفاده می‌شوند.

در FastAPI، پارامترهای کوکی امکان دسترسی به مقادیر ذخیره شده در کوکی‌های ارسال شده از سمت کاربر را فراهم می‌کنند. این پارامترها مشابه پارامترهای Query و Path تعریف می‌شوند، اما به جای گرفتن داده از آدرس URL، مقدارشان را از کوکی‌های مرورگر دریافت می‌کنند.

### ۱۳-۱ وارد کردن Cookie

ابتدا Cookie را وارد کنید:

```
from typing import Annotated
from fastapi import Cookie, FastAPI

app = FastAPI()

@app.get("/items/")
async def read_items(ads_id: Annotated[str | None, Cookie()] = None):
    return {"ads_id": ads_id}
```

### ۱۳-۲ تعریف پارامترهای Cookie

سپس پارامترهای کوکی را با استفاده از همان ساختاری که برای Path و Query استفاده می‌کنید تعریف کنید. می‌توانید مقدار پیش فرض را مشخص کرده و همچنین تمام پارامترهای اضافی مربوط به اعتبارسنجی یا توصیف (annotation) را نیز تعریف کنید:

```
from typing import Annotated
from fastapi import Cookie, FastAPI

app = FastAPI()

@app.get("/items/")
async def read_items(ads_id: Annotated[str | None, Cookie()] = None):
    return {"ads_id": ads_id}
```

**جزئیات فنی:** Cookie یک کلاس «خواهر» برای Path و Query است. این کلاس نیز از همان کلاس پایه‌ی مشترک یعنی Param ارث‌بری می‌کند. اما به یاد داشته باشید که وقتی Query، Path، Cookie و سایر موارد را از fastapi وارد می‌کنید، در واقع این‌ها توابعی هستند که نمونه‌هایی از کلاس‌هایی با همان نام را برمی‌گردانند.

**اطلاعات:** برای تعریف کوکی‌ها باید از Cookie استفاده کنید؛ در غیر این صورت، پارامترها به عنوان پارامترهای پرس‌وجو تفسیر خواهند شد.

### جمع‌بندی

کوکی‌ها را با استفاده از Cookie تعریف کنید، با همان الگوی عمومی که برای Query و Path به کار می‌برید.

## فصل ۱۴: پارامترهای هدر

پارامترهای Header در FastAPI برای دریافت اطلاعات موجود در سرصفحه‌ی درخواست HTTP طراحی شده‌اند و می‌توانند برای اعتبارسنجی، کنترل دسترسی، تنظیمات کلاینت و بسیاری کاربردهای دیگر به کار روند. می‌توانید پارامترهای Header را به همان شکلی تعریف کنید که پارامترهای Query، Path و Cookie را تعریف می‌کنید.

### ۱-۱۴ وارد کردن Header

ابتدا Header را وارد کنید:

```
from typing import Annotated
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")
async def read_items(user_agent: Annotated[str | None, Header()] = None):
    return {"User-Agent": user_agent}
```

### ۲-۱۴ تعریف پارامترهای Header

سپس پارامترهای header را با همان ساختاری که برای Path، Query و Cookie استفاده می‌شود، تعریف کنید. می‌توانید مقدار پیش‌فرض و همین‌طور تمام پارامترهای اضافی اعتبارسنجی یا توضیحی (annotation) را نیز تعیین کنید:

```
from typing import Annotated
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")
async def read_items(user_agent: Annotated[str | None, Header()] = None):
    return {"User-Agent": user_agent}
```

**جزئیات فنی:** Header یک کلاس «خواهر» برای Path، Query و Cookie است. این کلاس نیز از کلاس مشترک Param ارث‌بری می‌کند. اما به یاد داشته باشید که وقتی Query، Path، Header و دیگر موارد را از fastapi وارد می‌کنید، در واقع توابعی را وارد می‌کنید که نمونه‌هایی از کلاس‌هایی با همان نام را برمی‌گردانند.

**اطلاعات:** برای تعریف هدرها، باید از Header استفاده کنید، زیرا در غیر این صورت، پارامترها به‌عنوان پارامترهای پرس‌وجو تفسیر خواهند شد.

### ۳-۱۴ تبدیل خودکار

Header قابلیت‌های اضافی کوچکی فراتر از آنچه Path، Query و Cookie ارائه می‌دهند دارد. اکثر هدرهای استاندارد HTTP با کاراکتر «خط تیره» یا «منفی» (-) از هم جدا می‌شوند. اما متغیری مانند user-agent در پایتون نامعتبر است. بنابراین، به صورت پیش‌فرض، Header کاراکترهای زیرخط ( \_ ) را به خط تیره (-) تبدیل می‌کند تا بتواند هدرها را استخراج کرده و در مستندات نمایش دهد.

همچنین، هدرهای HTTP نسبت به حروف بزرگ و کوچک حساس نیستند، بنابراین می‌توانید آن‌ها را با سبک استاندارد پایتون (معروف به snake\_case) تعریف کنید. بنابراین، می‌توانید از user\_agent همان‌گونه که به‌طور معمول در کد پایتون استفاده می‌کنید بهره ببرید، بدون اینکه مجبور باشید آن را به شکلی مانند User\_Agent یا چیزی مشابه بنویسید. اگر به هر دلیلی نیاز دارید تبدیل خودکار زیرخط به خط تیره را غیرفعال کنید، پارامتر convert\_underscores را در Header برابر با False قرار دهید:

```
from typing import Annotated
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")
async def read_items(
    strange_header: Annotated[str | None, Header(convert_underscores=False)]
    = None,
):
    return {"strange_header": strange_header}
```

**هشدار:** قبل از تنظیم convert\_underscores=False، به این نکته توجه داشته باشید که برخی پراکسی‌ها و سرورهای HTTP اجازه استفاده از هدرهایی که دارای زیرخط ( \_ ) هستند را نمی‌دهند.

## ۴-۱۴ هدرهای تکراری

امکان دریافت هدرهای تکراری نیز وجود دارد. به این معنی که یک هدر خاص می‌تواند چندین مقدار داشته باشد. برای تعریف چنین حالتی، می‌توانید در تعریف نوع، از لیست استفاده کنید. تمام مقادیر آن هدر تکراری را در قالب یک لیست پایتونی دریافت خواهید کرد. برای مثال، برای تعریف هدری به نام X-Token که ممکن است بیش از یک بار تکرار شود، می‌توانید چنین بنویسید:

```
from typing import Annotated
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")
async def read_items(x_token: Annotated[list[str] | None, Header()] = None):
    return {"X-Token values": x_token}
```

اگر با این مسیر با ارسال دو هدر مانند زیر ارتباط برقرار کنید:

```
X-Token: foo
```

```
X-Token: bar
```

پاسخ به صورت زیر خواهد بود:

```
{
  "X-Token values": [
    "bar",
    "foo"
  ]
}
```

### جمع‌بندی

هدرها را با استفاده از Header تعریف کنید، با همان الگوی عمومی که برای Path, Query و Cookie استفاده می‌شود، و نگران زیرخط‌های متغیرهایتان نباشید، FastAPI به‌طور خودکار آن‌ها را به خط تیره تبدیل خواهد کرد.

## فصل ۱۵: مدل‌های پارامتر کوکی

اگر گروهی از کوکی‌ها دارید که به هم مرتبط هستند، می‌توانید یک مدل Pydantic برای تعریف آن‌ها ایجاد کنید. این کار به شما اجازه می‌دهد که مدل را در چندین محل مختلف استفاده کرده و همچنین اعتبارسنجی‌ها و فراداده را برای همه پارامترها به صورت یکجا تعریف کنید.

**نکته:** این ویژگی از نسخه‌ی FastAPI 0.115.0 پشتیبانی می‌شود.

**راهنما:** همین تکنیک برای Query، Cookie و Header نیز قابل استفاده است.

### ۱۵-۱ کوکی‌ها با مدل Pydantic

پارامترهای کوکی موردنیازتان را در یک مدل Pydantic تعریف کرده و سپس پارامتر را به صورت Cookie اعلان کنید:

```
from typing import Annotated

from fastapi import Cookie, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Cookies(BaseModel):
    session_id: str
    facebook_tracker: str | None = None
    googall_tracker: str | None = None

@app.get("/items/")
async def read_items(cookies: Annotated[Cookies, Cookie()]):
    return cookies
```

FastAPI داده‌های مربوط به هر فیلد را از کوکی‌های دریافت‌شده در درخواست استخراج می‌کند و مدلی که تعریف کرده‌اید را در اختیارتان قرار می‌دهد.

### ۱۵-۲ مستندات را بررسی کنید

می‌توانید کوکی‌های تعریف‌شده را در رابط مستندات موجود در مسیر `/docs` مشاهده کنید:



FastAPI 0.1.0 OAS 3.1  
/openapi.json

default

GET /items/ Read Items

Parameters

Try it out

Name	Description
session_id * required string (cookie)	session_id
fatebook_tracker string (cookie)	fatebook_tracker
googall_tracker string (cookie)	googall_tracker

Responses

**اطلاعات:** در نظر داشته باشید که مرورگرها کوکی‌ها را به شکل خاصی و در پشت‌صحنه مدیریت می‌کنند و به سادگی اجازه دسترسی جاوااسکریپت به آن‌ها را نمی‌دهند. اگر به رابط مستندات API در آدرس /docs بروید، می‌توانید مستندات مربوط به کوکی‌ها برای عملیات‌های مسیر خود را مشاهده کنید. اما حتی اگر داده‌ها را پر کرده و روی "Execute" کلیک کنید، چون رابط مستندات با جاوااسکریپت کار می‌کند، کوکی‌ها ارسال نخواهند شد، و با پیامی خطا مواجه خواهید شد که گویی هیچ مقداری وارد نکرده‌اید.

### ۳-۱۵ جلوگیری از کوکی‌های اضافی

در برخی موارد خاص (که احتمالاً رایج نیستند)، ممکن است بخواهید دریافت کوکی‌ها را به کوکی‌های خاصی محدود کنید. اکنون API شما می‌تواند بر رضایت مربوط به کوکی‌های خود کنترل داشته باشد. می‌توانید از تنظیمات مدل Pydantic استفاده کنید تا از پذیرش فیلدهای اضافی جلوگیری شود:

```
from typing import Annotated, Union
from fastapi import Cookie, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Cookies(BaseModel):
    model_config = {"extra": "forbid"}

    session_id: str
    fatebook_tracker: Union[str, None] = None
    googall_tracker: Union[str, None] = None

@app.get("/items/")
async def read_items(cookies: Annotated[Cookies, Cookie()]):
```

```
return cookies
```

اگر یک کلاینت سعی کند کوکی‌های اضافی ارسال کند، یک پاسخ خطا دریافت خواهد کرد. بنرهای کوکی ضعیف، با تمام تلاششان برای گرفتن رضایت، با پاسخ رد روبه‌رو خواهند شد! برای مثال، اگر کلاینت تلاش کند کوکی‌ای به نام `santa_tracker` با مقدار `good-list-please` ارسال کند، پاسخ خطایی مشابه زیر دریافت خواهد کرد که به او می‌گوید این کوکی مجاز نیست:

```
{
  "detail": [
    {
      "type": "extra_forbidden",
      "loc": ["cookie", "santa_tracker"],
      "msg": "Extra inputs are not permitted",
      "input": "good-list-please"
    }
  ]
}
```

### خلاصه

می‌توانید از مدل‌های `Pydantic` برای تعریف کوکی‌ها در `FastAPI` استفاده کنید.

## فصل ۱۶: مدل‌های پارامتر هدر

اگر گروهی از پارامترهای مرتبط به هدر دارید، می‌توانید با ایجاد یک مدل Pydantic آن‌ها را تعریف کنید. این کار به شما اجازه می‌دهد تا مدل را در چند جای مختلف استفاده کنید و همچنین اعتبارسنجی‌ها و فراداده‌ها را برای تمام پارامترها به صورت یک جا تعریف نمایید.

**نکته:** این ویژگی از نسخه‌ی FastAPI 0.115.0 پشتیبانی می‌شود.

### ۱۶-۱ پارامترهای Header با یک مدل Pydantic

پارامترهای مورد نیاز خود را در یک مدل Pydantic تعریف کرده، و سپس این پارامتر را با Header مشخص کنید:

```
from typing import Annotated
from fastapi import FastAPI, Header
from pydantic import BaseModel

app = FastAPI()

class CommonHeaders(BaseModel):
    host: str
    save_data: bool
    if_modified_since: str | None = None
    traceparent: str | None = None
    x_tag: list[str] = []

@app.get("/items/")
async def read_items(headers: Annotated[CommonHeaders, Header()]):
    return headers
```

FastAPI داده‌های هر فیلد را از هدرهای درخواست استخراج کرده و مدلی از نوع Pydantic که تعریف کرده‌اید به شما تحویل می‌دهد.

### ۱۶-۲ بررسی در مستندات

می‌توانید هدرهای مورد نیاز را در رابط کاربری مستندات در مسیر `/docs` مشاهده کنید:

**FastAPI** 0.1.0 OAS 3.1  
/openapi.json

**default**

**GET** /items/ Read Items

**Parameters** Cancel

Name	Description
<b>host</b> * required string (header)	host
<b>save_data</b> * required boolean (header)	--
<b>if_modified_since</b> string (header)	if_modified_since
<b>traceparent</b> string (header)	traceparent
<b>x_tag</b> array[string] (header)	<span>Add string item</span>

**Servers**

These operation-level options override the global server options.

/

**Execute**

**Responses**

### ۱۶-۳ جلوگیری از هدرهای اضافه

در برخی موارد خاص (که احتمالاً خیلی رایج نیستند)، ممکن است بخواهید تنها مجموعه خاصی از هدرها را بپذیرید. برای این کار می‌توانید از پیکربندی مدل Pydantic استفاده کرده و هر فیلد اضافه‌ای را ممنوع کنید:

```
from typing import Annotated
from fastapi import FastAPI, Header
```

```

from pydantic import BaseModel

app = FastAPI()

class CommonHeaders(BaseModel):
    model_config = {"extra": "forbid"}

    host: str
    save_data: bool
    if_modified_since: str | None = None
    traceparent: str | None = None
    x_tag: list[str] = []

@app.get("/items/")
async def read_items(headers: Annotated[CommonHeaders, Header()]):
    return headers

```

اگر کلاینت تلاش کند هدر اضافه‌ای ارسال کند، یک پاسخ خطا دریافت خواهد کرد. برای مثال، اگر کلاینت سعی کند هدری به نام tool با مقدار plumbus ارسال کند، پاسخ خطایی دریافت می‌کند که می‌گوید پارامتر هدر tool مجاز نیست:

```

{
  "detail": [
    {
      "type": "extra_forbidden",
      "loc": ["header", "tool"],
      "msg": "Extra inputs are not permitted",
      "input": "plumbus"
    }
  ]
}

```

#### ۴-۱۶ غیرفعال کردن تبدیل زیرخط

درست مانند پارامترهای عادی هدر، زمانی که نام پارامتر دارای کاراکتر زیرخط ( \_ ) باشد، به‌طور خودکار به خط تیره (-) تبدیل می‌شود. برای مثال، اگر پارامتر هدر در کد شما save\_data باشد، هدر مورد انتظار در HTTP به صورت save-data خواهد بود و همین‌طور در مستندات نمایش داده می‌شود. اگر به هر دلیلی نیاز دارید که این تبدیل خودکار را غیرفعال کنید، می‌توانید این کار را برای مدل‌های Pydantic که برای هدرها استفاده می‌شوند نیز انجام دهید:

```

from typing import Annotated
from fastapi import FastAPI, Header
from pydantic import BaseModel

app = FastAPI()

class CommonHeaders(BaseModel):
    host: str
    save_data: bool
    if_modified_since: str | None = None
    traceparent: str | None = None
    x_tag: list[str] = []

```

```
@app.get("/items/")
async def read_items(
    headers: Annotated[CommonHeaders, Header(convert_underscores=False)],
):
    return headers
```

**هشدار:** پیش از تنظیم `convert_underscores=False`، در نظر داشته باشید که برخی از پراکسی‌ها و سرورهای HTTP استفاده از هدرهایی که دارای زیرخط هستند را مجاز نمی‌دانند.

### خلاصه

می‌توانید از مدل‌های Pydantic برای تعریف Headerها در FastAPI استفاده کنید.

## فصل ۱۷: مدل پاسخ – نوع بازگشتی

می‌توانید نوع داده‌ای را که برای پاسخ استفاده می‌شود، با استفاده از نوع بازگشتی تابع مسیر (path operation function) تعیین کنید. می‌توانید از اعلان نوع به همان شکلی استفاده کنید که برای داده‌های ورودی در پارامترهای تابع استفاده می‌کنید؛ می‌توانید از مدل‌های Pydantic، لیست‌ها، دیکشنری‌ها، مقادیر اسکالر مانند اعداد صحیح، بولی و غیره استفاده کنید.

