

Blazing-Fast APIs with FastAPI

Hamburg, 28.09.2022



APIGENIO
MAKE IT SIMPLE

Hello!

I am David Pinezich

Founder & CEO of apigenio GmbH

Software Architect & Lecturer

You can find me at @dpinezich



Table of contents

#1

Intro

#2

Preparation

#3

Python-CC

#4

Getting started

#5

Security & Benchmarks

#6

Project

Schedule

14:00 - 15:30 Part 1 - Introduction & Getting started

15:30 - 16:00 Break (with food & beverages outside)

16:00 - 17:30 Part 2 - More topics & Project

17:30 - ca. 18:00 Q&A

1.

Intro

An introduction
to FastAPI and
its story



FastAPI wouldn't exist if not for the previous work of others.

There have been many tools created before that have helped inspire its creation.

I have been avoiding the creation of a new framework for several years. First, I tried to solve all the features covered by **FastAPI** using many different frameworks, plug-ins, and tools.

But at some point, there was no other option than creating something that provided all these features, taking the best ideas from previous tools and combining them in the best way possible, using language features that weren't even available before (Python 3.6+ type hints).

Sebastián Ramírez

Alternatives

Let us talk about some of the alternatives and how they influenced FastAPI.

Alternatives

- Django
 - It's the most popular Python framework and is widely trusted. It is used to build systems like Instagram.
- Django REST Framework
 - Django REST framework was created to be a flexible toolkit for building Web APIs using Django underneath, to improve its API capabilities.
 - Inspiration:
 - Have an automatic API documentation web user interface.

Alternatives

- Flask
 - Inspiration:
 - Have a simple and easy to use routing system.
 - Be a micro-framework. Making it easy to mix and match the tools and parts needed.

Alternatives

- Requests
 - **FastAPI** is not actually an alternative to **Requests**. Their scope is very different. It would actually be common to use Requests *inside* of a FastAPI application.
 - Inspiration:
 - Have sensible defaults, but powerful customizations
 - Use HTTP method names (operations) directly, in a straightforward and intuitive way.
 - Have a simple and intuitive API.

Alternatives

- Swagger / OpenAPI
 - The main feature I wanted from Django REST Framework was the automatic API documentation.
 - Inspiration:
 - Adopt and use an open standard for API specifications, instead of a custom schema. And integrate standards-based user interface tools: Swagger UI and ReDoc

Alternatives

- Marshmallow
 - One of the main features needed by API systems is data "serialization" which is taking data from the code (Python) and converting it into something that can be sent through the network
 - Inspiration:
 - Use code to define "schemas" that provide data types and validation, automatically.

Alternatives

- Webargs
 - Another big feature required by APIs is parsing data from incoming requests.
 - Inspiration:
 - Have automatic validation of incoming request data.

Alternatives

- APISpec
 - Marshmallow and Webargs provide validation, parsing and serialization as plug-ins.
But documentation is still missing. Then APISpec was created.
 - Inspiration:
 - Support the open standard for APIs, OpenAPI.
- Flask-apispec
 - It's a Flask plug-in, that ties together Webargs, Marshmallow and APISpec.
 - Inspiration:
 - Generate the OpenAPI schema automatically, from the same code that defines serialization and validation.

Alternatives

- NestJS and Angular
 - Inspiration:
 - Use Python types to have great editor support.
 - Have a powerful dependency injection system. Find a way to minimize code repetition.
- Sanic
 - It used uvloop instead of the default Python asyncio loop. That's what made it so fast.
 - Inspiration:
 - That's why **FastAPI** is based on Starlette, as it is the fastest framework available (tested by third-party benchmarks).

Alternatives

- Falcon
 - Falcon is another high performance Python framework, it is designed to be minimal, and work as the foundation of other frameworks like Hug
 - Inspiration:
 - Find ways to get great performance. Along with Hug (as Hug is based on Falcon) inspired FastAPI to declare a response parameter in functions.
 - Although in FastAPI it's optional, and is used mainly to set headers, cookies, and alternative status codes.

Helping Giants

It is said that FastAPI has the help of Giants.
Namely Pydantic, Starlette and Uvicorn!



Pydantic

- Pydantic is a library to define data validation, serialization and documentation (using JSON Schema) based on Python type hints.
- Pydantic is very intuitive.
- Inspiration / Help:
 - Handle all the **data validation, data serialization and automatic model documentation (based on JSON Schema)**.
 - FastAPI then takes that JSON Schema data and puts it in OpenAPI, apart from all the other things it does.

Starlette

- Starlette is a lightweight ASGI framework/toolkit, which is ideal for building high-performance asyncio services.
- It is very simple and intuitive.
- It's designed to be easily extensible, and have modular components.
- Inspiration / Help:
 - Handle all the core web parts and adding features on top.
 - The class FastAPI itself inherits directly from the class Starlette.
 - So, anything that you can do with Starlette, you can do it directly with FastAPI, as it is basically Starlette on steroids.

Uvicorn

- Uvicorn is a lightning-fast ASGI server, built on uvloop and http tools. It is not a web framework, but a server.
- Inspiration / Help:
 - The main web server to run FastAPI applications.
 - You can combine it with Gunicorn, to have an asynchronous multi-process server.

2. Preparation

Get your Dev-
Environment up
to speed

Dev-Tooling

It is important to have your Dev-Tooling ready for this FastAPI Session. There is not much to install but some tools are beneficial:

- A state of the art (Chromium) Browser.
- An IDE which is capable to work with PIP and VENV (ex. PyCharm or VScode).
- Postman (to test the API).
- Optional: Git.

Sources

You will get all the sources, hosted on Github. You can check them out or download a zip-file.

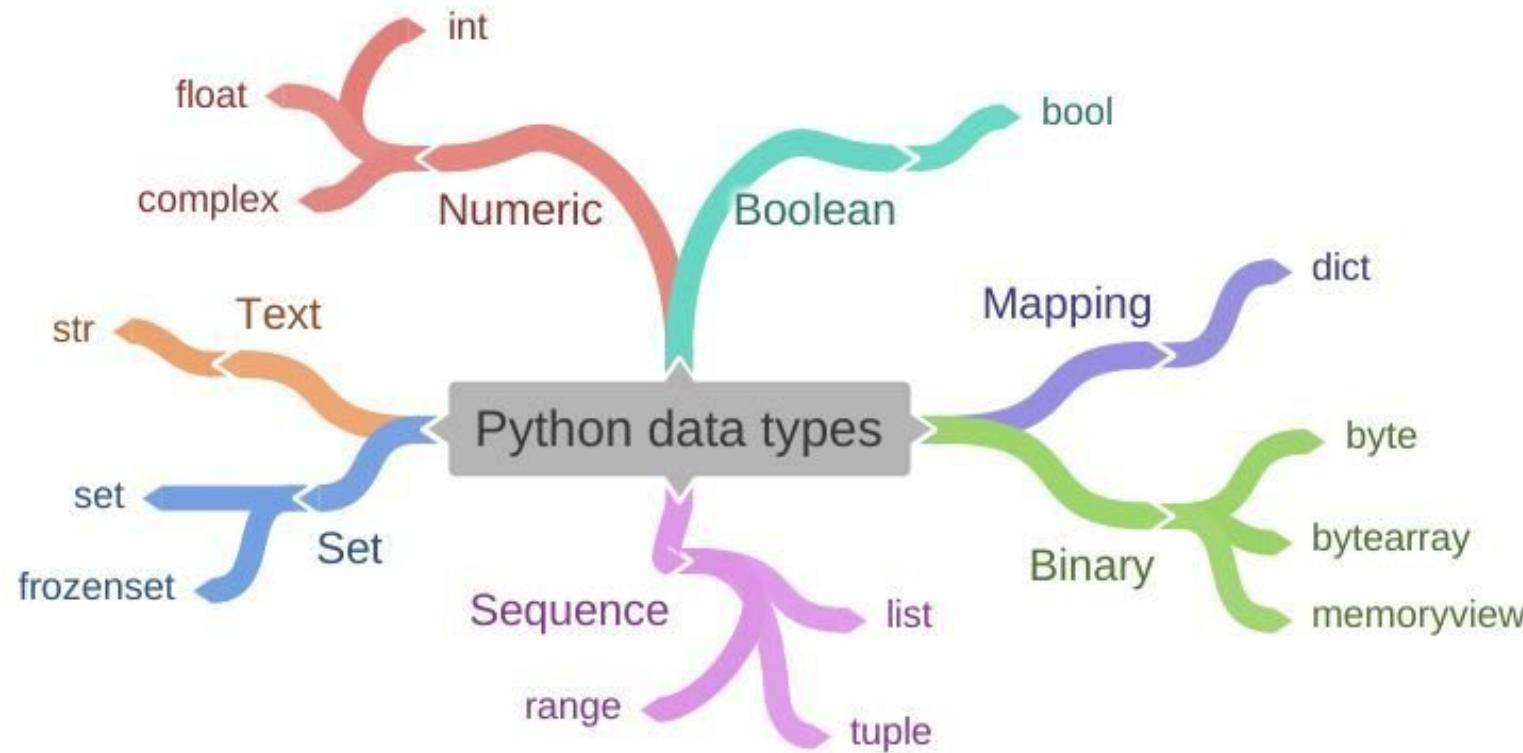
- Git: `git@github.com:apigenio-GmbH/blazing-fast-fastapi.git`
 - <https://bit.ly/3DWDLiR>
- Zip: <https://github.com/apigenio-GmbH/blazing-fast-fastapi/archive/refs/heads/main.zip>
 - <https://bit.ly/3fsTt15>

Python-CrashCourse

3.