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list[str] = []

@app.post("/items/")
async def create_item(item: Item) -> Item:
    return item

@app.get("/items/")
async def read_items() -> list[Item]:
    return [
        Item(name="Portal Gun", price=42.0),
        Item(name="Plumbus", price=32.0),
    ]
```

FastAPI از این نوع بازگشتی برای موارد زیر استفاده می‌کند:

- اعتبارسنجی داده‌های بازگشتی. اگر داده‌ها نامعتبر باشند (مثلاً یک فیلد جا افتاده باشد)، یعنی کد اپلیکیشن شما خراب است و داده‌ای که باید بازگرداند را بازنگردانده، و در نتیجه به جای بازگرداندن داده نادرست، خطای سمت سرور برمی‌گرداند. این کار باعث می‌شود شما و کلاینت‌هایتان مطمئن باشید که داده‌ها را با ساختار مورد انتظار دریافت می‌کنید.
- اضافه کردن یک اسکیمای JSON برای پاسخ در عملیات مسیر OpenAPI. این اسکیمای در مستندات خودکار استفاده می‌شود.
- همچنین در ابزارهای تولید خودکار کد کلاینت نیز استفاده می‌شود.

اما مهم‌تر از همه:

- داده‌های خروجی را به آنچه در نوع بازگشتی تعریف شده محدود و فیلتر می‌کند.  
این موضوع به‌ویژه برای امنیت اهمیت دارد که در ادامه بیشتر به آن می‌پردازیم.

### ۱۷-۱ پارامتر `response_model`

برخی موارد وجود دارد که نیاز دارید یا می‌خواهید داده‌هایی را بازگردانید که دقیقاً مطابق نوع تعریف‌شده نباشند. برای مثال، ممکن است بخواهید یک دیکشنری یا شیء دیتابیس را بازگردانید، اما آن را به‌عنوان یک مدل Pydantic تعریف کنید. در این حالت، مدل Pydantic تمام مستندسازی، اعتبارسنجی و غیره را برای شیء بازگشتی (مثلاً یک دیکشنری یا شیء دیتابیس) انجام می‌دهد. اگر از حاشیه‌نویسی نوع بازگشتی استفاده کرده باشید، ابزارها و ویرایشگرها با خطا هشدار می‌دهند که نوع بازگشتی شما (مثلاً `dict`) با نوع اعلام‌شده (مثلاً مدل Pydantic) متفاوت است.

در چنین مواردی، می‌توانید از پارامتر `response_model` در دکوریتور عملیات مسیر استفاده کنید، به‌جای آنکه از نوع بازگشتی تابع استفاده کنید. می‌توانید از پارامتر `response_model` در هر کدام از عملیات مسیر استفاده کنید:

- `@app.get()`
- `@app.post()`
- `@app.put()`
- `@app.delete()`
- و غیره.

مثال:

```
from typing import Any
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list[str] = []

@app.post("/items/", response_model=Item)
async def create_item(item: Item) -> Any:
    return item

@app.get("/items/", response_model=list[Item])
async def read_items() -> Any:
    return [
        {"name": "Portal Gun", "price": 42.0},
        {"name": "Plumbus", "price": 32.0},
    ]
```



**نکته:** توجه کنید که `response_model` یک پارامتر برای متد دکوریتر (`post`, `get`، و غیره) است، نه پارامتر تابع عملیات مسیر شما، مانند پارامترهای بدنه و ورودی دیگر.

`response_model` همان نوعی را می‌پذیرد که برای فیلدهای مدل `Pydantic` تعریف می‌کنید. بنابراین می‌تواند یک مدل `Pydantic` باشد، یا مثلاً یک لیست از مدل‌های `Pydantic` مانند `List[Item]`.

`FastAPI` از `response_model` برای مستندسازی داده‌ها، اعتبارسنجی، تبدیل و فیلتر داده‌های خروجی مطابق با نوع اعلام‌شده استفاده می‌کند.

**نکته:** اگر در ویرایشگر، ابزار `mypy` یا سایر ابزارها بررسی‌های سختگیرانه نوع دارید، می‌توانید نوع بازگشتی تابع را `Any` اعلام کنید. با این کار به ویرایشگر اعلام می‌کنید که عمداً هر چیزی را باز می‌گردانید. اما `FastAPI` همچنان عملیات مستندسازی، اعتبارسنجی، فیلتر و غیره را با استفاده از `response_model` انجام می‌دهد.

## ۲-۱۷ اولویت `response_model`

اگر هم نوع بازگشتی و هم `response_model` را تعریف کنید، `response_model` اولویت دارد و توسط `FastAPI` استفاده می‌شود. به این ترتیب می‌توانید نوع‌ها را به‌درستی برای توابع خود تعریف کنید حتی اگر خروجی متفاوتی از `response_model` دارید، و در عین حال `FastAPI` بتواند از `response_model` برای اعتبارسنجی، مستندسازی و غیره استفاده کند.

همچنین می‌توانید از `response_model=None` استفاده کنید تا مدل پاسخ برای آن مسیر غیرفعال شود؛ مثلاً اگر نوع‌هایی استفاده می‌کنید که برای فیلدهای `Pydantic` معتبر نیستند. مثالی از این حالت را در یکی از بخش‌های بعدی خواهید دید.

## ۳-۱۷ بازگرداندن همان داده ورودی

در اینجا یک مدل `UserIn` تعریف می‌کنیم که شامل رمز عبور به‌صورت متن ساده است:

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None

# Don't do this in production!
@app.post("/user/")
async def create_user(user: UserIn) -> UserIn:
    return user
```

**اطلاعات:** برای استفاده از EmailStr، ابتدا باید بسته email-validator را نصب کنید. مطمئن شوید که یک محیط مجازی ساخته و فعال کرده‌اید، سپس یکی از دستورات زیر را اجرا کنید:

```
$ pip install email-validator
```

یا:

```
$ pip install "pydantic[email]"
```

در اینجا از این مدل برای تعریف ورودی و همچنین خروجی استفاده شده است:

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None

# Don't do this in production!
@app.post("/user/")
async def create_user(user: UserIn) -> UserIn:
    return user
```

در این صورت، هر زمان مرورگر کاربری را با رمز عبور ایجاد می‌کند، API همان رمز عبور را در پاسخ بازمی‌گرداند. در این مثال، شاید مشکلی پیش نیاید، چون همان کاربر رمز را ارسال کرده است. اما اگر از همین مدل برای عملیات مسیر دیگری استفاده کنیم، ممکن است رمزهای کاربران را برای همه کلاینت‌ها ارسال کنیم.

**هشدار:** هیچ‌گاه رمز عبور خام کاربران را ذخیره نکنید یا در پاسخ بازنگردانید، مگر اینکه کاملاً با تبعات آن آشنا باشید و بدانید چه می‌کنید.

## ۴-۱۷ افزودن یک مدل خروجی

در عوض می‌توانیم یک مدل ورودی با رمز عبور خام و یک مدل خروجی بدون آن ایجاد کنیم:

```
from typing import Any
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None
```

```
class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None

@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

در اینجا، اگرچه تابع عملیات مسیر، همان ورودی `user` که شامل رمز عبور است را بازمی‌گرداند:

```
from typing import Any
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None

class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None

@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
    return user
```

اما ما `response_model` را مدل `UserOut` اعلام کرده‌ایم که رمز عبور را شامل نمی‌شود:

```
from typing import Any
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None

class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None

@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn) -> Any:
```

```
return user
```

در نتیجه، FastAPI مسئول فیلتر کردن همه داده‌هایی خواهد بود که در مدل خروجی تعریف نشده‌اند (با استفاده از Pydantic).

### ۱۷-۵ response\_model یا نوع بازگشتی

در این حالت، چون دو مدل متفاوت هستند، اگر نوع بازگشتی تابع را به صورت UserOut مشخص کنیم، ویرایشگر و ابزارهای مربوطه خط خواهند داد؛ زیرا آنچه باز می‌گردد، از نوعی متفاوت با UserOut است. به همین دلیل، در این مثال باید آن را در پارامتر response\_model تعیین کنیم. اما ادامه‌ی مطلب را بخوانید تا ببینید چطور می‌توان از این محدودیت عبور کرد.

### ۱۷-۶ نوع بازگشتی و فیلتر شدن داده‌ها

از مثال قبلی ادامه می‌دهیم. می‌خواستیم نوع بازگشتی تابع را با یک نوع خاص مشخص کنیم، اما می‌خواستیم بتوانیم چیزی بازگردانیم که در واقع داده‌های بیشتری را شامل شود. ما می‌خواهیم FastAPI همچنان داده‌ها را بر اساس مدل پاسخ فیلتر کند. یعنی حتی اگر تابع داده‌های بیشتری بازگرداند، پاسخ فقط شامل فیلدهایی باشد که در مدل پاسخ تعریف شده‌اند.

در مثال قبلی، چون کلاس‌ها متفاوت بودند، مجبور بودیم از پارامتر response\_model استفاده کنیم. اما این باعث می‌شود که ویرایشگر و ابزارهای بررسی نوع، پشتیبانی لازم از نوع بازگشتی تابع را ارائه ندهند. ولی در بیشتر مواقعی که می‌خواهیم چنین کاری انجام دهیم، فقط می‌خواهیم مدل، بخشی از داده‌ها را فیلتر یا حذف کند، همان‌طور که در این مثال اتفاق می‌افتد.

در چنین مواردی، می‌توانیم از کلاس‌ها و وراثت استفاده کنیم تا از مزایای حاشیه‌نویسی‌های نوع در توابع بهره‌مند شویم، و در عین حال فیلتر شدن داده‌ها توسط FastAPI را نیز حفظ کنیم:

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class BaseUser(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None

class UserIn(BaseUser):
    password: str

@app.post("/user/")
async def create_user(user: UserIn) -> BaseUser:
    return user
```

با این کار، پشتیبانی لازم از سوی ویرایشگر و ابزارهایی مانند mypy فراهم می‌شود چون این کد از نظر نوع‌دهی صحیح است، و در عین حال فیلتر شدن داده‌ها توسط FastAPI نیز انجام می‌شود. این چطور کار می‌کند؟ اجازه دهید بررسی کنیم.

### ۱-۶-۱۷ حاشیه‌نویسی نوع و ابزارها

ابتدا ببینیم که ویرایشگر، mypy و ابزارهای دیگر چطور این را تفسیر می‌کنند. BaseUser شامل فیلدهای پایه است. سپس UserIn از BaseUser ارث‌بری می‌کند و فیلد password را اضافه می‌کند؛ بنابراین شامل همه‌ی فیلدهای هر دو کلاس خواهد بود.

ما نوع بازگشتی تابع را BaseUser مشخص می‌کنیم، اما در واقع یک شیء از نوع UserIn باز می‌گردانیم. ویرایشگر، mypy و سایر ابزارها به این ایراد نمی‌گیرند چون از نظر نوع‌دهی، UserIn یک زیرکلاس از BaseUser است، و بنابراین در جایی که BaseUser انتظار می‌رود، معتبر است.

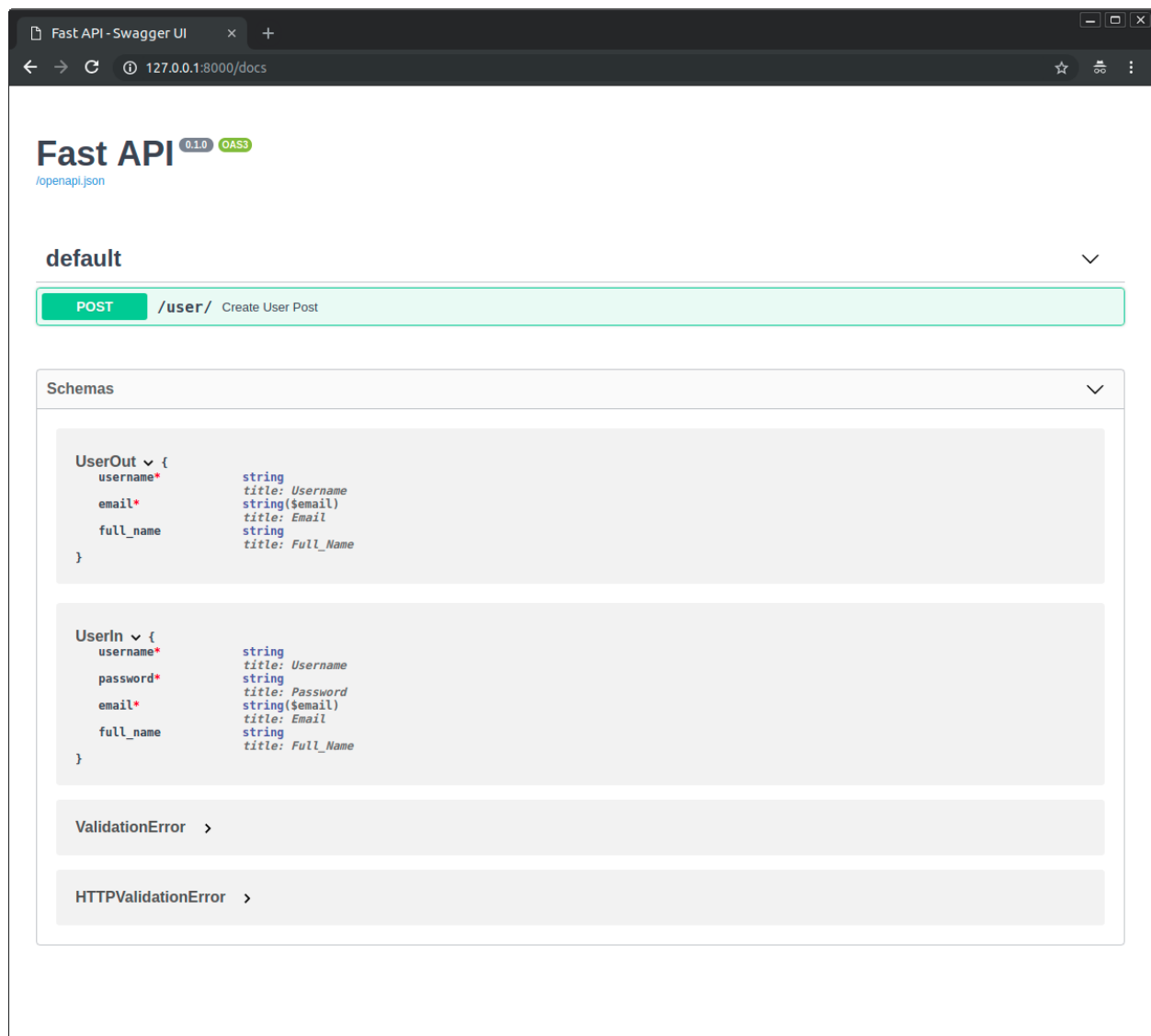
### ۲-۶-۱۷ فیلتر کردن داده‌ها توسط FastAPI

حالا، از دید FastAPI، این نوع بازگشتی را بررسی می‌کند و اطمینان حاصل می‌کند که آنچه باز می‌گردد، فقط شامل فیلدهایی است که در آن نوع تعریف شده‌اند. FastAPI در درون، چندین کار با Pydantic انجام می‌دهد تا مطمئن شود که همان قواعد وراثت کلاس‌ها در زمان فیلتر کردن داده‌ی بازگشتی اعمال نمی‌شود، در غیر این صورت ممکن بود داده‌های بیشتری از حد انتظار بازگردانده شوند.

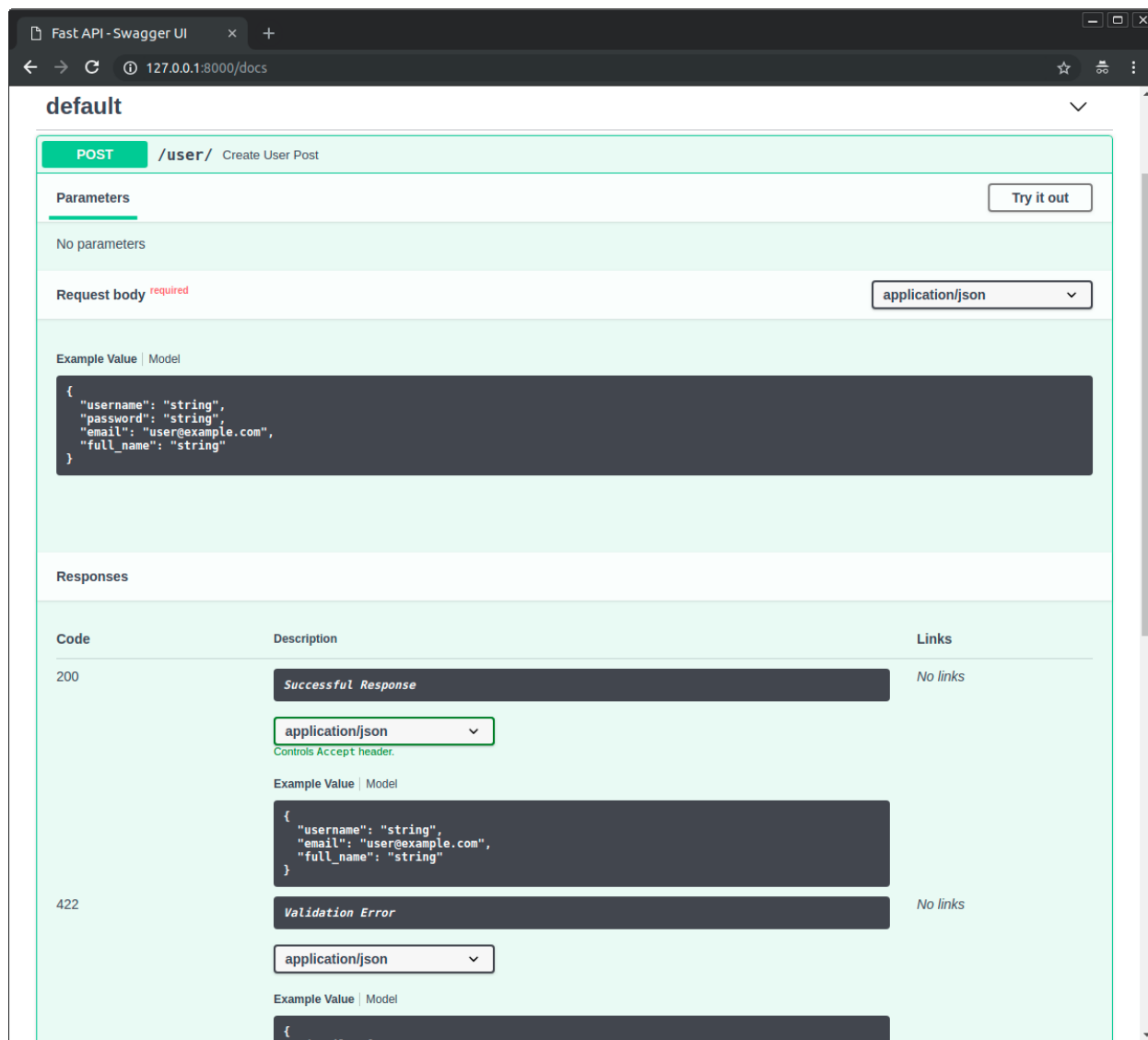
به این ترتیب، می‌توانید از هر دو جهان بهترین بهره را ببرید: حاشیه‌نویسی‌های نوع همراه با پشتیبانی ابزارها و فیلتر شدن داده‌ها.

### ۳-۶-۱۷ نمایش در مستندات

وقتی مستندات خودکار را مشاهده می‌کنید، می‌توانید ببینید که مدل ورودی و مدل خروجی، هر کدام اسکیمای JSON مخصوص خود را دارند:



و هر دو مدل در مستندات تعاملی API استفاده می شوند:



## ۸-۱۷ سایر حاشیه‌نویسی نوع بازگشتی

ممکن است در مواردی چیزی را بازگردانید که یک فیلد معتبر در `Pydantic` نیست، و فقط برای بهره‌مندی از پشتیبانی ابزارها (ویرایشگر، `mypy` و غیره) آن را در تابع حاشیه‌نویسی کرده باشید.

### ۸-۱۷-۱ بازگرداندن مستقیم `Response`

رایج‌ترین حالت این است که مستقیم یک شیء `Response` بازگردانید، همان‌طور که در مستندات پیشرفته توضیح داده شده است:

```
from fastapi import FastAPI, Response
from fastapi.responses import JSONResponse, RedirectResponse

app = FastAPI()

@app.get("/portal")
async def get_portal(teleport: bool = False) -> Response:
```

```

if teleport:
    return RedirectResponse(
        url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
return JSONResponse(content={"message": "Here's your
    interdimensionalportal."})

```

این حالت ساده به‌طور خودکار توسط FastAPI پشتیبانی می‌شود چون نوع بازگشتی یک کلاس (یا زیرکلاس) از Response است. ابزارها نیز مشکلی نخواهند داشت چون هم RedirectResponse و هم JSONResponse زیرکلاس‌های Response هستند و بنابراین حاشیه‌نویسی نوع صحیح است.

### ۲-۸-۱۷ حاشیه‌نویسی نوع با یک زیرکلاس Response

می‌توانید در نوع بازگشتی تابع، مستقیماً یک زیرکلاس از Response را استفاده کنید:

```

from fastapi import FastAPI
from fastapi.responses import RedirectResponse

app = FastAPI()

@app.get("/teleport")
async def get_teleport() -> RedirectResponse:
    return RedirectResponse(
        url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")

```

این نیز کار می‌کند چون RedirectResponse یک زیرکلاس از Response است و FastAPI این حالت ساده را به‌طور خودکار مدیریت می‌کند.

### ۳-۸-۱۷ حاشیه‌نویسی نوع بازگشتی نامعتبر

اما وقتی چیزی مانند یک شیء دلخواه (مثلاً یک شیء دیتابیس) باز می‌گردانید که یک نوع معتبر در Pydantic نیست، و آن را در حاشیه‌نویسی تابع مشخص می‌کنید، FastAPI تلاش می‌کند تا از آن یک مدل Pydantic بسازد و شکست خواهد خورد. همین اتفاق زمانی رخ می‌دهد که از Union بین چند نوع استفاده کنید که یکی یا بیشتر از آن‌ها نوعی نامعتبر در Pydantic باشند. برای مثال، این کد شکست خواهد خورد:

```

from fastapi import FastAPI, Response
from fastapi.responses import RedirectResponse

app = FastAPI()

@app.get("/portal")
async def get_portal(teleport: bool = False) -> Response | dict:
    if teleport:
        return
    RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return {"message": "Here's your interdimensional portal."}

```



این شکست می‌خورد چون حاشیه‌نویسی نوع، یک نوع Pydantic نیست و فقط یک کلاس (یا زیر کلاس) Response هم نیست، بلکه ترکیبی (Union) از Response و dict است.

#### ۴-۸-۱۷ غیرفعال کردن مدل پاسخ

در ادامه‌ی مثال بالا، ممکن است نخواهید اعتبارسنجی، مستندسازی، فیلتر کردن و ... پیش‌فرضی که FastAPI انجام می‌دهد را داشته باشید. اما ممکن است بخواهید حاشیه‌نویسی نوع بازگشتی را حفظ کنید تا از پشتیبانی ابزارهایی مثل ویرایشگرها و چک‌کننده‌های نوع (مثل mypy) بهره‌مند شوید. در این صورت می‌توانید با تعیین `response_model=None` تولید مدل پاسخ را غیرفعال کنید:

```
from fastapi import FastAPI, Response
from fastapi.responses import RedirectResponse

app = FastAPI()

@app.get("/portal", response_model=None)
async def get_portal(teleport: bool = False) -> Response | dict:
    if teleport:
        return
    RedirectResponse(url="https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    return {"message": "Here's your interdimensional portal."}
```

با این کار FastAPI تولید مدل پاسخ را نادیده می‌گیرد و در نتیجه می‌توانید هر نوع حاشیه‌نویسی نوع بازگشتی که نیاز دارید را داشته باشید بدون اینکه بر رفتار FastAPI تأثیری بگذارد.

#### ۹-۱۷ پارامترهای رمزنگاری مدل پاسخ

مدل پاسخ شما می‌تواند مقادیر پیش‌فرض داشته باشد، مانند:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5
    tags: list[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62,
    "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5,
    "tags": []},
}
```

```
@app.get("/items/{item_id}", response_model=Item,
response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

**description: Union[str, None] = None •**

(یا در پایتون ۳٫۱۰: None = None | str) مقدار پیش فرض None دارد.

**tax: float = 10.5 •**

مقدار پیش فرض ۱۰٫۵ دارد.

**tags: List[str] = [] •**

مقدار پیش فرض یک لیست خالی دارد: []

اما ممکن است بخواهید آن‌ها را در صورتی که واقعاً در داده ذخیره نشده‌اند، از نتیجه حذف کنید. برای مثال، اگر مدل‌هایی با ویژگی‌های اختیاری زیاد در پایگاه داده NoSQL دارید، اما نمی‌خواهید پاسخ JSON شما بسیار طولانی و پر از مقادیر پیش فرض باشد.

### ۱۷-۹-۱ استفاده از پارامتر `response_model_exclude_unset`

می‌توانید پارامتر `response_model_exclude_unset=True` را در دکوراتور عملیات مسیر تنظیم کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5
    tags: list[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62,
            "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5,
            "tags": []},
}

@app.get("/items/{item_id}", response_model=Item,
response_model_exclude_unset=True)
async def read_item(item_id: str):
    return items[item_id]
```

و در این صورت، مقادیر پیش فرض در پاسخ لحاظ نخواهند شد، فقط مقادیری که واقعاً تنظیم شده‌اند بازگردانده می‌شوند. بنابراین، اگر درخواست به مسیر مربوط به آیتم با شناسه foo ارسال شود، پاسخ (بدون مقادیر پیش فرض) به شکل زیر خواهد بود:

```
{
  "name": "Foo",
  "price": 50.2
}
```

**اطلاعات:** در نسخه ۱ از Pydantic، متد `dict()` استفاده می‌شد. این متد در نسخه ۲ منسوخ شده (اما همچنان پشتیبانی می‌شود) و با متد جدید `model_dump()` جایگزین شده است. مثال‌های بالا از `dict()` برای سازگاری با نسخه ۱ استفاده می‌کنند، اما در صورت امکان بهتر است از `model_dump()` در نسخه ۲ استفاده کنید.

**اطلاعات:** FastAPI از متد `dict()` مدل‌های Pydantic به همراه پارامتر `exclude_unset` برای این کار استفاده می‌کند.

**اطلاعات:** همچنین می‌توانید از موارد زیر استفاده کنید:

```
response_model_exclude_defaults=True .
```

```
response_model_exclude_none=True .
```

که در مستندات Pydantic برای `exclude_defaults` و `exclude_none` توضیح داده شده‌اند.

## ۲-۹-۱۷ داده‌هایی با مقادیر تنظیم شده برای فیلدهای دارای مقدار پیش فرض

اگر داده‌ی شما مقادیر مشخصی برای فیلدهایی با مقادیر پیش فرض داشته باشد، مانند آیتم با شناسه bar:

```
{
  "name": "Bar",
  "description": "The bartenders",
  "price": 62,
  "tax": 20.2
}
```

این مقادیر در پاسخ بازگردانده خواهند شد.

## ۳-۹-۱۷ داده‌هایی با مقادیر برابر با مقادیر پیش فرض

اگر داده‌ها مقادیری برابر با مقادیر پیش فرض داشته باشند، مانند آیتم با شناسه baz:

```
{
  "name": "Baz",
  "description": None,
```

```

    "price": 50.2,
    "tax": 10.5,
    "tags": []
}

```

FastAPI (در واقع Pydantic) به اندازه کافی هوشمند است که تشخیص دهد اگرچه `description`، `tax` و `tags` همان مقادیر پیش فرض را دارند، اما به صورت صریح تنظیم شده اند (و از پیش فرض گرفته نشده اند). بنابراین در پاسخ JSON گنجانده خواهند شد.

**نکته:** توجه داشته باشید که مقادیر پیش فرض می توانند هر چیزی باشند، نه فقط `None`. می توانند لیست (`[]`)، عدد اعشاری مانند `10.5` و غیره باشند.

### ۱۷-۱۰ `response_model_exclude` و `response_model_include`

همچنین می توانید از پارامترهای `response_model_exclude` و `response_model_include` در دکوراتور عملیات مسیر استفاده کنید. این پارامترها مجموعه ای از رشته ها (`set of str`) را دریافت می کنند که نام ویژگی هایی هستند که باید گنجانده شوند (و بقیه حذف شوند) یا حذف شوند (و بقیه گنجانده شوند). این کار می تواند راهی سریع باشد اگر فقط یک مدل Pydantic دارید و می خواهید بعضی داده ها را از خروجی حذف کنید.

**نکته:** اما همچنان توصیه می شود از راهکارهای بالا و استفاده از کلاس های متفاوت استفاده کنید. چرا که اسکیمای JSON تولید شده در OpenAPI برنامه (و مستندات) همچنان مربوط به مدل کامل خواهد بود، حتی اگر از `response_model_include` یا `response_model_exclude` برای حذف برخی ویژگی ها استفاده کنید. این نکته درباره ی `response_model_by_alias` که به روش مشابهی کار می کند نیز صدق می کند.

```

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The Bar fighters", "price": 62,
            "tax": 20.2},
    "baz": {
        "name": "Baz",
        "description": "There goes my baz",
        "price": 50.2,
        "tax": 10.5,
    }
}

```

```

    },
}

@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include={"name", "description"},
)

async def read_item_name(item_id: str):
    return items[item_id]

@app.get("/items/{item_id}/public", response_model=Item,
    response_model_exclude={"tax"})
async def read_item_public_data(item_id: str):
    return items[item_id]

```

**نکته:** سینتکس {"name", "description"} یک مجموعه (set) با این دو مقدار ایجاد می‌کند. این معادل `set(["name", "description"])` است.

### ۱۰-۱۷ استفاده از لیست به جای مجموعه

اگر فراموش کردید از set استفاده کنید و به جای آن از list یا tuple استفاده کردید، FastAPI آن را به مجموعه تبدیل کرده و همه چیز به درستی کار خواهد کرد:

```

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float = 10.5

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The Bar fighters", "price": 62,
            "tax": 20.2},
    "baz": {
        "name": "Baz",
        "description": "There goes my baz",
        "price": 50.2,
        "tax": 10.5,
    },
}

@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include=["name", "description"],
)

```

```

async def read_item_name(item_id: str):
    return items[item_id]

@app.get("/items/{item_id}/public", response_model=Item,
         response_model_exclude=["tax"])
async def read_item_public_data(item_id: str):
    return items[item_id]

```

### جمع‌بندی

- از پارامتر `response_model` در دکوراتور عملیات مسیر برای تعریف مدل پاسخ و حذف اطلاعات خصوصی استفاده کنید.
- از `response_model_exclude_unset` برای بازگرداندن فقط مقادیر صراحتاً تنظیم‌شده استفاده نمایید.

## فصل ۱۸: مدل‌های اضافی

ادامه‌ی مثالی که قبلاً دیدیم، در بسیاری از مواقع بیش از یک مدل مرتبط خواهید داشت. این مورد به‌ویژه در مدل‌های کاربری رایج است، زیرا:

- مدل ورودی باید بتواند رمز عبور را داشته باشد.
- مدل خروجی نباید رمز عبور را داشته باشد.
- مدل پایگاه داده احتمالاً باید رمز عبور هش شده را داشته باشد.

**هشدار:** هرگز رمز عبور کاربر را به‌صورت متنی ذخیره نکنید. همیشه یک «هش امن» از رمز عبور را ذخیره کنید تا بعداً بتوانید آن را بررسی (verify) کنید. اگر با مفهوم هش رمز عبور آشنایی ندارید، در فصل‌های مربوط به امنیت آن را یاد خواهید گرفت.

### ۱-۱۸ چند مدل مختلف

در ادامه یک نمای کلی از مدل‌ها با فیلدهای رمز عبور و مکان‌هایی که استفاده می‌شوند می‌بینید:

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: str | None = None

class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None

class UserInDB(BaseModel):
    username: str
    hashed_password: str
    email: EmailStr
    full_name: str | None = None

def fake_password_hasher(raw_password: str):
    return "supersecret" + raw_password

def fake_save_user(user_in: UserIn):
    hashed_password = fake_password_hasher(user_in.password)
    user_in_db = UserInDB(**user_in.dict(), hashed_password=hashed_password)
    print("User saved! ..not really")
    return user_in_db
```

```
@app.post("/user/", response_model=UserOut)
async def create_user(user_in: UserIn):
    user_saved = fake_save_user(user_in)
    return user_saved
```

**اطلاعات:** در نسخه‌ی ۱ از Pydantic، متد `dict()` وجود داشت. در نسخه‌ی ۲ این متد توصیه‌نشده ولی همچنان پشتیبانی شده محسوب می‌شود و به `model_dump()` تغییر نام داده است. مثال‌های بالا از `dict()` برای سازگاری با نسخه‌ی ۱ استفاده می‌کنند، ولی اگر از نسخه‌ی ۲ استفاده می‌کنید، بهتر است از `model_dump()` استفاده نمایید.

## ۱۸-۲ دوباره‌ی `**user_in.dict()`

### ۱۸-۲-۱ متد `dict()` در Pydantic

`user_in` یک مدل Pydantic از کلاس `UserIn` است. مدل‌های Pydantic متدی به نام `dict()` دارند که داده‌های مدل را به صورت یک دیکشنری پایتون بازمی‌گرداند. مثلاً اگر این شیء را بسازیم:

```
user_in = UserIn(username="john", password="secret",
email="john.doe@example.com")
```

و بعد فراخوانی کنیم:

```
user_dict = user_in.dict()
```

اکنون متغیر `user_dict` یک دیکشنری پایتونی حاوی داده‌های مدل است، نه یک شیء مدل. اگر بنویسیم:

```
print(user_dict)
```

ما یک دیکشنری پایتون شبیه این دریافت می‌کنیم:

```
{
    'username': 'john',
    'password': 'secret',
    'email': 'john.doe@example.com',
    'full_name': None,
}
```

### ۱۸-۲-۲ بازکردن (Unpacking) یک دیکشنری

اگر یک دیکشنری مانند `user_dict` داشته باشیم و آن را به یک تابع یا کلاس با `**user_dict` پاس بدهیم، پایتون آن را باز می‌کند، یعنی کلیدها و مقادیر را به صورت آرگومان‌های کلید-مقدار ارسال می‌کند. پس اگر بنویسیم:

```
UserInDB(**user_dict)
```



این معادل است با:

```
UserInDB(
    username="john",
    password="secret",
    email="john.doe@example.com",
    full_name=None,
)
```

یا دقیق‌تر:

```
UserInDB(
    username = user_dict["username"],
    password = user_dict["password"],
    email = user_dict["email"],
    full_name = user_dict["full_name"],
)
```

### ۳-۲-۱۸ ساختن مدل Pydantic از محتوای یک مدل دیگر

همان‌طور که در بالا `user_dict` را از `user_in.dict()` گرفتیم، کد زیر:

```
user_dict = user_in.dict()
UserInDB(**user_dict)
```

معادل است با:

```
UserInDB(**user_in.dict())
```

چون `user_in.dict()` یک دیکشنری است، و وقتی آن را با `**` به کلاس `UserInDB` می‌دهیم، پایتون آن را باز می‌کند. پس می‌توانیم یک مدل Pydantic را از داده‌های مدل دیگر بسازیم.

### ۴-۲-۱۸ باز کردن دیکشنری و اضافه کردن آرگومان اضافی

اگر بخواهیم علاوه بر باز کردن دیکشنری، یک آرگومان اضافی هم اضافه کنیم، مانند:

```
UserInDB(**user_in.dict(), hashed_password=hashed_password)
```

در واقع معادل است با:

```
UserInDB(
    username = user_dict["username"],
    password = user_dict["password"],
    email = user_dict["email"],
    full_name = user_dict["full_name"],
    hashed_password = hashed_password,
```

)

**هشدار:** توابع کمکی مانند `fake_password_hasher` و `fake_save_user` فقط برای نمایش جریان داده‌ها هستند و هیچ امنیت واقعی‌ای ارائه نمی‌دهند.

### ۳-۱۸ کاهش تکرار

کاهش تکرار کد یکی از اصول مهم در FastAPI است. زیرا تکرار کد احتمال بروز باگ، مشکلات امنیتی، ناسازگاری در کد (مثلاً آپدیت در یک بخش ولی نه در بقیه)، و موارد مشابه را افزایش می‌دهد. در مدل‌هایی که در بالا دیدیم، داده‌ها بسیار مشابه هستند و نام و نوع فیلدها تکرار شده‌اند. می‌توانیم این را بهتر انجام دهیم.

ما می‌توانیم مدلی به نام `UserBase` تعریف کنیم که به‌عنوان پایه‌ای برای سایر مدل‌های ما عمل کند. سپس می‌توانیم زیرکلاس‌هایی از این مدل بسازیم که ویژگی‌ها (اعلان نوع‌ها، اعتبارسنجی و غیره) را از آن به ارث ببرند. تمام فرایندهای تبدیل داده، اعتبارسنجی، مستندسازی و غیره همچنان به‌صورت عادی عمل خواهند کرد. با این روش، فقط تفاوت‌های میان مدل‌ها را اعلام می‌کنیم (مدلی با رمز عبور به‌صورت متنی ساده، مدلی با `hashed_password`، و مدلی بدون رمز عبور):

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserBase(BaseModel):
    username: str
    email: EmailStr
    full_name: str | None = None

class UserIn(UserBase):
    password: str

class UserOut(UserBase):
    pass

class UserInDB(UserBase):
    hashed_password: str

def fake_password_hasher(raw_password: str):
    return "supersecret" + raw_password

def fake_save_user(user_in: UserIn):
    hashed_password = fake_password_hasher(user_in.password)
    user_in_db = UserInDB(**user_in.dict(), hashed_password=hashed_password)
    print("User saved! ..not really")
    return user_in_db

@app.post("/user/", response_model=UserOut)
async def create_user(user_in: UserIn):
    user_saved = fake_save_user(user_in)
    return user_saved
```

اکنون بدون تکرار زیاد، ساختار مدل‌ها تمیزتر، امن‌تر و قابل نگهداری‌تر شده است.