Let us talk some
Python first
before working
with FastAPI

Basic Data Types in Python



Numeric



```
1 # 00_crashcourse/numeric.py
2
3 a = 5
4 print("Type of a: ", type(a))
5
6 b = 5.0
7 print("\nType of b: ", type(b))
8
9 c = 2 + 4j
10 print("\nType of c: ", type(c))
```

Text (Strings or Char-Seq)



```
1 # 00_crashcourse/strings.py
2
3 string1 = 'Welcome to the World'
4 print("String with the use of Single Quotes: ")
5 print(string1)
6
7 string2 = "I'm a Hero"
8 print("\nString with the use of Double Quotes: ")
9 print(string2)
10
11 string3 = '''I'm a Hero and I live in a world of "Heros"'''
12 print("\nString with the use of Triple Quotes: ")
13 print(string3)
14
15 string4 = '''Hero
16             For
17             Life'''
18 print("\nCreating a multiline String: ")
19 print(string4)
```

List



```
1 # 00_crashcourse/lists.py
2
3 list = []
4 print("Initial blank List: ")
5 print(list)
6
7 list2 = ['HeroesAreEverywhere']
8 print("\nList with the use of String: ")
9 print(list2)
10
11 list3 = ["Hero", "For", "Heroes"]
12 print("\nList containing multiple values: ")
13 print(list3[0])
14 print(list3[2])
15
16 list4 = [[ 'Hero' , 'For' ], [ 'Heroes' ] ]
17 print("\nMulti-Dimensional List: ")
18 print(list4)
```

Tuple



```
1 # 00_crashcourse/tuples.py
2
3 tuple1 = ()
4 print("Initial empty Tuple: ")
5 print(tuple1)
6
7 tuple2 = ('Heroes', 'For')
8 print("\nTuple with the use of String: ")
9 print(tuple2)
10
11 list1 = [1, 2, 4, 5, 6]
12 print("\nTuple using List: ")
13 print(tuple(list1))
14
15 tuple3 = tuple('Heroes')
16 print("\nTuple with the use of function: ")
17 print(tuple3)
18
19 tuple4 = (0, 1, 2, 3)
20 tuple5 = ('python', 'hero')
21 tuple6 = (tuple4, tuple5)
22 print("\nTuple with nested tuples: ")
23 print(tuple6)
```

Question

What is the difference between a list and a tuple?

Lists are mutable - tuples are immutable.

Set



```
1 # 00_crashcourse/sets.py
2
3 set1 = set()
4 print("Initial blank Set: ")
5 print(set1)
6
7 set2 = set("HeroForHeroes")
8 print("\nSet with the use of String: ")
9 print(set2)
10
11 set3 = set(["Hero", "For", "Heroes"])
12 print("\nSet with the use of List: ")
13 print(set3)
14
15 set4 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
16 print("\nSet with the use of Mixed Values")
17 print(set4)
```

Question

What is unique about a Set?

Every element can only exist once.

Frozen-Set

Freeze the list, and make it unchangeable:

```
● ● ●  
1 # 00_crashcourse/frozensets.py  
2  
3 mylist = ['apple', 'banana', 'cherry']  
4 x = frozenset(mylist)  
5 x[1] = "strawberry" #Error!  
6
```

Dictionaries



```
1 # 00_crashcourse/dictionaries.py
2
3 dict1 = {}
4 print("Empty Dictionary: ")
5 print(dict1)
6
7 dict2 = {1: 'Hero', 2: 'For', 3: 'Heroes'}
8 print("\nDictionary with the use of Integer Keys: ")
9 print(dict2)
10
11 dict3 = {'Name': 'Hero', 1: [1, 2, 3, 4]}
12 print("\nDictionary with the use of Mixed Keys: ")
13 print(dict3)
14
15 dict4 = dict({1: 'Hero', 2: 'For', 3: 'Heroes'})
16 print("\nDictionary with the use of dict(): ")
17 print(dict4)
18
19 dict5 = dict([(1, 'Heroes'), (2, 'For')])
20 print("\nDictionary with each item as a pair: ")
21 print(dict5)
```

Typing in Python

Python has support for optional "type hints" (3.5+).

These "type hints" are a special syntax that allows declaring the type of a variable.

By declaring types for your variables, editors and tools can give you better support.

Motivation for Types



```
1 def get_full_name(first_name, last_name):
2     full_name = first_name.title() + " " + last_name.title()
3     return full_name
4
5
6 print(get_full_name("john", "doe"))
```

What happens?

- Takes a `first_name` and `last_name`.
- Converts the first letter of each one to upper case with `title()`.
- Concatenates them with a space in the middle.

Motivation for Types

```
● ● ●  
1 def get_full_name(first_name, last_name):  
2     full_name = first_name.title() + " " + last_name.title()  
3     return full_name  
4  
5  
6 print(get_full_name("john", "doe"))
```

It is a very easy program, but after some time you ask yourself:

Was it upper? Was it uppercase? first_uppercase? capitalize?

Motivation for Types

Also, sadly no magic from the IDE:

A screenshot of a code editor showing a completion dropdown. The code being typed is:

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

The cursor is at the end of "first_name.". A completion dropdown is open, listing:

- def
- first_name
- full_name
- get_full_name
- last_name

The background of the code editor is dark, and the completion dropdown has a light gray background with a blue header bar.

Motivation for Typing

Let us add some typing:

```
● ● ●  
1 def get_full_name(first_name: str, last_name: str):  
2     full_name = first_name.title() + " " + last_name.title()  
3     return full_name  
4  
5  
6 print(get_full_name("john", "doe"))
```

Small sidenote: Your IDE will be much better with Syntax-highlighting.

Motivation for Typing

Careful, it is not the same as:



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

Instead of an **expected type**, this adds a **default value** if the parameter is not set by calling the function.

Motivation for Types

Now, the IDE magic has returned:

A screenshot of a Python IDE showing code completion for the `str.format` method. The code being typed is:

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

The cursor is at the end of `first_name.`. A code completion dropdown menu is open, listing several methods starting with `*`, including `format`. The `format` option is highlighted. To the right of the dropdown, a tooltip provides documentation for `format`:

`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 '}').

Motivation for Types

Now, the IDE magic has returned - title appears!

The screenshot shows a code editor with the following code:

```
1 def get_full_name(first_name: str, last_name: str):
2     full_name = first_name + " " + last_name
```

A tooltip is displayed over the line `full_name = first_name + " " + last_name` at the cursor position. The tooltip shows the signature `str.title(self)` and the documentation: "Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case." The word `title` in the tooltip is highlighted with a blue selection bar.

Even more motivation for Typing

Again some type hints are given:

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

Can you spot the error?