#### ۱۸-۴ Union یا anyOf

می‌توانید یک پاسخ را به صورت Union (اتحاد) از دو یا چند نوع تعریف کنید، یعنی پاسخ می‌تواند هر کدام از آن‌ها باشد. در مشخصات OpenAPI این مورد با anyOf تعریف می‌شود. برای این کار، از راهنمای نوع استاندارد پایتون typing.Union استفاده کنید:

**نکته:** هنگام تعریف یک Union، نوع خاص‌تر را ابتدا قرار دهید و سپس نوع عام‌تر را بنویسید. در مثال زیر، نوع خاص‌تر یعنی PlaneItem قبل از CarItem در Union[PlaneItem, CarItem] آمده است.

```
from typing import Union
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class BaseItem(BaseModel):
    description: str
    type: str

class CarItem(BaseItem):
    type: str = "car"

class PlaneItem(BaseItem):
    type: str = "plane"
    size: int

items = {
    "item1": {"description": "All my friends drive a low rider", "type": "car"},
    "item2": {
        "description": "Music is my aeroplane, it's my aeroplane",
        "type": "plane",
        "size": 5,
    },
}

@app.get("/items/{item_id}", response_model=Union[PlaneItem, CarItem])
async def read_item(item_id: str):
    return items[item_id]
```

#### ۱۸-۴-۱ Union در پایتون ۳,۱۰

در این مثال ما Union[PlaneItem, CarItem] را به عنوان مقدار آرگومان response\_model ارسال کرده‌ایم. از آنجا که ما آن را به عنوان مقداری برای یک آرگومان (و نه در یک اعلان نوع) ارسال می‌کنیم، حتی در پایتون ۳,۱۰ نیز باید از Union استفاده کنیم. اگر در نوع‌دهی متغیر استفاده می‌شد، می‌توانستیم از علامت میله عمودی | استفاده کنیم:

```
some_variable: PlaneItem | CarItem
```

اما اگر این را در انتسابی مثل `response_model=PlaneItem | CarItem` قرار دهیم، خطا دریافت خواهیم کرد، چون پایتون تلاش می کند یک عملیات نامعتبر بین `PlaneItem` و `CarItem` انجام دهد، به جای اینکه آن را به عنوان حاشیه نویسی نوع تفسیر کند.

### ۱۸-۵ لیستی از مدل ها

به همان روش، می توانید پاسخ هایی را که لیستی از اشیاء هستند، تعریف کنید. برای این کار از `typing.List` استاندارد پایتون استفاده کنید (یا فقط `list` در پایتون ۳,۹ به بعد):

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str

items = [
    {"name": "Foo", "description": "There comes my hero"},
    {"name": "Red", "description": "It's my aeroplane"},
]

@app.get("/items/", response_model=list[Item])
async def read_items():
    return items
```

### ۱۸-۶ پاسخ با دیکشنری دلخواه

همچنین می توانید یک پاسخ را با یک دیکشنری دلخواه (بدون استفاده از مدل `Pydantic`) تعریف کنید، فقط با تعیین نوع کلیدها و مقادیر. این کار زمانی مفید است که از قبل نام های معتبر فیلدها/ویژگی ها (که در مدل های `Pydantic` لازم هستند) را نمی دانید. در این حالت می توانید از `typing.Dict` استفاده کنید (یا فقط `dict` در پایتون ۳,۹ به بعد):

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/keyword-weights/", response_model=dict[str, float])
async def read_keyword_weights():
    return {"foo": 2.3, "bar": 3.4}
```

## جمع بندی

از چند مدل مختلف Pydantic استفاده کرده و در صورت نیاز آزادانه از آن‌ها ارث‌بری کنید. نیازی نیست که فقط یک مدل داده برای هر موجودیت داشته باشید، به‌ویژه اگر آن موجودیت باید بتواند حالت‌های مختلفی داشته باشد. مانند حالت‌های مختلف یک موجودیت «کاربر» شامل password\_hash, password یا بدون رمز عبور.

## فصل ۱۹: کد وضعیت پاسخ

کد وضعیت پاسخ (**Response Status Code**) عددی سه رقمی است که همراه با هر پاسخ HTTP ارسال می شود و وضعیت آن پاسخ را مشخص می کند. این کد به کلاینت اطلاع می دهد که درخواست با موفقیت انجام شده، خطایی رخ داده یا نیاز به اقدام دیگری است. به همان شکلی که می توانید مدل پاسخ را مشخص کنید، می توانید کد وضعیت HTTP مورد استفاده برای پاسخ را نیز با پارامتر `status_code` در هر یک از عملیات های مسیر تعیین کنید:

- `@app.get()`
- `@app.post()`
- `@app.put()`
- `@app.delete()`
- و غیره.

```
from fastapi import FastAPI

app = FastAPI()

@app.post("/items/", status_code=201)
async def create_item(name: str):
    return {"name": name}
```

**نکته:** توجه کنید که `status_code` یک پارامتر برای متد «دکوراتور» (مثل `get`، `post` و غیره) است، نه برای تابع عملیات مسیر شما، مثل سایر پارامترها و بدنه.

پارامتر `status_code` یک عدد مربوط به کد وضعیت HTTP دریافت می کند.

**اطلاعات:** `status_code` می تواند به جای عدد، یک مقدار از نوع `IntEnum` نیز دریافت کند، مانند `http.HTTPStatus` پایتون.

این کار باعث می شود:

- آن کد وضعیت در پاسخ بازگردانده شود.
- در اسکیمای OpenAPI (و در نتیجه در رابط های کاربری) نیز به همان صورت مستند شود.

Fast API 0.1.0 OAS3  
/openapi.json

default

**POST** /items/ Create Item Post

**Parameters** Try it out

Name Description

**name** \* required  
string  
(query)

**Responses**

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

**نکته:** برخی از کدهای پاسخ (در بخش بعدی می بینید) نشان می دهند که پاسخ هیچ بدنه ای ندارد. FastAPI این را می داند و مستندات OpenAPI را طوری تولید می کند که اعلام کند پاسخ بدنه ای ندارد.

## ۱۹-۱ درباره کدهای وضعیت HTTP

**نکته:** اگر از قبل با کدهای وضعیت HTTP آشنا هستید، به بخش بعدی بروید.

در HTTP، شما یک کد وضعیت عددی سه‌رقمی را به عنوان بخشی از پاسخ ارسال می‌کنید. این کدها نام‌هایی دارند برای شناسایی بهتر، ولی بخش مهم همان عدد است. به طور خلاصه:

- ۱۰۰ تا ۱۹۹ برای پیام‌های «اطلاع‌رسانی» هستند. به ندرت مستقیماً از آن‌ها استفاده می‌کنید. پاسخ‌هایی با این کدها نمی‌توانند بدنه داشته باشند.
- ۲۰۰ تا ۲۹۹ برای پاسخ‌های «موفق» هستند. این دسته، بیشترین استفاده را دارند.
  - ۲۰۰، کد پیش فرض است و به این معنی است که همه چیز OK بوده.
  - مثال دیگر ۲۰۱، به معنای «ایجاد شده (Created)» است. معمولاً پس از ایجاد یک رکورد جدید در پایگاه داده استفاده می‌شود.
  - حالت خاص ۲۰۴، به معنای «بدون محتوا (No Content)» است. زمانی که پاسخی برای بازگرداندن وجود ندارد و نباید بدنه‌ای در پاسخ باشد.
- ۳۰۰ تا ۳۹۹ برای «تغییر مسیر (Redirection)» هستند. پاسخ‌هایی با این کدها ممکن است بدنه داشته باشند یا نداشته باشند، به جز ۳۰۴ («تغییری نکرده» (Not Modified) که نباید بدنه داشته باشد).
- ۴۰۰ تا ۴۹۹ برای خطاهای «سمت کلاینت» هستند. این دسته نیز بعد از موفقیت‌ها، پرکاربردترین هستند.
  - مثال ۴۰۴: برای «یافت نشد (Not Found)».
  - برای خطاهای عمومی سمت کلاینت، می‌توانید از ۴۰۰ استفاده کنید.
- ۵۰۰ تا ۵۹۹ برای خطاهای سمت سرور هستند. این کدها را معمولاً مستقیماً استفاده نمی‌کنید. اگر مشکلی در کد برنامه یا سرور پیش آید، یکی از این کدها به طور خودکار بازگردانده می‌شود.

**نکته:** برای اطلاعات بیشتر درباره هر کد وضعیت و کاربرد آن، به مستندات MDN در مورد [HTTP status codes](https://developer.mozilla.org/en-US/docs/Web/HTTP/status_codes) مراجعه کنید.

## ۲-۱۹ میان‌بری برای به خاطر سپردن نام‌ها

اجازه دهید مثال قبلی را دوباره ببینیم:

```
from fastapi import FastAPI

app = FastAPI()

@app.post("/items/", status_code=201)
async def create_item(name: str):
    return {"name": name}
```

کد ۲۰۱ به معنای "Created" است. اما نیازی نیست همه‌ی این کدها را حفظ کنید. می‌توانید از متغیرهای آماده در `fastapi.status` استفاده کنید:

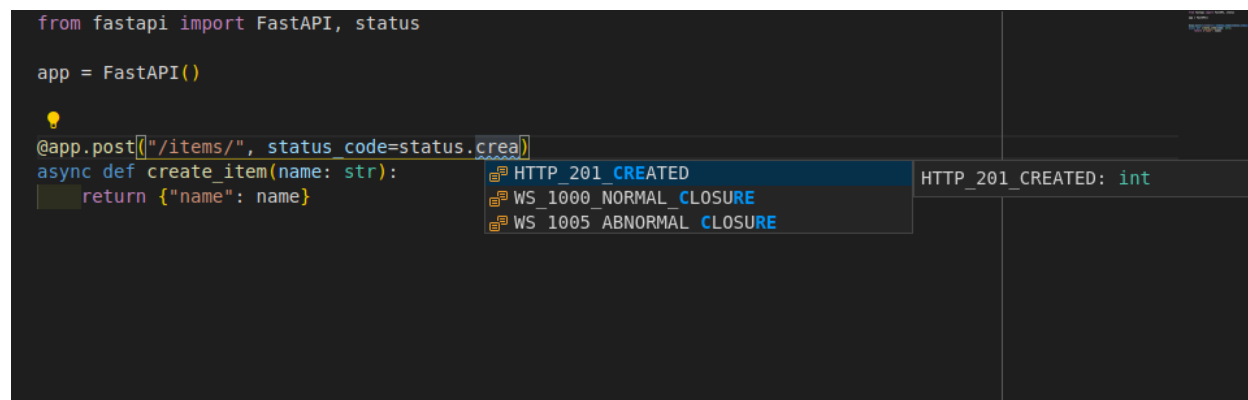


```
from fastapi import FastAPI, status

app = FastAPI()

@app.post("/items/", status_code=status.HTTP_201_CREATED)
async def create_item(name: str):
    return {"name": name}
```

این متغیرها صرفاً برای راحتی هستند و همان عدد را نگه می‌دارند، ولی این امکان را می‌دهند که از قابلیت تکمیل خود کار و پیشنهادهای ویرایشگر بهره‌مند شوید:



**جزئیات فنی:** شما می‌توانید به جای آن از `from starlette import status` هم استفاده کنید. FastAPI همان `starlette.status` را به صورت `fastapi.status` نیز ارائه می‌دهد تا برای شما، توسعه‌دهنده، راحت‌تر باشد. ولی در واقع این ماژول از Starlette می‌آید.

### ۳-۱۹ تغییر مقدار پیش‌فرض

در ادامه، در بخش راهنمای کاربران پیشرفته، خواهید دید که چگونه می‌توان پاسخی با کد وضعیت متفاوت از کد پیش‌فرض تعیین شده بازگرداند.

## فصل ۲۰: داده‌های فرم

داده‌های فرم (Form Data) اطلاعاتی هستند که کاربر از طریق فرم‌های HTML (مانند فرم ورود یا ثبت‌نام) وارد کرده و مرورگر آن‌ها را به سرور ارسال می‌کند. زمانی که به جای دریافت داده‌های JSON نیاز دارید فیلدهای فرم را دریافت کنید، می‌توانید از Form استفاده کنید.

**اطلاعات:** برای استفاده از فرم‌ها، ابتدا باید بسته‌ی python-multipart را نصب کنید. مطمئن شوید که یک محیط مجازی (virtual environment) ساخته‌اید، آن را فعال کرده‌اید، و سپس آن را نصب کرده‌اید. برای مثال:

```
$ pip install python-multipart
```

### ۱-۲۰ فرم وارد کردن

Form را از fastapi وارد کنید:

```
from typing import Annotated
from fastapi import FastAPI, Form

app = FastAPI()

@app.post("/login/")
async def login(username: Annotated[str, Form()], password: Annotated[str,
    Form()]):
    return {"username": username}
```

### ۲-۲۰ تعریف پارامترهای فرم

پارامترهای فرم را دقیقاً به همان روشی که برای Body یا Query تعریف می‌کنید، تعریف کنید:

```
from typing import Annotated
from fastapi import FastAPI, Form

app = FastAPI()

@app.post("/login/")
async def login(username: Annotated[str, Form()], password: Annotated[str,
    Form()]):
    return {"username": username}
```

برای مثال، در یکی از روش‌هایی که مشخصات OAuth2 استفاده می‌کند (که "جریان رمز عبور" یا password flow نام دارد)، الزام بر این است که username و password به‌عنوان فیلدهای فرم ارسال شوند. در این مشخصات، نام فیلدها باید دقیقاً username و password باشند و باید به‌صورت فیلدهای فرم ارسال شوند، نه JSON.

با استفاده از Form، می‌توانید همان پیکربندی‌هایی را که برای Body (و Query، Path، Cookie) انجام می‌دهید، مانند اعتبارسنجی، نمونه‌ها، نام مستعار (مثلاً user-name به جای username) و ... اعلام کنید.

**اطلاعات:** Form کلاسی است که مستقیماً از Body ارث‌بری می‌کند.

**نکته:** برای اعلام بدنه‌ی فرم، باید به‌طور صریح از Form استفاده کنید، زیرا در غیر این صورت، پارامترها به‌عنوان پارامترهای پرس‌وجو یا JSON تفسیر خواهند شد.

### ۳-۲۰ درباره‌ی فیلدهای فرم

نحوه‌ی ارسال داده‌ها به سرور توسط فرم‌های `<form></form>` HTML معمولاً از یک نوع کدگذاری خاص استفاده می‌کند که با JSON متفاوت است. FastAPI مطمئن می‌شود که این داده‌ها را از محل درست (و نه از JSON) بخواند.

**جزئیات فنی:** داده‌های فرم معمولاً با استفاده از نوع رسانه‌ای `application/x-www-form-urlencoded` (media type) کدگذاری می‌شوند. اما زمانی که فرم شامل فایل باشد، با `multipart/form-data` کدگذاری می‌شود. در فصل بعدی درباره‌ی مدیریت فایل‌ها توضیح داده خواهد شد. اگر می‌خواهید بیشتر درباره‌ی این نوع کدگذاری‌ها و فیلدهای فرم بدانید، به مستندات وب MDN برای متد POST مراجعه کنید.

**هشدار:** شما می‌توانید چندین پارامتر Form را در یک مسیر تعریف کنید، اما نمی‌توانید همزمان فیلدهایی از نوع Body (که انتظار می‌رود به‌صورت JSON ارسال شوند) نیز تعریف کنید، چرا که درخواست دارای بدنه‌ای با کدگذاری `application/x-www-form-urlencoded` خواهد بود، نه `application/json`. این محدودیت مربوط به FastAPI نیست، بلکه بخشی از پروتکل HTTP است.

### جمع‌بندی

برای اعلام پارامترهایی که از داده‌های فرم دریافت می‌شوند، از Form استفاده کنید.

## فصل ۲۱: مدل‌های فرم

مدل‌های فرم (Form Models) در FastAPI به شما این امکان را می‌دهند که فیلدهای دریافتی از فرم‌های HTML را با استفاده از مدل‌های Pydantic تعریف و اعتبارسنجی کنید. این کار باعث ساده‌سازی پردازش داده‌های فرم و تضمین صحت آن‌ها می‌شود.

**اطلاعات:** برای استفاده از فرم‌ها، ابتدا باید پکیج python-multipart را نصب کنید. مطمئن شوید که یک محیط مجازی (virtual environment) ایجاد و فعال کرده‌اید و سپس آن را نصب کنید. برای مثال:

```
$ pip install python-multipart
```

**نکته:** این ویژگی از نسخه‌ی 0.113.0 FastAPI پشتیبانی می‌شود.

### ۲۱-۱ مدل‌های Pydantic برای فرم‌ها

تنها کاری که باید انجام دهید این است که یک مدل Pydantic تعریف کنید با فیلدهایی که می‌خواهید از طریق فرم دریافت کنید، و سپس پارامتر را با Form اعلام نمایید:

```
from typing import Annotated
from fastapi import FastAPI, Form
from pydantic import BaseModel

app = FastAPI()

class FormData(BaseModel):
    username: str
    password: str

@app.post("/login/")
async def login(data: Annotated[FormData, Form()]):
    return data
```

FastAPI داده‌ی هر فیلد را از اطلاعات فرم در درخواست استخراج کرده و مدل Pydantic تعریف‌شده‌ی شما را برمی‌گرداند.

### ۲۱-۲ بررسی در مستندات

می‌توانید این مورد را در رابط مستندات موجود در مسیر docs/ بررسی کنید:

### ۳-۲۱ جلوگیری از ارسال فیلدهای اضافه در فرم

در برخی کاربردهای خاص (که احتمالاً چندان رایج نیستند)، ممکن است بخواهید فیلدهای فرم را فقط به همان‌هایی محدود کنید که در مدل Pydantic تعریف شده‌اند و هر فیلد اضافی را غیرمجاز اعلام نمایید.

**نکته:** این ویژگی از نسخه‌ی 0.114.0 FastAPI پشتیبانی می‌شود.

می‌توانید از پیکربندی مدل Pydantic برای جلوگیری از پذیرش فیلدهای اضافه استفاده کنید:

```
from typing import Annotated
from fastapi import FastAPI, Form
from pydantic import BaseModel

app = FastAPI()

class FormData(BaseModel):
    username: str
    password: str
    model_config = {"extra": "forbid"}

@app.post("/login/")
async def login(data: Annotated[FormData, Form()]):
    return data
```

اگر کلاینت تلاش کند تا اطلاعات اضافی ارسال کند، پاسخ خطایی دریافت خواهد کرد. برای مثال، اگر کلاینت بخواهد فیلدهای فرم زیر را ارسال کند:

```
username: Rick
password: Portal Gun
extra: Mr. Poopybutthole
```

پاسخی مانند زیر دریافت خواهد کرد که اعلام می‌کند فیلد extra مجاز نیست:

```
{
  "detail": [
    {
      "type": "extra_forbidden",
      "loc": ["body", "extra"],
      "msg": "Extra inputs are not permitted",
      "input": "Mr. Poopybutthole"
    }
  ]
}
```

### خلاصه

می‌توانید از مدل‌های Pydantic برای تعریف فیلدهای فرم در FastAPI استفاده کنید.

## فصل ۲۲: فایل درخواست

می‌توانید فایل‌هایی را که قرار است کاربر آپلود کند، با استفاده از File تعریف کنید.

**اطلاعات:** برای دریافت فایل‌های آپلودشده، ابتدا باید بسته‌ی python-multipart را نصب کنید. مطمئن شوید که یک محیط مجازی (virtual environment) ایجاد کرده، آن را فعال کرده‌اید و سپس آن را نصب می‌کنید. برای مثال:

```
$ pip install python-multipart
```

زیرا فایل‌های آپلودشده به صورت «فرم دیتا» (form data) ارسال می‌شوند.

### ۲۲-۱ وارد کردن File

File و UploadFile را از fastapi وارد کنید:

```
from typing import Annotated
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(file: Annotated[bytes, File()]):
    return {"file_size": len(file)}

@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile):
    return {"filename": file.filename}
```

### ۲۲-۲ تعریف پارامترهای فایل

پارامترهای فایل را همانند پارامترهای Body یا Form تعریف کنید:

```
from typing import Annotated
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(file: Annotated[bytes, File()]):
    return {"file_size": len(file)}

@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile):
    return {"filename": file.filename}
```

**اطلاعات:** File کلاسی است که مستقیماً از Form ارث‌بری می‌کند. اما به یاد داشته باشید که وقتی File، Path، Query و سایر موارد را از fastapi وارد می‌کنید، در واقع توابعی هستند که نمونه‌هایی از کلاس‌هایی با همان نام را باز می‌گردانند.

**نکته:** برای تعریف بدنه‌هایی از نوع فایل، باید از File استفاده کنید، زیرا در غیر این صورت پارامترها به عنوان پارامترهای پرس‌وجو یا بدنه (از نوع JSON) تفسیر خواهند شد.

فایل‌ها به صورت «فرم دیتا» آپلود می‌شوند. اگر نوع پارامتر تابع مسیر را bytes تعریف کنید، FastAPI فایل را برای شما می‌خواند و محتوای آن را به صورت bytes دریافت می‌کند. به خاطر داشته باشید که این یعنی کل محتوای فایل در حافظه بارگذاری می‌شود. این روش برای فایل‌های کوچک مناسب است. اما در بسیاری از موارد استفاده از UploadFile مزایای بیشتری دارد.

### ۲۲-۳ پارامتر فایل با UploadFile

پارامتر فایل را با نوع UploadFile تعریف کنید:

```
from typing import Annotated
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(file: Annotated[bytes, File()]):
    return {"file_size": len(file)}

@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile):
    return {"filename": file.filename}
```

مزایای استفاده از UploadFile نسبت به bytes:

- نیازی نیست در مقدار پیش فرض پارامتر از File() استفاده کنید.
- از فایل‌های spooled استفاده می‌کند:
  - فایلی که تا یک حد مشخص در حافظه نگهداری می‌شود و پس از آن در دیسک ذخیره می‌شود.
  - این بدان معناست که برای فایل‌های بزرگ مانند تصاویر، ویدئوها یا فایل‌های باینری بزرگ به خوبی عمل می‌کند بدون آنکه حافظه را اشغال کند.
- می‌توانید فراداده‌ی فایل آپلودشده را دریافت کنید.
- رابط async شبیه به فایل دارد.
- یک شیء واقعی از نوع SpooledTemporaryFile پایتون را فراهم می‌کند که می‌توانید مستقیماً به کتابخانه‌هایی که شیء شبیه فایل نیاز دارند، بدهید.

#### ۲۲-۳-۱ UploadFile



UploadFile دارای ویژگی‌های زیر است:

- filename: رشته‌ای با نام فایل اصلی آپلودشده (مثلاً myimage.jpg)
- content\_type: رشته‌ای شامل نوع محتوا (MIME/media type) (مثلاً image/jpeg)
- file: شیء SpooledTemporaryFile (شبه به فایل). این شیء واقعی فایل پایتون است که می‌توانید مستقیماً به سایر توابع یا کتابخانه‌هایی که انتظار فایل دارند بدهید.

UploadFile متدهای async زیر را دارد که همه آن‌ها در واقع متدهای همان فایل داخلی را فراخوانی می‌کنند:

- write(data): نوشتن داده در فایل. (bytes یا str)
  - read(size): خواندن اندازه بایت/کاراکتر از فایل. (int)
  - seek(offset): رفتن به موقعیت بایتی مشخص offset در فایل. (int)
- برای مثال await myfile.seek(0) به ابتدای فایل می‌رود.
- این برای زمانی مفید است که یک بار await myfile.read() را انجام داده‌اید و می‌خواهید دوباره محتوا را بخوانید.
- close(): بستن فایل

چون این‌ها متدهای async هستند، باید آن‌ها را با await فراخوانی کنید. مثلاً، درون یک تابع مسیر async می‌توانید محتوای فایل را این‌گونه بگیرید:

```
contents = await myfile.read()
```

اگر در یک تابع معمولی def هستید، می‌توانید مستقیماً به UploadFile.file دسترسی داشته باشید:

```
contents = myfile.file.read()
```

**جزئیات فنی async:** وقتی از متدهای async استفاده می‌کنید، FastAPI آن‌ها را در یک ThreadPool اجرا می‌کند و منتظر نتیجه می‌ماند.

**جزئیات فنی Starlette:** UploadFile در FastAPI مستقیماً از UploadFile در Starlette ارث‌بری می‌کند، اما اجزایی را اضافه می‌کند تا با Pydantic و سایر بخش‌های FastAPI سازگار باشد.

۴-۲۲ فرم دیتا چیست؟

فرم‌های HTML (مثلاً `<form></form>`) داده‌ها را به صورت خاصی به سرور ارسال می‌کنند که با JSON متفاوت است. FastAPI مطمئن می‌شود که این داده‌ها را از جای درست (به جای JSON) بخواند.

**جزئیات فنی:** داده‌های فرم، زمانی که شامل فایل نیستند، معمولاً با `media type application/x-www-form-urlencoded` رمزگذاری می‌شوند. اما وقتی فرم شامل فایل باشد، با `multipart/form-data` رمزگذاری می‌شود. اگر از File استفاده کنید، FastAPI متوجه می‌شود که فایل‌ها باید از بخش مناسب بدنه خوانده شوند. برای آشنایی بیشتر با این رمزگذاری‌ها و فیلدهای فرم، می‌توانید به مستندات MDN درباره POST مراجعه کنید.

**هشدار:** می‌توانید چندین پارامتر File و Form را در یک تابع مسیر تعریف کنید، اما نمی‌توانید به طور هم‌زمان فیلدهای Body را که انتظار دارید به صورت JSON ارسال شوند، تعریف کنید، چون بدنه‌ی درخواست با `multipart/form-data` رمزگذاری می‌شود، نه `application/json`. این محدودیت FastAPI نیست، بلکه بخشی از پروتکل HTTP است.

## ۵-۲۲ آپلود اختیاری فایل

می‌توانید با استفاده از حاشیه‌نویسی نوع استاندارد و تنظیم مقدار پیش فرض آن به None آپلود یک فایل را اختیاری کنید:

```
from typing import Annotated
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(file: Annotated[bytes | None, File()] = None):
    if not file:
        return {"message": "No file sent"}
    else:
        return {"file_size": len(file)}

@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile | None = None):
    if not file:
        return {"message": "No upload file sent"}
    else:
        return {"filename": file.filename}
```

## ۶-۲۲ UploadFile همراه با فراداده‌ی اضافی

می‌توانید از `File()` همراه با `UploadFile` نیز استفاده کنید، مثلاً برای تنظیم فراداده‌های اضافی:

```
from typing import Annotated
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")
```

```

async def create_file(file: Annotated[bytes, File(description="A file read as
    bytes)]):
    return {"file_size": len(file)}

@app.post("/uploadfile/")
async def create_upload_file(
    file: Annotated[UploadFile, File(description="A file read as UploadFile")],
):
    return {"filename": file.filename}

```

## ۷-۲۲ آپلود چند فایل

امکان آپلود چند فایل به صورت هم‌زمان وجود دارد. این فایل‌ها به یک «فیلد فرم» یکسان نسبت داده می‌شوند که با استفاده از «فرم دیتا» ارسال شده است. برای استفاده از این قابلیت، یک لیست از bytes یا UploadFile تعریف کنید:

```

from typing import Annotated
from fastapi import FastAPI, File, UploadFile
from fastapi.responses import HTMLResponse

app = FastAPI()

@app.post("/files/")
async def create_files(files: Annotated[list[bytes], File()]):
    return {"file_sizes": [len(file) for file in files]}

@app.post("/uploadfiles/")
async def create_upload_files(files: list[UploadFile]):
    return {"filenames": [file.filename for file in files]}

@app.get("/")
async def main():
    content = """
<body>
<form action="/files/" enctype="multipart/form-data" method="post">
<input name="files" type="file" multiple>
<input type="submit">
</form>
<form action="/uploadfiles/" enctype="multipart/form-data" method="post">
<input name="files" type="file" multiple>
<input type="submit">
</form>
</body>
    """
    return HTMLResponse(content=content)

```

شما همان‌طور که تعریف کرده‌اید، یک لیست از bytes یا UploadFile دریافت خواهید کرد.

**جزئیات فنی:** می‌توانستید از `from starlette.responses import HTMLResponse` نیز استفاده کنید. FastAPI همان کلاس‌های پاسخ‌دهی مربوط به `Starlette` را تحت `fastapi.responses` ارائه می‌دهد تا کار شما را راحت‌تر کند. اما بیشتر پاسخ‌های موجود، مستقیماً از `Starlette` می‌آیند.

## ۸-۲۲ آپلود چند فایل همراه با فراداده‌ی اضافی

به همان شکلی که پیش‌تر دیدید، می‌توانید از `File()` برای تنظیم پارامترهای اضافی حتی برای `UploadFile` نیز استفاده کنید:

```
from typing import Annotated
from fastapi import FastAPI, File, UploadFile
from fastapi.responses import HTMLResponse

app = FastAPI()

@app.post("/files/")
async def create_files(
    files: Annotated[list[bytes], File(description="Multiple files as bytes")],
):
    return {"file_sizes": [len(file) for file in files]}

@app.post("/uploadfiles/")
async def create_upload_files(
    files: Annotated[
        list[UploadFile], File(description="Multiple files as UploadFile")
    ],
):
    return {"filenames": [file.filename for file in files]}

@app.get("/")
async def main():
    content = """
<body>
<form action="/files/" enctype="multipart/form-data" method="post">
<input name="files" type="file" multiple>
<input type="submit">
</form>
<form action="/uploadfiles/" enctype="multipart/form-data" method="post">
<input name="files" type="file" multiple>
<input type="submit">
</form>
</body>
"""
    return HTMLResponse(content=content)
```

## جمع‌بندی

از `File`، `bytes` و `UploadFile` برای تعریف فایل‌هایی که قرار است در درخواست ارسال شوند (با استفاده از فرم دیتا)، استفاده کنید.

## فصل ۲۳: فرم‌ها و فایل‌های درخواست

می‌توانید فایل‌ها و فیلدهای فرم را هم‌زمان با استفاده از File و Form تعریف کنید.

**اطلاعات:** برای دریافت فایل‌های آپلود شده و/یا داده‌های فرم، ابتدا باید بسته‌ی python-multipart را نصب کنید. حتماً یک محیط مجازی (virtual environment) ایجاد، آن را فعال، و سپس نصب را انجام دهید، برای مثال:

```
$ pip install python-multipart
```

### ۲۳-۱ وارد کردن Form و File

```
from typing import Annotated
from fastapi import FastAPI, File, Form, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(
    file: Annotated[bytes, File()],
    fileb: Annotated[UploadFile, File()],
    token: Annotated[str, Form()],
):
    return {
        "file_size": len(file),
        "token": token,
        "fileb_content_type": fileb.content_type,
    }
```

### ۲۳-۲ تعریف پارامترهای Form و File

پارامترهای فایل و فرم را به همان روشی تعریف کنید که برای Body یا Query انجام می‌دهید:

```
from typing import Annotated
from fastapi import FastAPI, File, Form, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(
    file: Annotated[bytes, File()],
    fileb: Annotated[UploadFile, File()],
    token: Annotated[str, Form()],
):
    return {
        "file_size": len(file),
        "token": token,
        "fileb_content_type": fileb.content_type,
```

```
}
```

فایل‌ها و فیلدهای فرم به صورت داده‌های فرم ارسال می‌شوند و شما این فایل‌ها و فیلدهای فرم را دریافت خواهید کرد. همچنین می‌توانید بعضی از فایل‌ها را به صورت bytes و برخی را به صورت UploadFile تعریف کنید.

**هشدار:** می‌توانید چندین پارامتر File و Form را در یک عملیات مسیر تعریف کنید، اما نمی‌توانید در کنار آن‌ها فیلدهای Body تعریف کنید که انتظار دارید به صورت JSON دریافت شوند، زیرا بدنه‌ی درخواست با استفاده از multipart/form-data رمزگذاری خواهد شد، نه application/json. این محدودیت از FastAPI نیست، بلکه بخشی از پروتکل HTTP است.

### جمع‌بندی

زمانی که نیاز دارید داده‌ها و فایل‌ها را در یک درخواست دریافت کنید، از File و Form به صورت ترکیبی استفاده کنید.

## فصل ۲۴: مدیریت خطاها

در بسیاری از موارد، نیاز دارید که خطایی را به کلاینتی که از API شما استفاده می‌کند اطلاع دهید. این کلاینت می‌تواند یک مرورگر با رابط کاربری فرانت‌اند، کدی نوشته‌شده توسط شخصی دیگر، یک دستگاه IoT و غیره باشد. ممکن است لازم باشد به کلاینت اطلاع دهید که:

- کلاینت برای انجام آن عملیات، دسترسی یا مجوز کافی ندارد.
- کلاینت به منبع موردنظر دسترسی ندارد.
- آیتمی که کلاینت تلاش می‌کند به آن دست یابد، وجود ندارد.
- و موارد مشابه.