Even more motivation for Typing

Let us fix the error:

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

Even more motivation for Typing

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

Declaring Types

You can declare all Python standard types:

`int, float, bool, bytes`

... and also structures that can contain other values, like

`dict, list, set and tuple`

... but that is not all, newer versions of python are making it more easy :-)

Declaring Types (List)

Python 3.6+

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

Python 3.9+

```
● ● ●  
1 def process_items(items: list[str]):  
2     for item in items:  
3         print(item)
```

Declaring Types (Tuple and Set)

Python 3.6+

```
● ● ●  
1 from typing import Set, Tuple  
2  
3  
4 def process_items(items_t: Tuple[int, int, str], items_s: Set[bytes]):  
5     return items_t, items_s
```

Python 3.9+

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

Declaring Types (Dict)

Python 3.6+

```
● ● ●  
1 from typing import Dict  
2 def process_items(prices: Dict[str, float]):  
3     for item_name, item_price in prices.items():  
4         print(item_name)  
5         print(item_price)
```

Python 3.9+

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

Declaring Types (Union)

Python 3.6+

```
● ● ●  
1 from typing import Union  
2  
3 def process_item(item: Union[int, str]):  
4     print(item)
```

Python 3.10+

```
● ● ●  
1 def process_item(item: int | str):  
2     print(item)
```

Declaring Types (Optional)

Python 3.10+

● ● ●

```
1 def say_hi(name: str | None = None):  
2     if name is not None:  
3         print(f"Hey {name}!")  
4     else:  
5         print("Hello World")
```

Declaring Types (Optional)

Python 3.10+

```
● ● ●  
1 from typing import Optional  
2  
3  
4 def say_hi(name: Optional[str] = None):  
5     if name is not None:  
6         print(f"Hey {name}!")  
7     else:  
8         print("Hello World")
```

Pydantic models

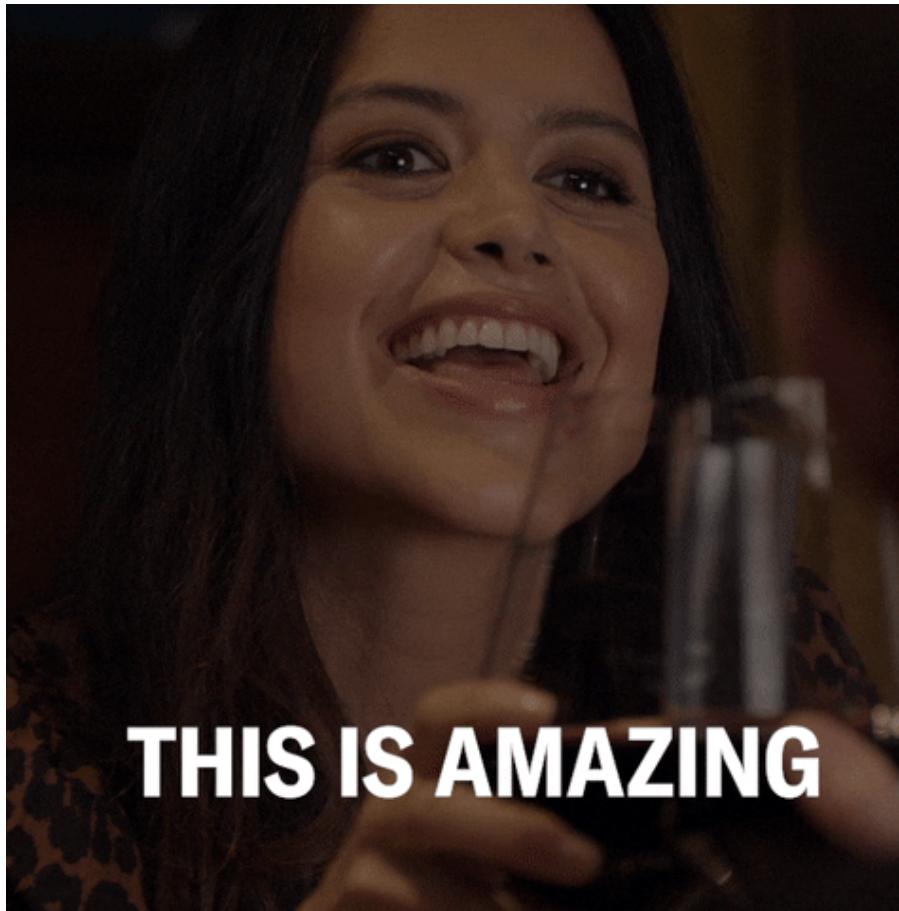
You declare the "shape" of the data as classes with attributes.

And each attribute has a type.

Then you create an instance of that class with some values, and it will validate the values, convert them to the appropriate type (if that's the case), and give you an object with all the data.

And you get all the editor support with that resulting object.

Pydantic models



Pydantic models

Python 3.6+



```
1 from datetime import datetime
2 from typing import List, Union
3 from pydantic import BaseModel
4
5 class User(BaseModel):
6     id: int
7     name = "John Doe"
8     signup_ts: Union[datetime, None] = None
9     friends: List[int] = []
10
11
12 external_data = {
13     "id": "123",
14     "signup_ts": "2017-06-01 12:22",
15     "friends": [1, 2, b"3"],
16 }
17 user = User(**external_data)
18 print(user)
19 # > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
20 print(user.id)
21 # > 123
```

Pydantic models

Python 3.9+



```
1 from datetime import datetime
2 from typing import Union
3 from pydantic import BaseModel
4
5 class User(BaseModel):
6     id: int
7     name = "John Doe"
8     signup_ts: Union[datetime, None] = None
9     friends: list[int] = []
10
11
12 external_data = {
13     "id": "123",
14     "signup_ts": "2017-06-01 12:22",
15     "friends": [1, 2, b"3"],
16 }
17 user = User(**external_data)
18 print(user)
19 # > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
20 print(user.id)
21 # > 123
```

Pydantic models

Python 3.10+



```
1 from datetime import datetime
2 from pydantic import BaseModel
3
4 class User(BaseModel):
5     id: int
6     name = "John Doe"
7     signup_ts: datetime | None = None
8     friends: list[int] = []
9
10
11 external_data = {
12     "id": "123",
13     "signup_ts": "2017-06-01 12:22",
14     "friends": [1, "2", b"3"],
15 }
16 user = User(**external_data)
17 print(user)
18 # > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
19 print(user.id)
```

Short intermezzo args* and kwargs**

Python has args* which allows us to pass the variable number an amount of "numbers" which is not defined before.

```
● ● ●  
1 # 00_crashcourse/args.py  
2  
3 def adder(*num):  
4     sum = 0  
5     for n in num:  
6         sum = sum + n  
7     print("Sum:", sum)  
8  
9 adder(3, 5)  
10 adder(4, 5, 6, 7)  
11 adder(1, 2, 3, 5, 6)
```

Short intermezzo args* and kwargs**

Python has kwargs** that allow us to pass the variable data as many key/value pairs as needed to the function.

```
● ● ●  
1 # 00_crashcourse/kwargs.py  
2  
3 def intro(**data):  
4     print("\nData type of argument:",type(data))  
5  
6     for key, value in data.items():  
7         print("{} is {}".format(key,value))  
8  
9 intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)  
10 intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com",  
11       Country="Wakanda", Age=25, Phone=9876543210)
```

Short intermezzo args* and kwargs**

So what is the difference between args* and kwargs**?

Args are only values (equal to a list) kwargs are key-value pairs (equal to a dictionary).

Short intermezzo args* and kwargs**

*args and **kwargs are particular keywords that allow the function to take variable length argument.

*args pass a variable number of non-keyworded arguments and on which operation of the tuple can be performed.

**kwargs pass a variable number of keyword arguments dictionary to function on which dictionary operation can be performed.

*args and **kwargs make the function flexible.

Type Hints in FastAPI

FastAPI takes advantage of these type hints to do several things.

With FastAPI you declare parameters with type hints and you get:

- Editor support.
- Type checks.
- ...and FastAPI uses the same declarations to:
 - Define requirements: from request path parameters, query parameters, headers, bodies, dependencies, etc.
 - Convert data: from the request to the required type.
 - Validate data: coming from each request.
 - Generating automatic errors: returned to the client when the data is invalid.
 - Document the API using OpenAPI: which is then used by the automatic interactive documentation user interfaces.

Getting started

4.

Basics are done,
let us now focus
on FastAPI

Hello World

Let us focus on FastAPI now!



Hello World

Hello World!



```
1 # 02_getting_started/helloworld.py
2
3 from fastapi import FastAPI
4
5 app = FastAPI()
6
7 @app.get("/")
8 async def root():
9     return {"message": "Hello World"}
```

Run it with:



```
1 cd examples/02_getting_started
2 uvicorn helloworld:app --reload
```

Hello World

Stack-Trace:

```
● ● ●

1 > cd examples/02_getting_started
2 > uvicorn helloworld:app --reload
3 INFO:    Will watch for changes in these directories: ['/Users/davidpinezich/Development/preparation-fastapi/ex
4 INFO:    Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
5 INFO:    Started reloader process [33498] using WatchFiles
6 INFO:    Started server process [33502]
7 INFO:    Waiting for application startup.
8 INFO:    Application startup complete.
```

Our hello world is now happily running on 127.0.0.1:8000

Hello World

Interactive Tooling: <http://127.0.0.1:8000/docs> powered by SwaggerUI

FastAPI 0.1.0 OAS3

[/openapi.json](#)

default

^

GET / Root

Try it out

Parameters

No parameters

Responses

Code	Description	Links
200	Successful Response	No links

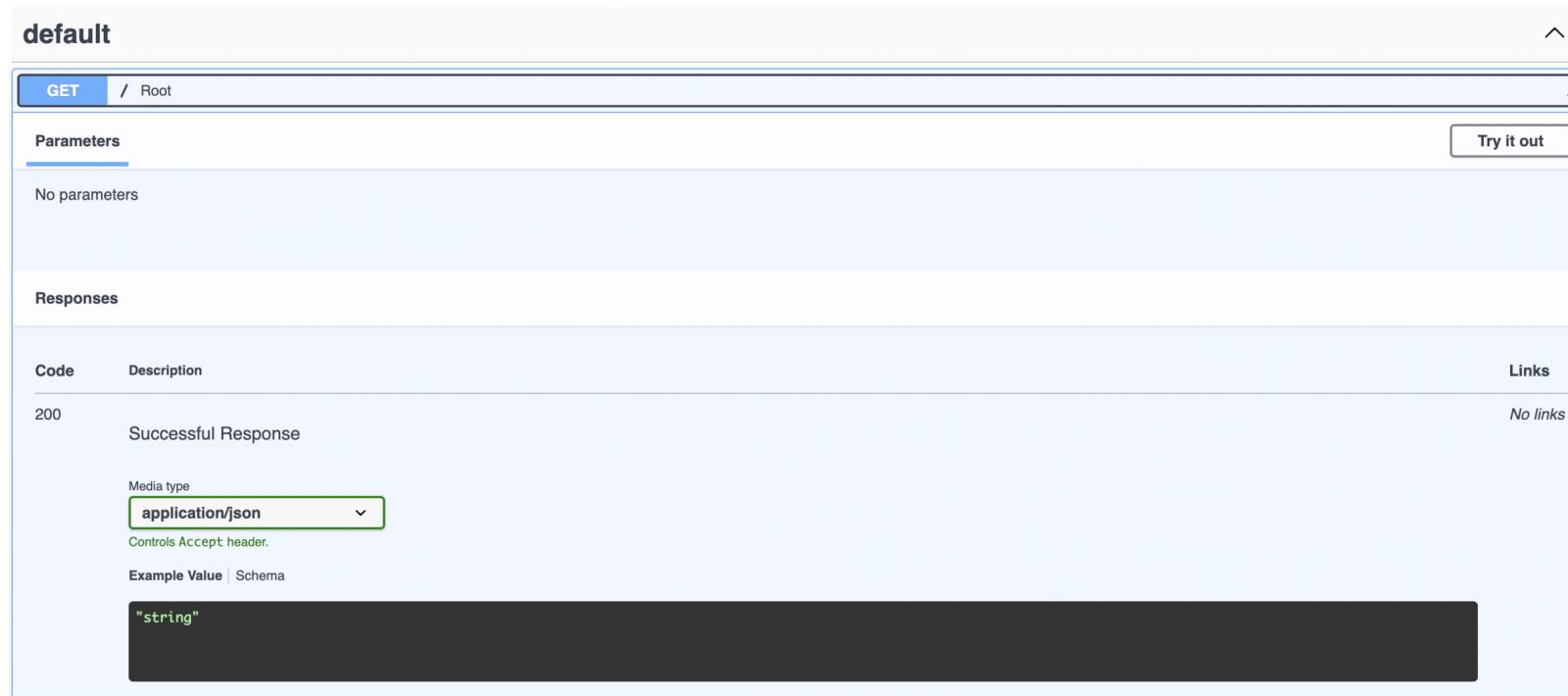
Media type

application/json

Controls Accept header.

Example Value | Schema

"string"



Hello World (alternative)

Interactive Tooling: <http://127.0.0.1:8000/redoc> powered by Redoc

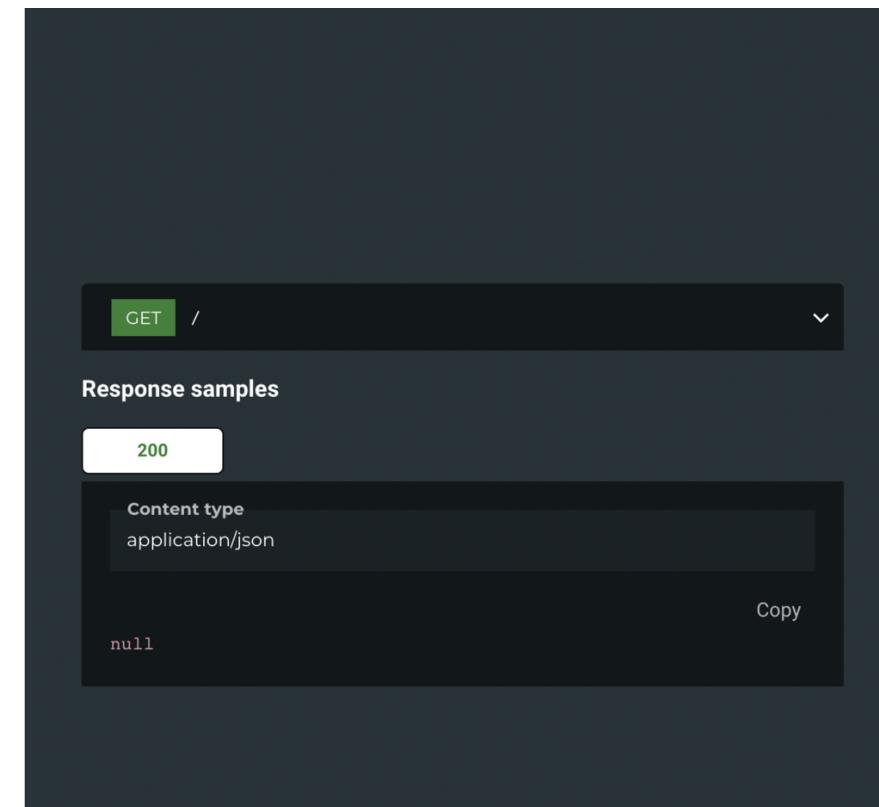
FastAPI (0.1.0)

Download OpenAPI specification: [Download](#)

Root

Responses

> 200 Successful Response



Hello World (OpenAPI)

OpenAPI:

<http://127.0.0.1:8000/openapi.json>

```
4  {
5      "openapi": "3.0.2",
6      "info": {
7          "title": "FastAPI",
8          "version": "0.1.0"
9      },
10     "paths": {
11         "/": {
12             "get": {
13                 "summary": "Root",
14                 "operationId": "root__get",
15                 "responses": {
16                     "200": {
17                         "description": "Successful Response",
18                         "content": {
19                             "application/json": {
20                                 "schema": {
21                                     ...
22                                 }
23                             }
24                         }
25                     }
26                 }
27             }
28         }
29     }
30 }
```

Hello World

Can be found in the folder
"postman" and easily
imported

The screenshot shows the Postman application interface. At the top, there's a navigation bar with tabs for Overview, POST https://te..., POST https://te..., FastAPI Session, GET Hello World, and a '+' button. To the right of the tabs, it says "No Environment". Below the navigation, the title "FastAPI Session / Hello World" is displayed, along with "Save" and "Edit" buttons.

The main area shows a request configuration for a "GET" method to "http://127.0.0.1:8000". Below this, there are tabs for "Params", "Authorization", "Headers (6)", "Body", "Pre-request Script", "Tests", "Settings", and "Cookies". The "Headers (6)" tab is currently selected.

A "Query Params" table is present, with columns for KEY, VALUE, DESCRIPTION, and Bulk Edit. It contains one row with "Key" and "Value" columns, and "Description" as the header for the last column.

At the bottom, there are tabs for "Body", "Cookies", "Headers (4)", and "Test Results". The "Headers (4)" tab is selected. To the right, there are status indicators: 200 OK, 8 ms, 150 B, and a "Save Response" button. Below these, there are buttons for "Pretty", "Raw", "Preview", "Visualize", and "JSON". The JSON response is displayed as:

```
1 {  
2   "message": "Hello World"  
3 }
```

At the very bottom, there are links for "Cookies", "Capture requests", "Bootcamp", "Runner", and "Trash".