در چنین مواردی، معمولاً باید یک کد وضعیت HTTP در بازه ۴۰۰ (از ۴۰۰ تا ۴۹۹) بازگردانید. این مشابه کدهای وضعیت HTTP در بازه ۲۰۰ (از ۲۰۰ تا ۲۹۹) است. این کدهای "۲۰۰" به این معنا هستند که درخواست به نحوی با موفقیت انجام شده است. کدهای وضعیت در بازه ۴۰۰ به این معنا هستند که خطا از سمت کلاینت رخ داده است. همان خطاهای معروف ۴۰۴ "Not Found" را به یاد دارید؟ (و شوخی‌های مرتبط با آن؟)

### ۱-۲۴ استفاده از HTTPException

برای بازگرداندن پاسخ‌های HTTP حاوی خطا به کلاینت، از کلاس HTTPException استفاده می‌شود.

#### ۱-۱-۲۴ وارد کردن HTTPException

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}

@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}
```

### ۲-۱-۲۴ ایجاد یک استثنای HTTPException در کد

HTTPException یک استثنا در پایتون است که دارای داده‌های اضافی مربوط به API است. از آنجا که این یک استثنای پایتون است، نباید آن را برگردانید (return)، بلکه باید آن را ایجاد کنید (raise).

این همچنین بدان معناست که اگر این استثنا را در درون تابعی کمکی رخ دهد که از درون یک تابع مسیر فراخوانی شده، باقی‌مانده‌ی کد اجرا نخواهد شد و همان لحظه خطای HTTP حاصل از HTTPException برای کلاینت ارسال می‌شود.. مزیت ایجاد استثنا نسبت به بازگرداندن مقدار، در بخش وابستگی‌ها و امنیت، بیشتر آشکار خواهد شد. در مثال زیر، زمانی که کلاینت یک آیتم را با شناسه‌ای درخواست می‌کند که وجود ندارد، یک استثنا با کد وضعیت ۴۰۴ ایجاد می‌شود:

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}

@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}
```

### ۳-۱-۲۴ پاسخ نهایی

اگر کلاینت درخواست <http://example.com/items/foo> را ارسال کند، (که foo وجود دارد)، پاسخ زیر را دریافت می‌کند:

```
{
  "item": "The Foo Wrestlers"
}
```

اما اگر کلاینت درخواست <http://example.com/items/bar> را ارسال کند (که bar وجود ندارد)، پاسخ زیر با کد وضعیت ۴۰۴ ارسال خواهد شد:

```
{
  "detail": "Item not found"
}
```

**نکته:** هنگام ایجاد HTTPException، می‌توانید هر مقدار قابل تبدیل به JSON را به عنوان پارامتر detail ارسال کنید، نه فقط رشته‌ها. می‌توانید یک دیکشنری، لیست و... ارسال کنید. FastAPI به صورت خودکار آن‌ها را به JSON تبدیل می‌کند.

### ۲-۲۴ افزودن هدرهای سفارشی

در برخی موارد خاص (مانند مسائل امنیتی)، ممکن است نیاز به افزودن هدرهای سفارشی به خطا داشته باشید. گرچه در اغلب موارد نیازی به این کار نخواهید داشت، اما در صورت نیاز، می‌توانید به شکل زیر عمل کنید:

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}
```



```
@app.get("/items-header/{item_id}")
async def read_item_header(item_id: str):
    if item_id not in items:
        raise HTTPException(
            status_code=404,
            detail="Item not found",
            headers={"X-Error": "There goes my error"},
        )
    return {"item": items[item_id]}
```

### ۳-۲۴ نصب هندلرهای استثنای سفارشی

می‌توانید با استفاده از ابزارهای Starlette هندلرهای (handler) سفارشی برای استثناها تعریف کنید. فرض کنیم شما یک استثنای سفارشی به نام UnicornException دارید که ممکن است توسط خودتان یا یک کتابخانه استفاده شود. برای مدیریت این استثنا به صورت سراسری در FastAPI، می‌توانید از دکوراتور `@app.exception_handler()` استفاده کنید:

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

class UnicornException(Exception):
    def __init__(self, name: str):
        self.name = name

app = FastAPI()

@app.exception_handler(UnicornException)
async def unicorn_exception_handler(request: Request, exc: UnicornException):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} did something. There goes a rainbow..."},
    )

@app.get("/unicorns/{name}")
async def read_unicorn(name: str):
    if name == "yolo":
        raise UnicornException(name=name)
    return {"unicorn_name": name}
```

در این مثال، اگر درخواست `/unicorns/yolo` ارسال شود، استثنای `UnicornException` ایجاد می‌شود. اما این استثنا توسط `unicorn_exception_handler` مدیریت خواهد شد و خروجی زیر با کد وضعیت ۴۱۸ ارسال می‌شود:

```
{"message": "Oops! yolo did something. There goes a rainbow..."}
```

**جزئیات فنی:** همچنین می‌توانید از ماژول `Starlette` به صورت مستقیم استفاده کنید:

```
from starlette.requests import Request
```

```
from starlette.responses import JSONResponse
```

FastAPI این کلاس‌ها را به صورت داخلی از Starlette وارد می‌کند و در اختیار شما قرار می‌دهد، اما منبع اصلی آن‌ها خود Starlette است.

#### ۴-۲۴ بازنویسی هندلرهای پیش فرض استثناها

FastAPI دارای هندلرهای پیش فرض برای مدیریت خطاهای متداول مانند `HTTPException` و خطاهای اعتبارسنجی داده‌ها است. شما می‌توانید این هندلرها را بازنویسی (`override`) کنید.

##### ۴-۲۴-۱ بازنویسی خطاهای اعتبارسنجی درخواست‌ها

زمانی که داده‌ی درخواست نامعتبر باشد، FastAPI به صورت داخلی استثنای `RequestValidationError` را ایجاد می‌کند. و یک هندلر پیش فرض نیز برای آن دارد. برای بازنویسی آن مانند کد زیر عمل کنید. هندلر استثنای یک شیء `Request` و خود استثنا را دریافت خواهد کرد:

```
from fastapi import FastAPI, HTTPException
from fastapi.exceptions import RequestValidationError
from fastapi.responses import PlainTextResponse
from starlette.exceptions import HTTPException as StarletteHTTPException

app = FastAPI()

@app.exception_handler(StarletteHTTPException)
async def http_exception_handler(request, exc):
    return PlainTextResponse(str(exc.detail), status_code=exc.status_code)

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    return PlainTextResponse(str(exc), status_code=400)

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 3:
        raise HTTPException(status_code=418, detail="Nope! I don't like 3.")
    return {"item_id": item_id}
```

در این مثال، اگر به مسیر `/items/foo` بروید، به جای دریافت خطای پیش فرض JSON مانند:

```
{
  "detail": [
    {
      "loc": ["path", "item_id"],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ]
}
```

پاسخ به صورت متن ساده خواهد بود:

```
1 validation error
path -> item_id
  value is not a valid integer (type=type_error.integer)
```

## ۲-۴-۲ تفاوت RequestValidationError با ValidationError

**هشدار:** این ها جزئیات فنی هستند که اگر در حال حاضر برای شما اهمیت ندارند، می توانید از آن ها عبور کنید.

RequestValidationError یک زیر کلاس از ValidationError مربوط به Pydantic است. FastAPI از این استثنا استفاده می کند تا اگر از یک مدل Pydantic در پارامتر response\_model استفاده کرده اید و داده ی شما دارای خطا است، خطا در لاگ ثبت شود. اما کلاینت/کاربر آن را نخواهد دید. بلکه، کلاینت یک خطای عمومی "خطای داخلی سرور" با کد وضعیت HTTP شماره ۵۰۰ دریافت خواهد کرد.

این رفتار عمدی است، چرا که اگر خطای ValidationError مربوط به Pydantic در پاسخ یا هر نقطه ای از کد شما (نه در داده های دریافتی از کاربر) رخ دهد، در واقع یک باگ در کد شما است. و تا زمانی که این خطا را اصلاح نکرده اید، کاربران نباید به اطلاعات داخلی خطا دسترسی داشته باشند، زیرا این می تواند منجر به آسیب پذیری امنیتی شود.

## ۵-۲۴ باز نویسی هندلر خطای HTTPException

به همان شیوه ای که برای RequestValidationError عمل می کنید، می توانید هندلر پیش فرض HTTPException را هم باز نویسی کنید. برای مثال، ممکن است بخواهید به جای ارسال پاسخ JSON، یک پاسخ متنی ساده ارسال شود:

```
from fastapi import FastAPI, HTTPException
from fastapi.exceptions import RequestValidationError
from fastapi.responses import PlainTextResponse
from starlette.exceptions import HTTPException as StarletteHTTPException

app = FastAPI()

@app.exception_handler(StarletteHTTPException)
async def http_exception_handler(request, exc):
    return PlainTextResponse(str(exc.detail), status_code=exc.status_code)

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    return PlainTextResponse(str(exc), status_code=400)

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 3:
        raise HTTPException(status_code=418, detail="Nope! I don't like 3.")
    return {"item_id": item_id}
```

**جزئیات فنی:** همچنین می‌توانید از PlainTextResponse به صورت زیر استفاده کنید:

```
from starlette.responses import PlainTextResponse
```

FastAPI این کلاس‌ها را از Starlette وارد کرده و به صورت مستقیم در اختیار توسعه‌دهنده قرار می‌دهد، اما بیشتر کلاس‌های مربوط به پاسخ‌ها مستقیماً متعلق به Starlette هستند.

## ۶-۲۴ استفاده از محتوای بدنه در RequestValidationError

کلاس RequestValidationError شامل بدنه‌ی دریافتی با داده‌های نامعتبر نیز هست. می‌توانید این ویژگی را هنگام توسعه برای لاگ‌برداری، اشکال‌زدایی، یا حتی بازگرداندن آن به کاربر استفاده کنید:

```
from fastapi import FastAPI, Request, status
from fastapi.encoders import jsonable_encoder
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse
from pydantic import BaseModel

app = FastAPI()

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc:
    RequestValidationError):
    return JSONResponse(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        content=jsonable_encoder({
            "detail": exc.errors(),
            "body": exc.body
        }),
    )

class Item(BaseModel):
    title: str
    size: int

@app.post("/items/")
async def create_item(item: Item):
    return item
```

اگر داده‌ای نامعتبر مانند زیر ارسال شود:

```
{
  "title": "towel",
  "size": "XL"
}
```

پاسخ دریافتی به این صورت خواهد بود:

```
{
  "detail": [
    {
      "loc": ["body", "size"],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ],
  "body": {
    "title": "towel",
    "size": "XL"
  }
}
```

### ۲۴-۶-۱ تفاوت HTTPException در FastAPI و Starlette

FastAPI کلاس مخصوص خود را برای HTTPException دارد. کلاس HTTPException در FastAPI در واقع از کلاس HTTPException در Starlette ارث‌بری می‌کند. تفاوت اصلی آن‌ها در این است که:

- در FastAPI می‌توانید هر داده قابل تبدیل به JSON را برای فیلد detail استفاده کنید.
- اما در Starlette، فقط رشته برای detail قابل استفاده است.

بنابراین، در کد خود می‌توانید همچنان از HTTPException مربوط به FastAPI استفاده کنید. اما اگر می‌خواهید هندلر استثنا ثبت کنید، باید آن را برای کلاس HTTPException متعلق به Starlette ثبت کنید. در مثال بالا، برای اینکه از هر دو نوع HTTPException به‌طور هم‌زمان استفاده شود، کلاس Starlette به این صورت تغییر نام داده شده است:

```
from starlette.exceptions import HTTPException as StarletteHTTPException
```

### ۲۴-۷ استفاده مجدد از هندلرهای پیش‌فرض FastAPI

اگر بخواهید از استثناها استفاده کرده و هم‌زمان از هندلرهای پیش‌فرض FastAPI نیز بهره ببرید، می‌توانید آن‌ها را از fastapi.exception\_handlers وارد کنید:

```
from fastapi import FastAPI, HTTPException
from fastapi.exception_handlers import (
    http_exception_handler,
    request_validation_exception_handler,
)
from fastapi.exceptions import RequestValidationError
from starlette.exceptions import HTTPException as StarletteHTTPException

app = FastAPI()

@app.exception_handler(StarletteHTTPException)
async def custom_http_exception_handler(request, exc):
    print(f"OMG! An HTTP error!: {repr(exc)}")
    return await http_exception_handler(request, exc)
```

```
@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    print(f"OMG! The client sent invalid data!: {exc}")
    return await request_validation_exception_handler(request, exc)

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 3:
        raise HTTPException(status_code=418, detail="Nope! I don't like 3.")
    return {"item_id": item_id}
```

در این مثال، فقط خطاها را با پیام‌های نمایشی خاص چاپ می‌کنیم، اما مفهوم کلی این است که می‌توانید پس از پردازش اولیه، همچنان از هندلرهای پیش‌فرض FastAPI استفاده کنید.

## فصل ۲۵: پیکربندی عملیات مسیر

پارامترهای مختلفی وجود دارد که می‌توانید برای پیکربندی عملیات مسیر خود به دکوراتور آن ارسال کنید.

**هشدار:** توجه داشته باشید که این پارامترها مستقیماً به دکوراتور عملیات مسیر ارسال می‌شوند، نه به تابع مربوط به آن عملیات مسیر.

### ۲۵-۱ کد وضعیت پاسخ

می‌توانید کد وضعیت HTTP مورد نظر برای پاسخ عملیات مسیر را با استفاده از پارامتر `status_code` مشخص کنید. می‌توانید این مقدار را مستقیماً به صورت یک عدد صحیح، مانند 404 وارد کنید. اما اگر شماره دقیق کدهای وضعیت را به خاطر ندارید، می‌توانید از ثابت‌های میان‌بر موجود در ماژول `status` استفاده کنید:

```
from fastapi import FastAPI, status
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()

@app.post("/items/", response_model=Item, status_code=status.HTTP_201_CREATED)
async def create_item(item: Item):
    return item
```

این کد وضعیت در پاسخ API استفاده شده و همچنین به اسکیمای OpenAPI نیز افزوده خواهد شد.

**جزئیات فنی:** شما همچنین می‌توانید از مسیر `status` از `starlette` استفاده کنید. ماژول `fastapi.status` در واقع همان `starlette.status` است که فقط برای سهولت کار توسعه‌دهنده در FastAPI نیز در دسترس قرار گرفته است. اما منبع اصلی آن Starlette است.

### ۲۵-۲ برچسب‌ها

می‌توانید به عملیات مسیر خود، برچسب (tag) اضافه کنید. برای این کار پارامتر `tags` را همراه با یک لیست از رشته‌ها (اغلب فقط یک رشته) وارد می‌کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel
```

```

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()

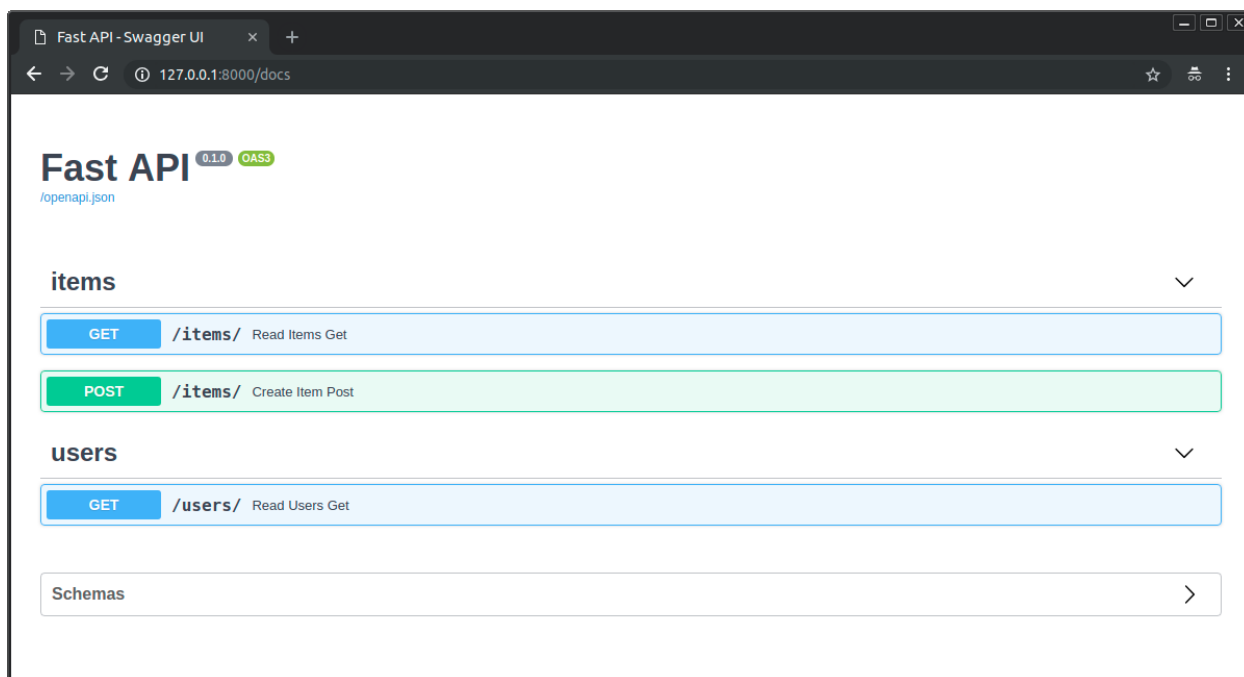
@app.post("/items/", response_model=Item, tags=["items"])
async def create_item(item: Item):
    return item

@app.get("/items/", tags=["items"])
async def read_items():
    return [{"name": "Foo", "price": 42}]

@app.get("/users/", tags=["users"])
async def read_users():
    return [{"username": "johndoe"}]

```

این برچسب‌ها به اسکیمای OpenAPI افزوده شده و در رابط‌های مستندسازی خودکار نمایش داده می‌شوند.



### ۱-۲-۲۵ استفاده از Enums برای برچسب‌ها

اگر برنامه بزرگی دارید، ممکن است در طول زمان تعداد زیادی برچسب ایجاد شود و بخواهید برای عملیات‌های مرتبط، از برچسب‌های یکسان و یکنواخت استفاده کنید. در این مواقع، می‌توانید برچسب‌ها را در قالب یک Enum ذخیره کرده و از آن استفاده کنید. FastAPI این قابلیت را به همان شکلی که برای رشته‌های ساده وجود دارد، پشتیبانی می‌کند:



```

from enum import Enum
from fastapi import FastAPI

app = FastAPI()

class Tags(Enum):
    items = "items"
    users = "users"

@app.get("/items/", tags=[Tags.items])
async def get_items():
    return ["Portal gun", "Plumbus"]

@app.get("/users/", tags=[Tags.users])
async def read_users():
    return ["Rick", "Morty"]

```

### ۳-۲۵ خلاصه و توضیحات

می‌توانید برای عملیات مسیر خود، خلاصه (summary) و توضیح (description) تعریف کنید:

```

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()

@app.post(
    "/items/",
    response_model=Item,
    summary="Create an item",
    description="Create an item with all the information, name, description, price, tax and a set of unique tags",
)
async def create_item(item: Item):
    return item

```

### ۴-۲۵ توضیح از سند رشته

از آنجا که توضیحات معمولاً طولانی هستند و شامل چندین خط می‌شوند، می‌توانید توضیح مسیر را در سند رشته (docstring) تابع تعریف کنید و FastAPI آن را از آنجا استخراج خواهد کرد. شما می‌توانید در سند رشته از نشانه گذاری Markdown استفاده کنید؛ این نشانه گذاری به درستی تفسیر شده و با در نظر گرفتن تورفتگی مناسب نمایش داده می‌شود.

```

from fastapi import FastAPI

```

```
from pydantic import BaseModel

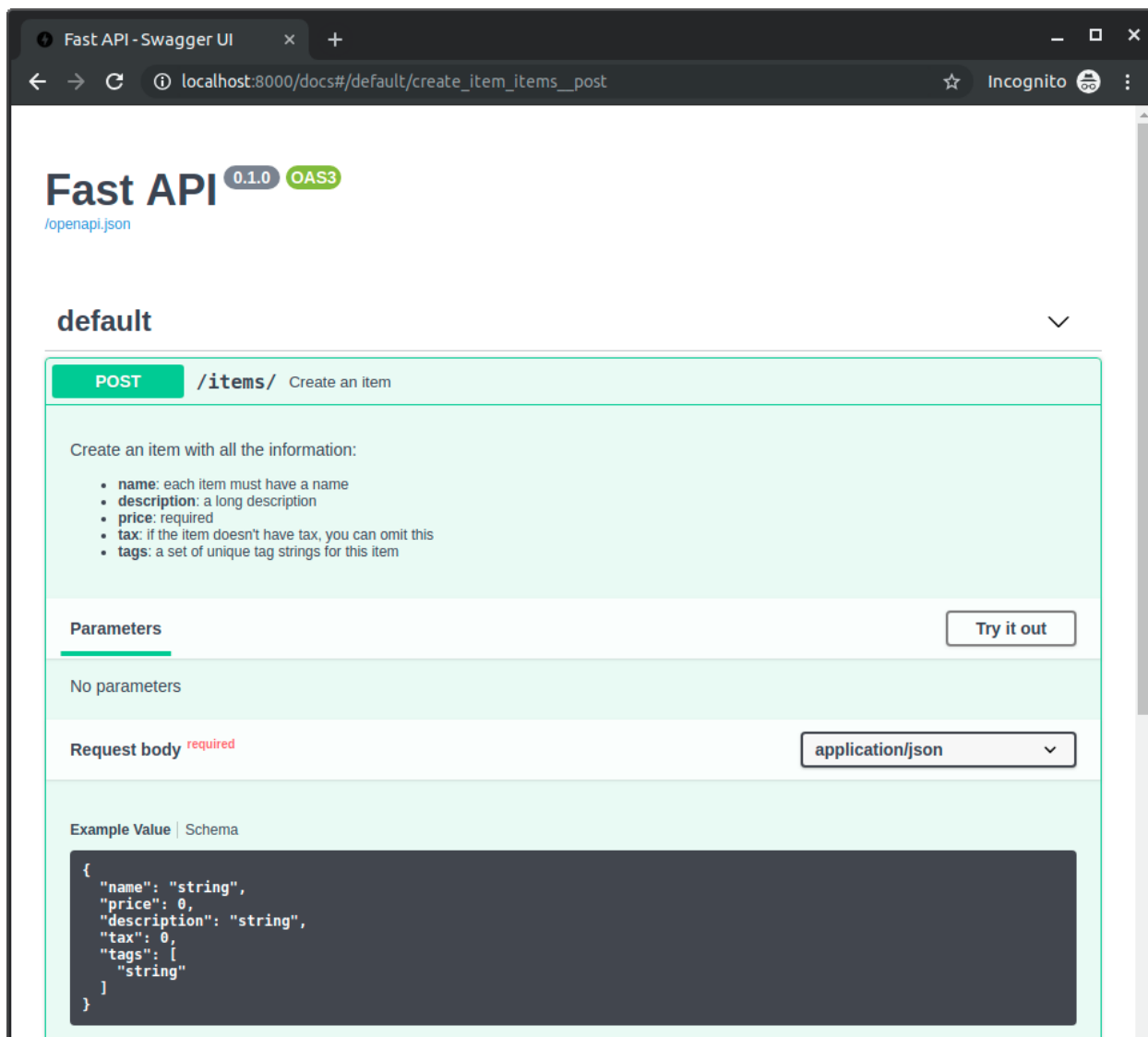
app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()

@app.post("/items/", response_model=Item, summary="Create an item")
async def create_item(item: Item):
    """
    Create an item with all the information:

    - **name**: each item must have a name
    - **description**: a long description
    - **price**: required
    - **tax**: if the item doesn't have tax, you can omit this
    - **tags**: a set of unique tag strings for this item
    """
    return item
```

این اطلاعات در مستندات تعاملی استفاده خواهد شد:



## ۲۵-۵ توضیح پاسخ

شما می توانید توضیح مربوط به پاسخ را با استفاده از پارامتر `response_description` مشخص کنید:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: set[str] = set()

@app.post()
```

```

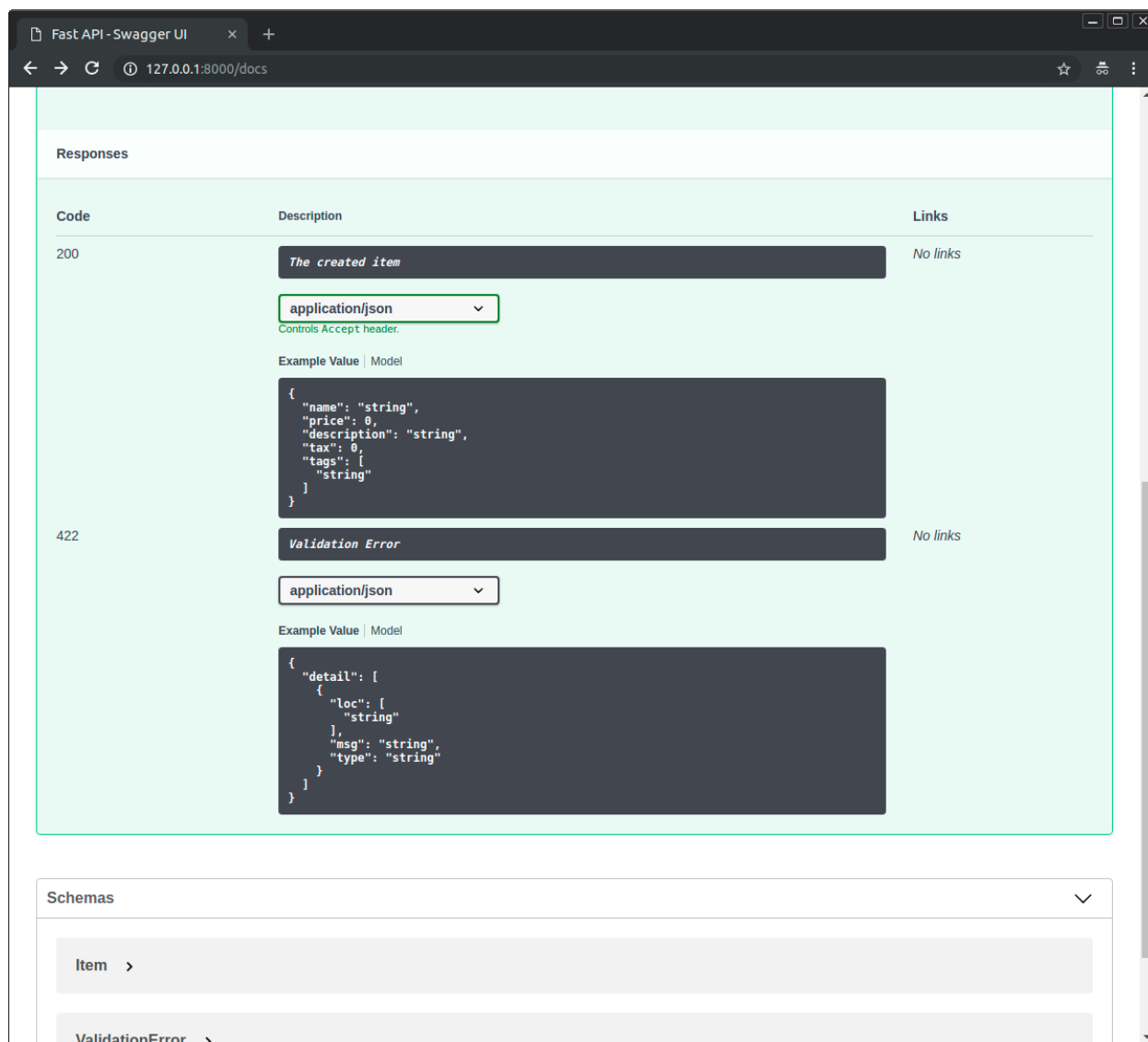
"/items/",
response_model=Item,
summary="Create an item",
response_description= "The created item",
)
async def create_item(item: Item):
    """
    Create an item with all the information:

    - **name**: each item must have a name
    - **description**: a long description
    - **price**: required
    - **tax**: if the item doesn't have tax, you can omit this
    - **tags**: a set of unique tag strings for this item
    """
    return item

```

**اطلاعات:** توجه داشته باشید که `response_description` به طور خاص به توضیح مربوط به پاسخ اشاره دارد در حالی که `description` به مسیر عملیات به صورت کلی مربوط است.

**نکته:** طبق مشخصات OpenAPI، هر مسیر عملیات باید دارای توضیح پاسخ باشد. در صورتی که توضیحی ارائه نکنید، FastAPI به صورت خودکار توضیحی با عنوان `Successful response` ایجاد خواهد کرد.



## ۶-۲۵ غیرفعال سازی مسیر عملیات

اگر لازم باشد یک مسیر عملیاتی را به عنوان منسوخ شده (deprecated) علامت گذاری کنید، بدون اینکه آن را حذف کنید، کافی است پارامتر deprecated را تنظیم نمایید:

```
from fastapi import FastAPI

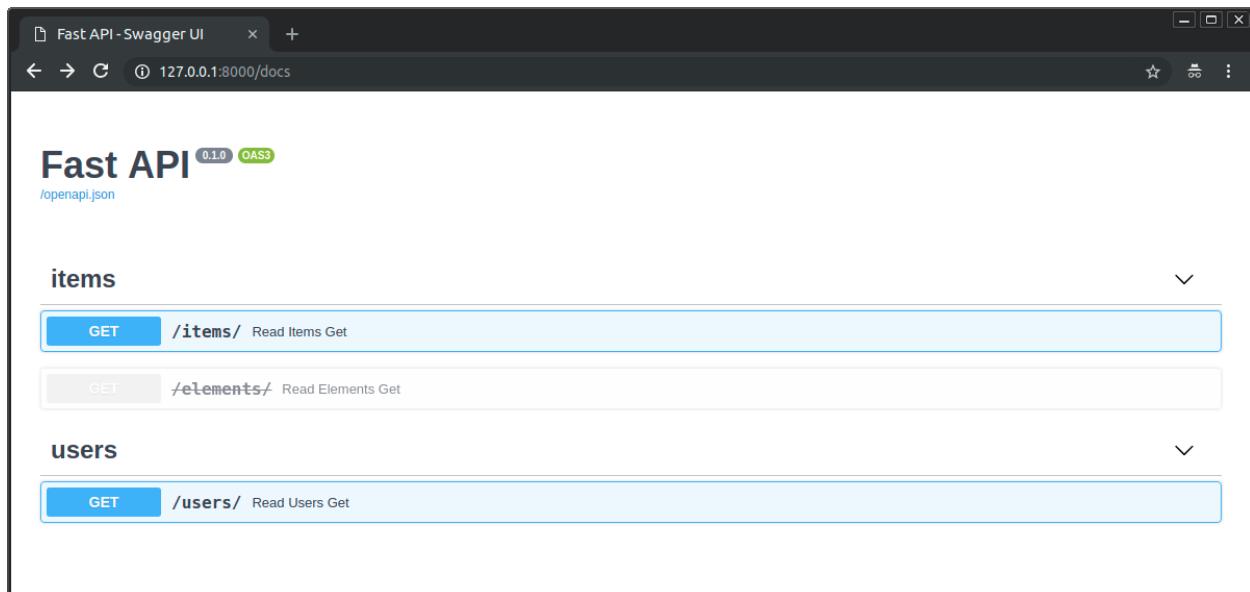
app = FastAPI()

@app.get("/items/", tags=["items"])
async def read_items():
    return [{"name": "Foo", "price": 42}]

@app.get("/users/", tags=["users"])
async def read_users():
    return [{"username": "johndoe"}]
```

```
@app.get("/elements/", tags=["items"], deprecated=True)
async def read_elements():
    return [{"item_id": "Foo"}]
```

در مستندات تعاملی، مسیرهای منسوخ به وضوح مشخص خواهند شد:



بررسی کنید که عملیات مسیر منسوخ شده و منسوخ نشده چگونه به نظر می‌رسند:

The screenshot displays the Swagger UI for a Fast API. The browser address bar shows the URL `127.0.0.1:8000/docs`. The interface lists two API endpoints:

- /items/**: A GET endpoint titled "Read Items Get". It has no parameters and a 200 response status. The response description is "Successful Response". A dropdown menu for "application/json" is visible, with a note "Controls Accept header." below it.
- /elements/**: A GET endpoint titled "Read Elements Get". It is marked as "Warning: Deprecated". It also has no parameters and a 200 response status. The response description is "Successful Response". A dropdown menu for "application/json" is visible, with a note "Controls Accept header." below it.

## جمع‌بندی

شما می‌توانید به راحتی با ارسال پارامترها به دکوراتورهای عملیات مسیر، فراداده را برای عملیات مسیر خود پیکربندی و اضافه کنید.

## فصل ۲۶: کدگذار سازگار با JSON

در برخی موارد، ممکن است نیاز داشته باشید یک نوع داده (مانند یک مدل Pydantic) را به قالبی سازگار با JSON (مانند دیکشنری، لیست و غیره) تبدیل کنید. برای مثال، اگر نیاز باشد آن را در یک پایگاه داده ذخیره کنید. برای این منظور، FastAPI تابع `jsonable_encoder()` را ارائه می‌دهد.

### ۱-۲۶ استفاده از `jsonable_encoder`

فرض کنید یک پایگاه داده به نام `fake_db` دارید که فقط داده‌های سازگار با JSON را می‌پذیرد. مثلاً این پایگاه داده، اشیاء `datetime` را نمی‌پذیرد، زیرا با JSON سازگار نیستند. بنابراین، یک شیء `datetime` باید به یک رشته حاوی داده در فرمت ISO تبدیل شود. به همین ترتیب، این پایگاه داده یک مدل Pydantic (یک شیء دارای ویژگی‌ها) را نمی‌پذیرد و فقط دیکشنری را قبول می‌کند. در اینجا می‌توانید از `jsonable_encoder` استفاده کنید. این تابع یک شیء (مانند یک مدل Pydantic) را دریافت می‌کند و یک نسخه سازگار با JSON از آن را برمی‌گرداند:

```
from datetime import datetime
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

fake_db = {}

class Item(BaseModel):
    title: str
    timestamp: datetime
    description: str | None = None

app = FastAPI()

@app.put("/items/{id}")
def update_item(id: str, item: Item):
    json_compatible_item_data = jsonable_encoder(item)
    fake_db[id] = json_compatible_item_data
```

در این مثال، مدل Pydantic به یک دیکشنری و `datetime` به یک رشته تبدیل می‌شود. نتیجه فراخوانی این تابع، ساختاری است که می‌توان آن را با تابع استاندارد `json.dumps()` پایتون کدگذاری کرد. این تابع یک رشته بزرگ حاوی داده‌ها در قالب JSON برنمی‌گرداند، بلکه یک ساختار داده استاندارد پایتون (مانند دیکشنری) را برمی‌گرداند که تمام مقادیر و زیرمقادیر آن با JSON سازگار هستند.

**نکته:** در واقع، FastAPI به صورت داخلی از `jsonable_encoder` برای تبدیل داده‌ها استفاده می‌کند، اما این تابع در بسیاری از سناریوهای دیگر نیز مفید است.