Hello World

Hello World!

```
● ● ●  
1 # 02_getting_started/helloworld.py  
2  
3 from fastapi import FastAPI  
4  
5 app = FastAPI() — instance  
6  
7 @app.get("/") — path  
8 async def root():  
9     return {"message": "Hello World"} — Function  
import
```

Run it with:

```
● ● ●  
1 cd examples/02_getting_started  
2 uvicorn helloworld:app --reload
```

The five steps

FastAPI is claimed to be very easy to learn, thanks to the four steps!

1. import FastAPI
2. create a FastAPI "instance"
3. create a path operation
4. define the path operation function
5. return the content

The five steps

1. import FastAPI



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5
6 @app.get("/")
7 async def root():
8     return {"message": "Hello World"}
```

The five steps

2. create a FastAPI "instance"



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5
6 @app.get("/")
7 async def root():
8     return {"message": "Hello World"}
```

The five steps

2. create a FastAPI "instance"



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5
6 @app.get("/")
7 async def root():
8     return {"message": "Hello World"}
```

Short HTTP flashback

Classic HTML-Methods:

- POST
- GET
- PUT
- DELETE

More exotic ones:

- OPTIONS
- HEAD
- PATCH
- TRACE

Short HTTP flashback

Classic HTML-Methods:

- POST
 - to create data.
- GET
 - to read data.
- PUT
 - to update data.
- DELETE
 - to delete data.

Short HTTP flashback

Classic HTML-Methods:

- POST
 - @app.post()
- GET
 - @app.get()
- PUT
 - @app.put()
- DELETE
 - @app.delete()

Short HTTP flashback

Classic HTML-Methods:

- OPTIONS
 - @app.options()
- HEAD
 - @app.head()
- PATCH
 - @app.patch()
- TRACE
 - @app.trace()

The five steps

4. define the path operation function



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5
6 @app.get("/")
7 async def root():
8     return {"message": "Hello World"}
```

The five steps

5. Return the content



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5
6 @app.get("/")
7 async def root():
8     return {"message": "Hello World"}
```

Getting started!

To build a solid foundation, we will now start with various topics of FastAPI.



Path Parameters

Let us inspect the path parameters in detail.



Items and their types

Hello World!



```
1 # 02_getting_started/withtypes.py
2
3 from fastapi import FastAPI
4 application = FastAPI()
5
6 @application.get("/items/{item_id}")
7 async def read_item(item_id: int):
8     return {"item_id": item_id}
```

How do you run it?



```
1 cd examples/02_getting_started
2 uvicorn withtypes:application --reload
```

Items and their types

What happens if you try:

```
● ● ●  
1 # 02_getting_started/helloworld.py  
2  
3 http://127.0.0.1:8000/items/foo
```

A well documented error message:

```
● ● ●  
1 {  
2     "detail": [  
3         {  
4             "loc": [  
5                 "path",  
6                 "item_id"  
7             ],  
8             "msg": "value is not a valid integer",  
9             "type": "type_error.integer"  
10        }  
11    ]  
12 }
```

Order matters

If you have multiple routes, be careful!

The rule of thumb is to always: **more definitive first**.



```
1 # 02_getting_started/order.py
2
3 from fastapi import FastAPI
4
5 app = FastAPI()
6
7
8 @app.get("/users/me")
9 async def read_user_me():
10     return {"user_id": "the current user"}
11
12
13 @app.get("/users/{user_id}")
14 async def read_user(user_id: str):
15     return {"user_id": user_id}
```

Predefined values

Look at this enum-rich example



```
1 # 02_getting_started/enumrich.py
2
3 from enum import Enum
4 from fastapi import FastAPI
5
6 class ModelName(str, Enum):
7     alexnet = "alexnet"
8     resnet = "resnet"
9     lenet = "lenet"
10
11                                         It gives you all the hints needed
12 app = FastAPI()
13
14 @app.get("/models/{model_name}")
15 async def get_model(model_name: ModelName):
16     if model_name is ModelName.alexnet:
17         return {"model_name": model_name, "message": "Deep Learning FTW!"}
18
19     if model_name.value == "lenet":
20         return {"model_name": model_name, "message": "LeCNN all the images"}
21
22     return {"model_name": model_name, "message": "Have some residuals"}
```

Predefined values

But we can do it even better:

```
● ● ●
1 # 02_getting_started/enumrich_better.py
2
3 from enum import Enum
4 from fastapi import FastAPI
5
6 class ModelName(str, Enum):
7     alexnet = "alexnet"
8     resnet = "resnet"
9     lenet = "lenet"
10
11 app = FastAPI()
12
13 @app.get("/models/{model_name}")
14 async def get_model(model_name: ModelName):
15     if model_name is ModelName.alexnet:
16         return {"model_name": model_name, "message": "Deep Learning FTW!"}
17
18     if model_name.value == "lenet":
19         return {"model_name": model_name, "message": "LeCNN all the images"}
20
21     return {"model_name": model_name, "message": "Have some residuals"}
```

Predefined values

Let us quickly check the docs to see what happened:



```
1 uvicorn enumrich_better:app --reload
```

default

The screenshot shows a Swagger UI interface for a 'GET /models/{model_name}' endpoint. The 'Parameters' section is active, displaying a table with one row. The row has two columns: 'Name' and 'Description'. The 'Name' column contains 'model_name * required' with a red asterisk, 'string', and '(path)' below it. The 'Description' column contains a dropdown menu with three options: 'alexnet' (which is selected and highlighted in blue), 'resnet', and 'lenet'. A 'Cancel' button is visible in the top right corner of the dropdown. At the bottom of the interface is a large blue 'Execute' button.

Name	Description
model_name * required string (path)	<input checked="" type="text"/> alexnet resnet lenet

Execute

Query Parameters

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters.

```
● ● ●  
1 # 02_getting_started/query.py  
2 #  
3 app = FastAPI()  
4  
5 fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]  
6  
7 @app.get("/items/")  
8 async def read_item(skip: int = 0, limit: int = 10):  
9     return fake_items_db[skip : skip + limit]
```

```
● ● ●  
1 uvicorn query:app --reload  
2 http://127.0.0.1:8000/items/?skip=0&limit=10
```

Query Parameters (Optional)

The same way, you can declare optional query parameters, by setting their default to `None`.

Python 3.6+

```
● ● ●
1 # 02_getting_started/query_optional.py
2 from typing import Union
3 from fastapi import FastAPI
4
5 app = FastAPI()
6
7 @app.get("/items/{item_id}")
8 async def read_item(item_id: str, q: Union[str, None] = None):
9     if q:
10         return {"item_id": item_id, "q": q}
11     return {"item_id": item_id}
```

Query Parameters (Optional)

The same way, you can declare optional query parameters, by setting their default to `None`.

Python 3.10+

```
1 # 02_getting_started/query_optional_3_10.py
2
3 from fastapi import FastAPI
4 app = FastAPI()
5
6 @app.get("/items/{item_id}")
7 async def read_item(item_id: str, q: str | None = None):
8     if q:
9         return {"item_id": item_id, "q": q}
10    return {"item_id": item_id}
```

Multiple path and query parameters

You can declare multiple path parameters and query parameters at the same time, FastAPI knows which is which.



```
1 # 02_getting_started/multiple_query.py
2
3 from typing import Union
4 from fastapi import FastAPI
5
6 app = FastAPI()
7
8 @app.get("/users/{user_id}/items/{item_id}")
9 async def read_user_item(
10     user_id: int, item_id: str, q: Union[str, None] = None, short: bool = False
11 ):
12     item = {"item_id": item_id, "owner_id": user_id}
13     if q:
14         item.update({"q": q})
15     if not short:
16         item.update(
17             {"description": "This is an amazing item that has a long description"}
18         )
19     return item
```

Required query parameters

In this example we have item_id default set and needy is needed :-)

```
● ● ●  
1 # 02_getting_started/required.py  
2  
3 from fastapi import FastAPI  
4 app = FastAPI()  
5  
6 @app.get("/items/{item_id}")  
7 async def read_user_item(item_id: str, needy: str):  
8     item = {"item_id": item_id, "needy": needy}  
9     return item
```

Required query parameters

In this example we have item_id default set and needy is needed :-)

```
1 // 20220927223512
2 // http://127.0.0.1:8000/items/foo-item
3
4 {
5     "detail": [
6         {
7             "loc": [
8                 "query",
9                 "needy"
10            ],
11            "msg": "field required",
12            "type": "value_error.missing"
13        }
14    ]
15 }
```

Required query parameters

Now we set the parameter needy=fulfilled and everything is ok.



A screenshot of a browser developer tools Network tab. The URL bar shows the address: 127.0.0.1:8000/items/foo-item?needy=fulfilled. The main content area displays the following JSON response:

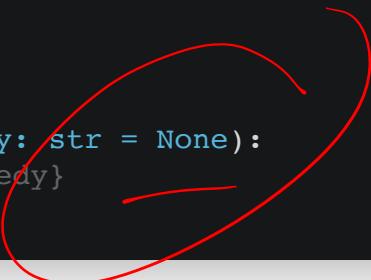
```
// 2022092723631
// http://127.0.0.1:8000/items/foo-item?needy=fulfilled
{
  "item_id": "foo-item",
  "needy": "fulfilled"
}
```

Required query parameters

If you don't want to add a specific value but just make it optional, set the default as None.



```
1 # 02_getting_started/not_required.py
2
3 from fastapi import FastAPI
4 app = FastAPI()
5
6 @app.get("/items/{item_id}")
7 async def read_user_item(item_id: str, needy: str = None):
8     item = {"item_id": item_id, "needy": needy}
9     return item
```



Required query parameters

If you don't want to add a specific value but just make it optional, set the default as None.



A screenshot of a browser developer tools network tab. The address bar shows the URL `127.0.0.1:8000/items/foo-item`. The request method is POST. The response body is a JSON object:

```
1 // 20220927224054
2 // http://127.0.0.1:8000/items/foo-item
3
4 {
5     "item_id": "foo-item",
6     "needy": null
7 }
```

Body parameters

Let us inspect the body parameters in detail.



Body parameters

When you need to send data from a client (let's say, a browser) to your API, you send it as a request body.

To declare a request body, you use Pydantic models with all their power and benefits.

BaseModel

Let us have a look at the BaseModel:



```
1 # 02_getting_started/base_model.py
2
3 from typing import Union
4 from fastapi import FastAPI
5 from pydantic import BaseModel
6
7 class Item(BaseModel):
8     name: str
9     description: Union[str, None] = None
10    price: float
11    tax: Union[float, None] = None
12
13 app = FastAPI()
14
15 @app.post("/items/")
16 async def create_item(item: Item):
17     return item
```

BaseModel

For example, this model above declares a JSON "object" (or Python dict) like:

```
● ● ●  
1 {  
2     "name": "Foo",  
3     "description": "An optional description",  
4     "price": 45.2,  
5     "tax": 3.5  
6 }
```

or... (because of optionals)

```
● ● ●  
1 {  
2     "name": "Foo",  
3     "price": 45.2  
4 }
```

BaseModel

The screenshot shows the Fast API Swagger UI interface at `127.0.0.1:8000/docs`. The title bar indicates it's version 0.1.0 OAS3. The main area displays a **default** endpoint for a **POST** request to `/items/` labeled "Create Item Post".

Parameters: No parameters.

Request body (required): `application/json`

Example Value | Model:

```
{
    "name": "string",
    "price": 0,
    "description": "string",
    "tax": 0
}
```

Responses:

Code	Description	Links
200	Successful Response Controls Accept header. <code>application/json</code>	No links
422	Validation Error <code>application/json</code>	No links

BaseModel

Editor support? EDITOR SUPPORT! (see 02_getting_started/base_model_editor.py)

```
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.| You, a few seconds ago • Uncommitted changes
18     return item.capitalize()  # Hovered over this line
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1196
1197
1198
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1208
1209
1210
1211
1212
1213
1214
1215
1216
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1308
1309
1310
1311
1312
1313
1314
1315
1316
1316
1317
1318
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1408
1409
1410
1411
1412
1413
1414
1415
1416
1416
1417
1418
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1447
1447
1448
1449
1449
1450
1451
1452
1453
1454
1455
1456
1457
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1466
1467
1468
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1496
1496
1497
1498
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1508
1509
1510
1511
1512
1513
1514
1515
1516
1516
1517
1518
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1537
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1547
1547
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1558
1559
1560
1561
1562
1563
1564
1565
1566
1566
1567
1568
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1608
1609
1610
1611
1612
1613
1614
1615
1616
1616
1617
1618
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1637
1638
1639
1639
1640
1641
1642
1643
1644
1645
1646
1647
1647
1648
1649
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1708
1709
1710
1711
1712
1713
1714
1715
1716
1716
1717
1718
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1737
1738
1739
1739
1740
1741
1742
1743
1744
1745
1746
1747
1747
1748
1749
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1758
1759
1760
1761
1762
1763
1764
1765
1766
1766
1767
1768
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1808
1809
1810
1811
1812
1813
1814
1815
1816
1816
1817
1818
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1908
1909
1910
1911
1912
1913
1914
1915
1916
1916
1917
1918
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2108
2109
2110
2111
2112
2113
2114
2115
2116
2116
2117
2118
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2208
2209
2210
2211
2212
2213
2214
2215
2216
2216
2217
2218
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2296
2297
2298
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2308
2309
2310
2311
2312
2313
2314
2315
2316
2316
2317
2318
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2387
2388
2389
2389
2390
2391

```

BaseModel

Now let us use the base model:

```
● ● ●  
1 # 02_getting_started/use_base_model.py  
2  
3 from typing import Union  
4  
5 from fastapi import FastAPI  
6 from pydantic import BaseModel  
7  
8 class Item(BaseModel):  
9     name: str  
10    description: Union[str, None] = None  
11    price: float  
12    tax: Union[float, None] = None  
13  
14 app = FastAPI()  
15  
16 @app.post("/items/")  
17 async def create_item(item: Item):  
18     item_dict = item.dict()  
19     if item.tax:  
20         price_with_tax = item.price + item.tax  
21         item_dict.update({"price_with_tax": price_with_tax})  
22     return item_dict
```

Path & Body

Now a combination of request body and path parameters

```
1 # 02_getting_started/request_body_path.py
2
3 from typing import Union
4
5 from fastapi import FastAPI
6 from pydantic import BaseModel
7
8 class Item(BaseModel):
9     name: str
10    description: Union[str, None] = None
11    price: float
12    tax: Union[float, None] = None
13
14 app = FastAPI()
15
16 @app.put("/items/{item_id}")
17 async def create_item(item_id: int, item: Item):
18     return {"item_id": item_id, **item.dict()}
```

Path & Body & Query

Now a combination of request body, path parameters and query

```
1 # 02_getting_started/request_body_path_query.py
2
3 from typing import Union
4
5 from fastapi import FastAPI
6 from pydantic import BaseModel
7
8 class Item(BaseModel):
9     name: str
10    description: Union[str, None] = None
11    price: float
12    tax: Union[float, None] = None
13
14 app = FastAPI()
15
16 @app.put("/items/{item_id}")
17 async def create_item(item_id: int, item: Item, q: Union[str, None] = None):
18     result = {"item_id": item_id, **item.dict()}
19     if q:
20         result.update({"q": q})
21     return result
```

Path & Body & Query

default ^

PUT /items/{item_id} Create Item ^

Parameters Try it out

Name	Description
item_id * required integer (path)	item_id
q string (query)	q

Request body required application/json ▾

Example Value | Schema

```
{  
  "name": "string",  
  "description": "string",  
  "price": 0,  
  "tax": 0  
}
```

Mix Path, Query and body parameters



```
1 # 02_getting_started/multiple_parameters.py
2
3 from typing import Union
4 from fastapi import FastAPI, Path
5 from pydantic import BaseModel
6 app = FastAPI()
7
8 class Item(BaseModel):
9     name: str
10    description: Union[str, None] = None
11    price: float
12    tax: Union[float, None] = None
13
14 @app.put("/items/{item_id}")
15 async def update_item(
16     *,
17     item_id: int = Path(title="The ID of the item to get", ge=0, le=1000),
18     q: Union[str, None] = None,
19     item: Union[Item, None] = None,
20 ):
21     results = {"item_id": item_id}
22     if q:
23         results.update({"q": q})
24     if item:
25         results.update({"item": item})
26     return results
```

Multiple Parameters



```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6 class Item(BaseModel):
7     name: str
8     description: str | None = None
9     price: float
10    tax: float | None = None
11
12
13 class User(BaseModel):
14     username: str
15     full_name: str | None = None
16
17
18 @app.put("/items/{item_id}")
19 async def update_item(item_id: int, item: Item, user: User):
20     results = {"item_id": item_id, "item": item, "user": user}
21     return results
```

Please have a look at the classes Item and User.

Body - Fields

The same way you can declare additional validation and metadata in path operation function parameters with Query, Path and Body, you can declare validation and metadata inside of Pydantic models using Pydantic's Field.