## فصل ۲۷: بدنه – به‌روزرسانی‌ها

### ۲۷-۱ به‌روزرسانی با استفاده از PUT

برای به‌روزرسانی یک آیتم، می‌توانید از عملیات HTTP نوع PUT استفاده کنید. می‌توانید از تابع `jsonable_encoder` برای تبدیل داده‌های ورودی به فرمی استفاده کنید که قابل ذخیره‌سازی به صورت JSON باشد (مثلاً برای پایگاه‌داده‌های NoSQL). به‌عنوان مثال، تبدیل نوع `datetime` به رشته.

```
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}

@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]

@app.put("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    update_item_encoded = jsonable_encoder(item)
    items[item_id] = update_item_encoded
    return update_item_encoded
```

در عملیات PUT، داده‌ای که دریافت می‌شود باید جایگزین کامل داده قبلی شود.

### ۲۷-۱-۱ هشدار درباره جایگزینی

این یعنی اگر بخواهید آیتم `bar` را با استفاده از PUT و بدنه زیر به‌روزرسانی کنید:

```
{
  "name": "Barz",
  "price": 3,
  "description": null
}
```

```
}
```

چون در این داده‌ی جدید، ویژگی ذخیره‌شده‌ی قبلی یعنی 20.2: "tax" وجود ندارد، مدل ورودی مقدار پیش‌فرض برای tax را که 10.5 است در نظر می‌گیرد و داده جدید با همین مقدار پیش‌فرض 10.5 tax ذخیره خواهد شد.

## ۲۷-۲ به‌روزرسانی جزئی با PATCH

شما همچنین می‌توانید از عملیات HTTP نوع PATCH برای به‌روزرسانی جزئی داده‌ها استفاده کنید. این به این معناست که فقط بخشی از داده‌ها که نیاز به به‌روزرسانی دارند ارسال می‌شوند و سایر فیلدها دست‌نخورده باقی می‌مانند.

**نکته:** PATCH کمتر از PUT استفاده و شناخته می‌شود. بسیاری از تیم‌ها حتی برای به‌روزرسانی جزئی نیز فقط از PUT استفاده می‌کنند. FastAPI محدودیتی در این زمینه اعمال نمی‌کند، بنابراین آزاد هستید از هر کدام که بخواهید استفاده کنید. این راهنما فقط نحوه‌ی معمول استفاده از آن‌ها را نشان می‌دهد.

### ۲۷-۲-۱ استفاده از پارامتر exclude\_unset در Pydantic

اگر بخواهید فقط داده‌هایی را دریافت کنید که برای به‌روزرسانی ارسال شده‌اند (و نه مقادیر پیش‌فرض)، استفاده از پارامتر exclude\_unset در متد model\_dump() مدل Pydantic بسیار مفید است. مثلاً:

```
item.model_dump(exclude_unset=True)
```

**اطلاعات:** در نسخه ۱ کتابخانه‌ی Pydantic این متد dict() نام داشت. در نسخه ۲ به model\_dump() تغییر نام داده ولی همچنان متد dict() پشتیبانی می‌شود. اگر از نسخه ۲ استفاده می‌کنید، بهتر است از model\_dump() استفاده کنید.

این کار دیکشنری‌ای تولید می‌کند که فقط شامل مقادیری است که واقعاً مقداردهی شده‌اند و مقادیر پیش‌فرض حذف می‌شوند. می‌توانید از این قابلیت برای ایجاد یک دیکشنری فقط شامل فیلدهایی که در درخواست ارسال شده‌اند استفاده کنید:

```
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax":
```

```

        20.2},
        "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5,
        "tags": []},
    }

@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]

@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item

```

## ۲-۲-۲ استفاده از پارامتر update در Pydantic

اکنون می‌توانید با استفاده از متد `model_copy()` یک کپی از مدل موجود ایجاد کرده و پارامتر `update` را با یک دیکشنری شامل داده‌هایی که باید به‌روزرسانی شوند، به آن منتقل کنید.

**اطلاعات:** در نسخه ۱ Pydantic، این متد با نام `copy()` شناخته می‌شد. در نسخه ۲ این متد منسوخ (اما هنوز پشتیبانی می‌شود) و به `model_copy()` تغییر نام یافت. مثال‌های زیر از `copy()` برای سازگاری با نسخه ۱ استفاده می‌کنند، اما اگر از Pydantic نسخه ۲ بهره می‌برید، توصیه می‌شود از `model_copy()` استفاده کنید.

مثال استفاده:

```
stored_item_model.model_copy(update=update_data)
```

مثال کامل:

```

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax":

```

```

    20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5,
            "tags": []},
}

@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]

@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item

```

### ۳-۲-۲۷ مرور کلی به روزرسانی جزئی

به طور خلاصه، برای اعمال به روزرسانی‌های جزئی، مراحل زیر را انجام دهید:

- به جای PUT از متد PATCH استفاده کنید. (اختیاری)
- داده‌های ذخیره شده را بازیابی کنید.
- آن داده‌ها را در قالب یک مدل Pydantic قرار دهید.
- از مدل ورودی، دیکشنری‌ای بدون مقادیر پیش فرض تولید کنید (با استفاده از `exclude_unset=True`).
- به این شکل، فقط مقادیری که کاربر به صورت صریح وارد کرده، به روزرسانی می‌شوند و مقادیر پیش فرض روی مقادیر ذخیره شده بازنویسی نمی‌شوند.
- یک کپی از مدل ذخیره شده ایجاد کرده و ویژگی‌های آن را با استفاده از پارامتر `update` به روزرسانی کنید.
- مدل کپی شده را به فرمتی تبدیل کنید که قابل ذخیره در پایگاه داده باشد (مثلاً با استفاده از `jsonable_encoder`).
- این کار مشابه استفاده مجدد از `model_dump()` است، اما تضمین می‌کند که مقادیر به فرمت‌های قابل JSON (مانند تبدیل `datetime` به `str`) تبدیل شده‌اند.
- داده‌ها را در پایگاه داده ذخیره کنید.
- مدل به روزرسانی شده را بازگردانید.

```

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

```

```

name: str | None = None
description: str | None = None
price: float | None = None
tax: float = 10.5
tags: list[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax":
        20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5,
        "tags": []},
}

@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]

@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item

```

**نکته:** در واقع می‌توانید از همین تکنیک در عملیات HTTP از نوع PUT نیز استفاده کنید. اما این مثال از PATCH استفاده کرده زیرا مخصوص چنین مواردی طراحی شده است.

**اطلاعات:** ورودی همچنان به‌طور کامل اعتبارسنجی می‌شود. بنابراین اگر می‌خواهید امکان ارسال به‌روزرسانی جزئی بدون نیاز به پر کردن هیچ ویژگی‌ای را فراهم کنید، باید مدلی تعریف کنید که همه ویژگی‌های آن اختیاری (دارای مقدار پیش‌فرض یا None) باشند. برای تفکیک میان مدل‌هایی با مقادیر اختیاری جهت به‌روزرسانی و مدل‌هایی با مقادیر اجباری جهت ایجاد (Create)، می‌توانید از ایده‌های مطرح‌شده در فصل ۱۸ (مدل‌های اضافی) استفاده کنید.

## فصل ۲۸: کلاس‌ها به عنوان وابستگی‌ها

پیش از آن‌که به عمق سیستم تزریق وابستگی (Dependency Injection) در FastAPI برویم، اجازه دهید مثال قبلی را ارتقاء دهیم.

### ۲۸-۱ دیکشنری در مثال قبلی

در مثال قبلی، ما از یک تابع وابسته، مقدار بازگشتی از نوع دیکشنری داشتیم:

```
from typing import Annotated
from fastapi import Depends, FastAPI

app = FastAPI()

async def common_parameters(q: str | None = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
async def read_items(common: Annotated[dict, Depends(common_parameters)]):
    return common

@app.get("/users/")
async def read_users(common: Annotated[dict, Depends(common_parameters)]):
    return common
```

اما در این حالت، مقدار `commons` فقط یک دیکشنری است، و می‌دانیم که ویرایشگرهای کد معمولاً نمی‌توانند پشتیبانی مناسبی مانند تکمیل خودکار برای کلیدها و مقادیر در دیکشنری ارائه دهند، چرا که نوع داده‌ها مشخص نیست. می‌توانیم این کار را بهتر انجام دهیم...

### ۲۸-۲ وابستگی چیست؟

تا اینجا، وابستگی‌ها را به شکل تابع تعریف کرده‌ایم. اما این تنها روش تعریف وابستگی نیست (گرچه رایج‌ترین روش است). نکته کلیدی این است که وابستگی باید قابل فراخوانی (callable) باشد. در پایتون، هر چیزی که بتوان آن را مانند یک تابع «فراخوانی» کرد، قابل فراخوانی محسوب می‌شود. مثلاً اگر یک شیء به نام `something` دارید که به صورت زیر اجرا می‌شود:

```
something()
```

یا

```
something(some_argument, some_keyword_argument="foo")
```

در این صورت، آن یک شیء قابل فراخوانی است.

### ۲۸-۳ کلاس‌ها به عنوان وابستگی

شاید متوجه شده باشید که ساخت یک نمونه (instance) از یک کلاس در پایتون نیز دقیقاً با همین سینتکس انجام می‌شود:

```
class Cat:
    def __init__(self, name: str):
        self.name = name

fluffy = Cat(name="Mr Fluffy")
```

در اینجا fluffy یک نمونه از کلاس Cat است. برای ساخت fluffy، ما کلاس Cat را **فراخوانی** کرده‌ایم. بنابراین، کلاس‌ها نیز در پایتون یک شی قابل فراخوانی هستند. به همین دلیل، در FastAPI نیز می‌توان از کلاس به عنوان یک وابستگی استفاده کرد. FastAPI در واقع فقط بررسی می‌کند که آیا شی موردنظر قابل فراخوانی هست یا نه (چه تابع، چه کلاس، یا هر چیز دیگر) و سپس پارامترهای آن را تحلیل می‌کند.

اگر یک شی قابل فراخوانی را به عنوان وابستگی در FastAPI پاس دهید، پارامترهای آن مانند پارامترهای یک تابع عملیات مسیر بررسی و پردازش می‌شوند، حتی اگر آن پارامترها خود دارای وابستگی‌های دیگری باشند. این موضوع برای هر شی قابل فراخوانی که هیچ پارامتری ندارند نیز صدق می‌کند — درست مانند توابع عملیات مسیر بدون پارامتر.

اکنون می‌توانیم وابستگی common\_parameters را که قبلاً یک تابع بود، به یک **کلاس** به نام CommonQueryParams تبدیل کنیم:

```
from typing import Annotated
from fastapi import Depends, FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]

class CommonQueryParams:
    def __init__(self, q: str | None = None, skip: int = 0, limit: int = 100):
        self.q = q
        self.skip = skip
        self.limit = limit

@app.get("/items/")
async def read_items(
    commons: Annotated[CommonQueryParams, Depends(CommonQueryParams)]:
    response = {}
    if commons.q:
        response.update({"q": commons.q})
    items = fake_items_db[commons.skip : commons.skip + commons.limit]
    response.update({"items": items})
    return response
```

به متد سازنده کلاس \_\_init\_\_ که برای ایجاد نمونه از کلاس استفاده شده است توجه کنید:

```
def __init__(self, q: str | None = None, skip: int = 0, limit: int = 100):
    self.q = q
    self.skip = skip
    self.limit = limit
```

این متد پارامترهای مشابهی با تابع `common_parameters` قبلی ما دارد:

```
from typing import Annotated
from fastapi import Depends, FastAPI

app = FastAPI()

async def common_parameters(q: str | None = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
async def read_items(common: Annotated[dict, Depends(common_parameters)]):
    return common

@app.get("/users/")
async def read_users(common: Annotated[dict, Depends(common_parameters)]):
    return common
```

FastAPI از این پارامترها برای تحلیل و حل وابستگی استفاده می‌کند. در هر دو حالت (چه تابع، چه کلاس)، این پارامترها را دارد:

- `q`: یک پارامتر اختیاری از نوع `str`
- `skip`: یک پارامتر از نوع `int` با مقدار پیش‌فرض صفر
- `limit`: یک پارامتر از نوع `int` با مقدار پیش‌فرض ۱۰۰

در هر دو حالت، داده‌ها تبدیل، اعتبارسنجی، مستندسازی در اسکیمای `OpenAPI` و غیره خواهند شد.

## ۴-۲۸ استفاده از آن

اکنون می‌توانید وابستگی خود را با استفاده از این کلاس اعلام کنید:

```
from typing import Annotated
from fastapi import Depends, FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]

class CommonQueryParams:
    def __init__(self, q: str | None = None, skip: int = 0, limit: int = 100):
        self.q = q
        self.skip = skip
```



```

        self.limit = limit

@app.get("/items/")
async def read_items(
    commons: Annotated[CommonQueryParams, Depends(CommonQueryParams)]):
    response = {}
    if commons.q:
        response.update({"q": commons.q})
    items = fake_items_db[commons.skip : commons.skip + commons.limit]
    response.update({"items": items})
    return response

```

FastAPI کلاس `CommonQueryParams` را فراخوانی می‌کند. این کار یک "نمونه" از آن کلاس می‌سازد و نمونه ساخته‌شده به عنوان پارامتر `commons` به تابع شما داده می‌شود.

## ۵-۲۸ حاشیه‌نویسی نوع در برابر وابستگی

به نحوه نوشتن `CommonQueryParams` در کد بالا دقت کنید:

```
commons: Annotated[CommonQueryParams, Depends(CommonQueryParams)]
```

آن `CommonQueryParams` آخر در بخش:

```
... Depends(CommonQueryParams)
```

در واقع همان چیزی است که FastAPI از آن برای تشخیص وابستگی استفاده می‌کند. FastAPI پارامترهای اعلام‌شده را از همین مورد استخراج می‌کند و همین را نیز فراخوانی می‌کند. در این حالت، `CommonQueryParams` اول در عبارت:

```
commons: Annotated[CommonQueryParams, ...
```

هیچ نقش خاصی برای FastAPI ندارد. FastAPI از آن برای تبدیل داده، اعتبارسنجی و ... استفاده نمی‌کند (بلکه از `Depends(CommonQueryParams)` برای این موارد استفاده می‌کند). در واقع می‌توانستید به سادگی بنویسید:

```
commons: Annotated[Any, Depends(CommonQueryParams)]
```

مانند:

```

from typing import Annotated, Any
from fastapi import Depends, FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]

class CommonQueryParams:
    def __init__(self, q: str | None = None, skip: int = 0, limit: int = 100):

```

```

        self.q = q
        self.skip = skip
        self.limit = limit

@app.get("/items/")
async def read_items(common: Annotated[Any, Depends(CommonQueryParams)]):
    response = {}
    if common.q:
        response.update({"q": common.q})
    items = fake_items_db[common.skip : common.skip + common.limit]
    response.update({"items": items})
    return response

```

اما مشخص کردن نوع توصیه می‌شود، زیرا در این صورت ویرایشگر کد شما می‌داند که چه چیزی به پارامتر `common` داده خواهد شد و می‌تواند در تکمیل خودکار کد، بررسی نوع‌ها و غیره به شما کمک کند.

```

1  from fastapi import Depends, FastAPI
2
3  app = FastAPI()
4
5
6  fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]
7
8
9  class CommonQueryParams:
10     def __init__(self, q: str = None, skip: int = 0, limit: int = 100):
11         self.q = q
12         self.skip = skip
13         self.limit = limit
14
15
16 @app.get("/items/")
17 async def read_items(common: CommonQueryParams = Depends(CommonQueryParams)):
18     response = {}
19     if common.q:
20         response.update({"q": common.q})
21     items = fake_items_db[common.skip : common.limit]
22     response.update({"items": items})
23     return response
24

```

## ۶-۲۸ میان‌بر

اما همان‌طور که می‌بینید، در اینجا تکرار کدی وجود دارد، چون `CommonQueryParams` را دوبار می‌نویسیم:

```
common: Annotated[CommonQueryParams, Depends(CommonQueryParams)]
```

`FastAPI` برای چنین مواردی یک میان‌بر فراهم کرده است؛ در مواردی که وابستگی مشخصاً یک کلاس است و `FastAPI` می‌تواند خودش یک نمونه از آن کلاس ایجاد کند. برای چنین موارد خاصی می‌توانید به جای نوشتن:

```
common: Annotated[CommonQueryParams, Depends(CommonQueryParams)]
```

بنویسید:

```
commons: Annotated[CommonQueryParams, Depends()]
```

شما وابستگی را به صورت نوع پارامتر اعلام می‌کنید و به جای نوشتن مجدد کلاس در `Depends(CommonQueryParams)`، فقط از `Depends()` استفاده می‌کنید. مثال بالا به این صورت درمی‌آید:

```
from typing import Annotated
from fastapi import Depends, FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]

class CommonQueryParams:
    def __init__(self, q: str | None = None, skip: int = 0, limit: int = 100):
        self.q = q
        self.skip = skip
        self.limit = limit

@app.get("/items/")
async def read_items(common: Annotated[CommonQueryParams, Depends()]):
    response = {}
    if common.q:
        response.update({"q": common.q})
    items = fake_items_db[common.skip : common.skip + common.limit]
    response.update({"items": items})
    return response
```

و FastAPI می‌داند که باید چه کار کند.

**نکته:** اگر این روش میان‌بر برایتان بیشتر گیج‌کننده است تا مفید، نگران نباشید، نیازی به استفاده از آن نیست. این فقط یک میان‌بر است. زیرا FastAPI اهمیت زیادی برای کاهش تکرار کد قائل است.