```
1 # 02_getting_started/body_fields.py
2
3 from typing import Union
4 from fastapi import Body, FastAPI
5 from pydantic import BaseModel, Field
6
7 app = FastAPI()
8
9 class Item(BaseModel):
10     name: str
11     description: Union[str, None] = Field(default=None, title="The description of the item", max_length=300)
12     price: float = Field(gt=0, description="The price must be greater than zero")
13     tax: Union[float, None] = None
14
15 @app.put("/items/{item_id}")
16 async def update_item(item_id: int, item: Item = Body(embed=True)):
17     results = {"item_id": item_id, "item": item}
18     return results
```

Body - Fields

Notice that Field is imported directly from pydantic, not from fastapi as are all the rest (Query, Path, Body, etc).



```
1 # 02_getting_started/body_fields.py
2
3 from typing import Union
4 from fastapi import Body, FastAPI
5 from pydantic import BaseModel, Field
6
7 app = FastAPI()
8
9 class Item(BaseModel):
10     name: str
11     description: Union[str, None] = Field(default=None, title="The description of the item", max_length=300)
12     price: float = Field(gt=0, description="The price must be greater than zero")
13     tax: Union[float, None] = None
14
15 @app.put("/items/{item_id}")
16 async def update_item(item_id: int, item: Item = Body(embed=True)):
17     results = {"item_id": item_id, "item": item}
18     return results
```

Body - Nested Models

You can define an attribute to be a subtype. For example, a Python list:

```
● ● ●  
1 # 02_getting_started/nested_models.py  
2  
3 from typing import Union  
4 from fastapi import FastAPI  
5 from pydantic import BaseModel  
6  
7 app = FastAPI()  
8  
9 class Item(BaseModel):  
10     name: str  
11     description: Union[str, None] = None  
12     price: float  
13     tax: Union[float, None] = None  
14     tags: list = []  
15  
16 @app.put("/items/{item_id}")  
17 async def update_item(item_id: int, item: Item):  
18     results = {"item_id": item_id, "item": item}  
19     return results
```

This will make tags be a list, but does not declare the type of the elements.

Body - Nested Models

In Python 3.9 and above you can use the standard list to declare these type annotations as we'll see below.



```
1 # 02_getting_started/typed_nested_models.py
2
3 from typing import List, Union
4 from fastapi import FastAPI
5 from pydantic import BaseModel
6
7 app = FastAPI()
8
9 class Item(BaseModel):
10     name: str
11     description: Union[str, None] = None
12     price: float
13     tax: Union[float, None] = None
14     tags: List[str] = []
15
16 @app.put("/items/{item_id}")
17 async def update_item(item_id: int, item: Item):
18     results = {"item_id": item_id, "item": item}
19     return results
```

Extra Data Types

Up to now, you have been using common data types, like:

- int
- float
- str
- bool

Extra Data Types

But you can also use more complex data types.

- UUID
 - A standard "Universally Unique Identifier", common as an ID in many databases and systems.
- datetime.datetime, datetime.date, datetime.time, datetime.timedelta
 - Python datetime.XXX
- frozenset
 - In requests and responses, treated the same as a set.
- bytes
 - Standard Python bytes.
- Decimal
 - Standard Python Decimal.

Extra Data Types



```
1 # 02_getting_started/extradata_types.py
2
3 from datetime import datetime, time, timedelta
4 from typing import Union
5 from uuid import UUID
6 from fastapi import Body, FastAPI
7
8 app = FastAPI()
9
10 @app.put("/items/{item_id}")
11 async def read_items(
12     item_id: UUID,
13     start_datetime: Union[datetime, None] = Body(default=None),
14     end_datetime: Union[datetime, None] = Body(default=None),
15     repeat_at: Union[time, None] = Body(default=None),
16     process_after: Union[timedelta, None] = Body(default=None),
17     ):
18     start_process = start_datetime + process_after
19     duration = end_datetime - start_process
20     return {
21         "item_id": item_id,
22         "start_datetime": start_datetime,
23         "end_datetime": end_datetime,
24         "repeat_at": repeat_at,
25         "process_after": process_after,
26         "start_process": start_process,
27         "duration": duration,
28     }
```

Cookies & Header

You can define Cookie & Header parameters the same way you define Query and Path parameters.



```
1 from typing import Union
2 from fastapi import Cookie, FastAPI
3
4 app = FastAPI()
5
6 @app.get("/items/")
7 async def read_items(ads_id: Union[str, None] = Cookie(default=None)):
8     return {"ads_id": ads_id}
```



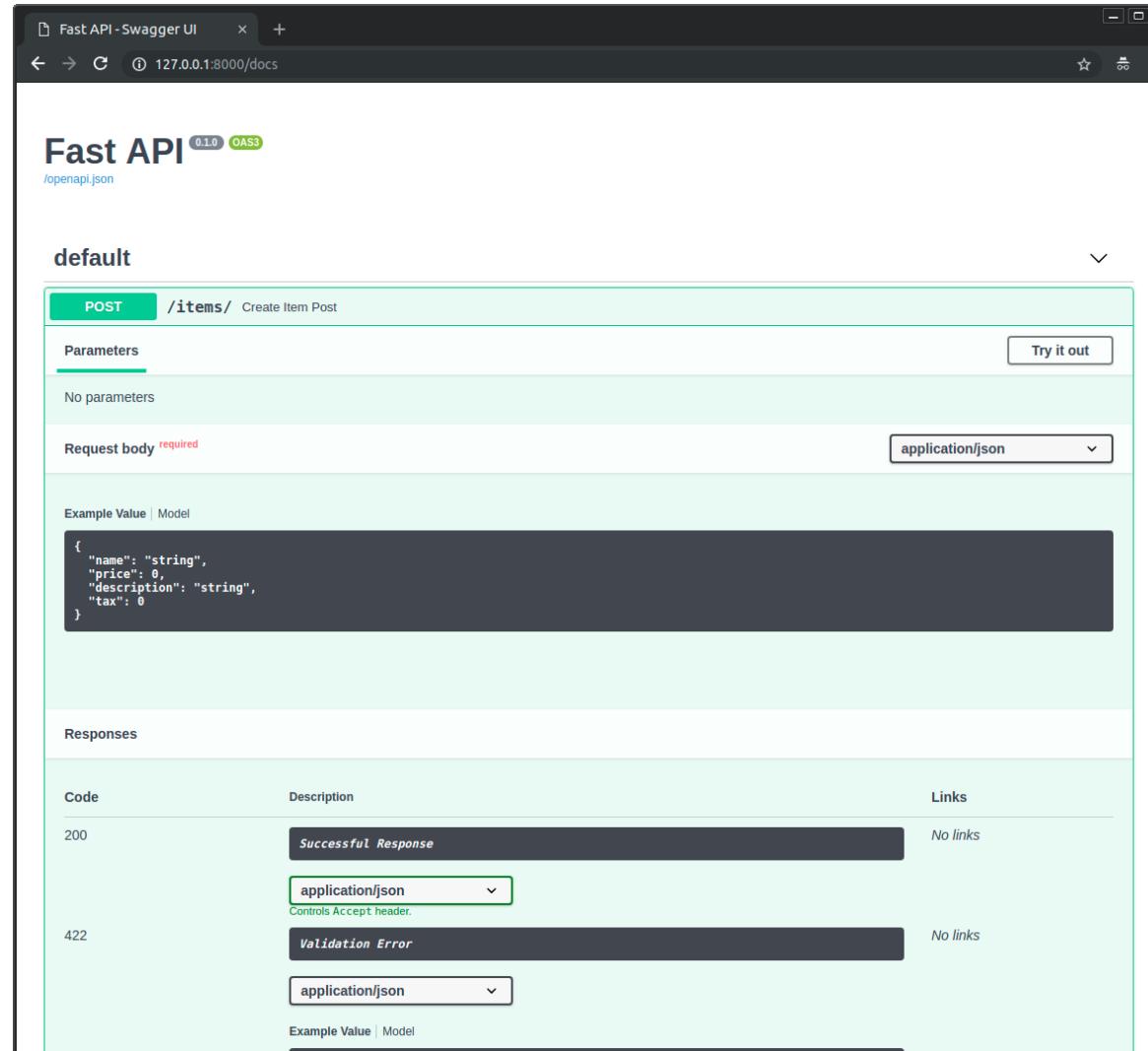
```
1 from typing import Union
2 from fastapi import FastAPI, Header
3
4 app = FastAPI()
5
6 @app.get("/items/")
7 async def read_items(user_agent: Union[str, None] = Header(default=None)):
8     return {"User-Agent": user_agent}
```

Response Status Code

The same way you can specify a response model, you can also declare the HTTP status code used for the response with the parameter `status_code` in any of the path operations:

```
● ● ●  
1 from fastapi import FastAPI  
2  
3 app = FastAPI()  
4  
5 @app.post("/items/", status_code=201)  
6 async def create_item(name: str):  
7     return {"name": name}
```

Response Status Code



Form Data

When you need to receive form fields instead of JSON, you can use Form.



```
1 from fastapi import FastAPI, Form
2
3 app = FastAPI()
4
5 @app.post("/login/")
6 async def login(username: str = Form(), password: str = Form()):
7     return {"username": username}
```

The way HTML forms (<form></form>) sends the data to the server normally uses a "special" encoding for that data; it is different from JSON.

FastAPI will make sure to read that data from the right place instead of JSON.

Exercise



Please head into the "exercises": 01_getting_started



```
1 cat = Animal(  
2     name="Cat",  
3     description="This is a small cat",  
4     height="0.2",  
5     weight=3.5,  
6     abilities=["fast", "cute", "small"]  
7 )
```



Handling Errors

There are many situations in where you need to notify an error to a client that is using your API.

You could need to tell the client that:

- The client doesn't have enough privileges for that operation.
- The client doesn't have access to that resource.
- The item the client was trying to access doesn't exist.
- etc.

HTTPException

To return HTTP responses with errors to the client you use `HTTPException`.



```
1 from fastapi import FastAPI, HTTPException
2
3 app = FastAPI()
4
5 items = {"foo": "The Foo Wrestlers"}
6
7 @app.get("/items/{item_id}")
8 async def read_item(item_id: str):
9     if item_id not in items:
10         raise HTTPException(status_code=404, detail="Item not found")
11     return {"item": items[item_id]}
```

Custom Errors

You could add a custom exception handler with `@app.exception_handler()`:



```
1 from fastapi import FastAPI, Request
2 from fastapi.responses import JSONResponse
3
4 class UnicornException(Exception):
5     def __init__(self, name: str):
6         self.name = name
7
8
9 app = FastAPI()
10
11 @app.exception_handler(UnicornException)
12 async def unicorn_exception_handler(request: Request, exc: UnicornException):
13     return JSONResponse(
14         status_code=418,
15         content={"message": f"Oops! {exc.name} did something. There goes a rainbow..."},)
16
17 @app.get("/unicorns/{name}")
18 async def read_unicorn(name: str):
19     if name == "yolo":
20         raise UnicornException(name=name)
21     return {"unicorn_name": name}
```

5. Security & Benchmark

Is it really
secure and
efficient?

Security

FastAPI can work with many modern security aspects:

OpenID Connect,

OAuth 1 + 2

OpenAPI

and many more...

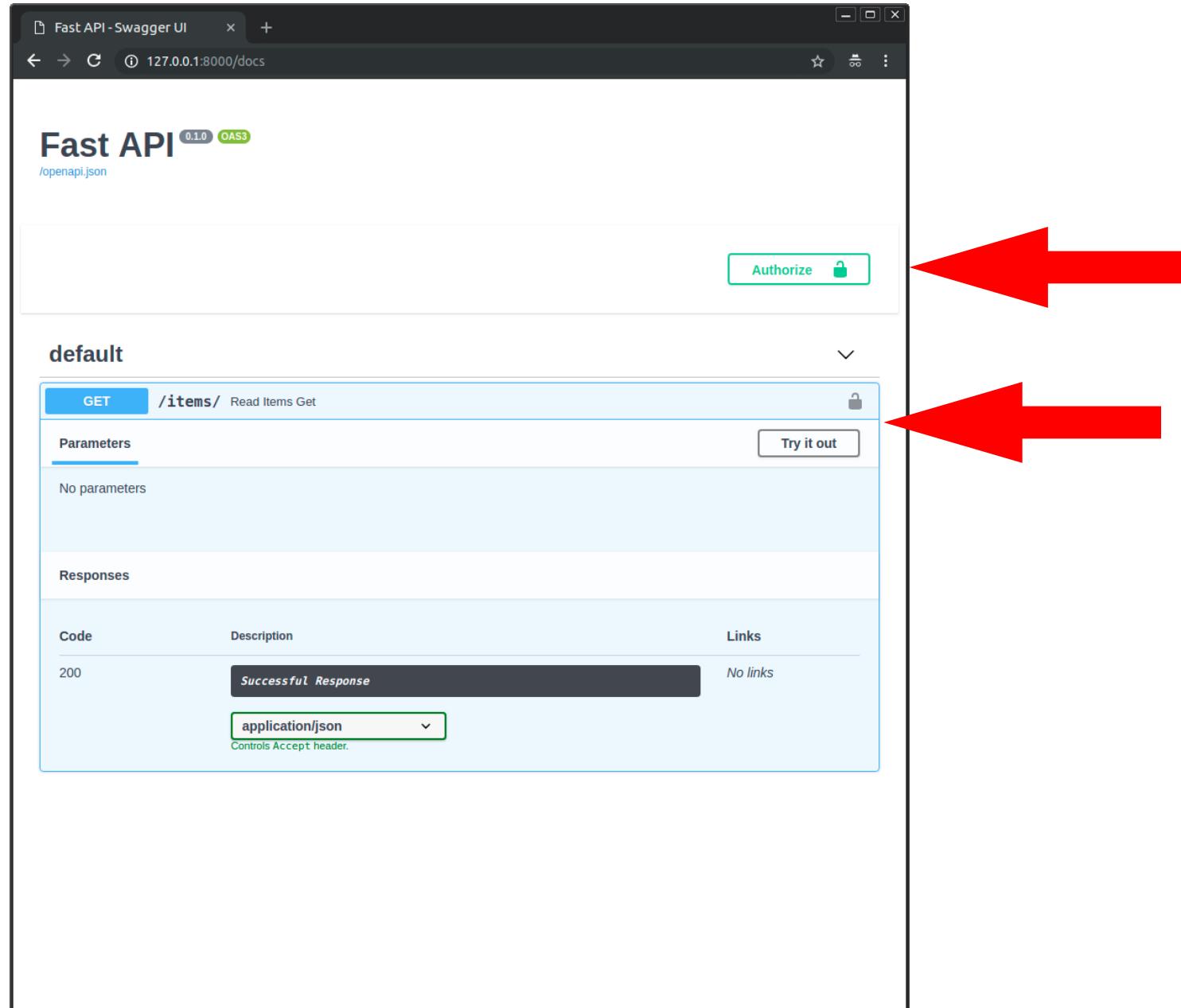
Security

Here a small example:

```
● ● ●  
1 from fastapi import Depends, FastAPI  
2 from fastapi.security import OAuth2PasswordBearer  
3  
4 app = FastAPI()  
5  
6 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")  
7  
8  
9 @app.get("/items/")  
10 async def read_items(token: str = Depends(oauth2_scheme)):  
11     return {"token": token}
```

Easy right?

Security



Security

```
 1 from typing import Union
 2
 3 from fastapi import Depends, FastAPI
 4 from fastapi.security import OAuth2PasswordBearer
 5 from pydantic import BaseModel
 6
 7 app = FastAPI()
 8
 9 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
10
11 class User(BaseModel):
12     username: str
13     email: Union[str, None] = None
14     full_name: Union[str, None] = None
15     disabled: Union[bool, None] = None
16
17
18 def fake_decode_token(token):
19     return User(
20         username=token + "fakedecoded", email="john@example.com", full_name="John Doe"
21     )
22
23 async def get_current_user(token: str = Depends(oauth2_scheme)):
24     user = fake_decode_token(token)
25     return user
26
27 @app.get("/users/me")
28 async def read_users_me(current_user: User = Depends(get_current_user)):
29     return current_user
```

Benchmarks

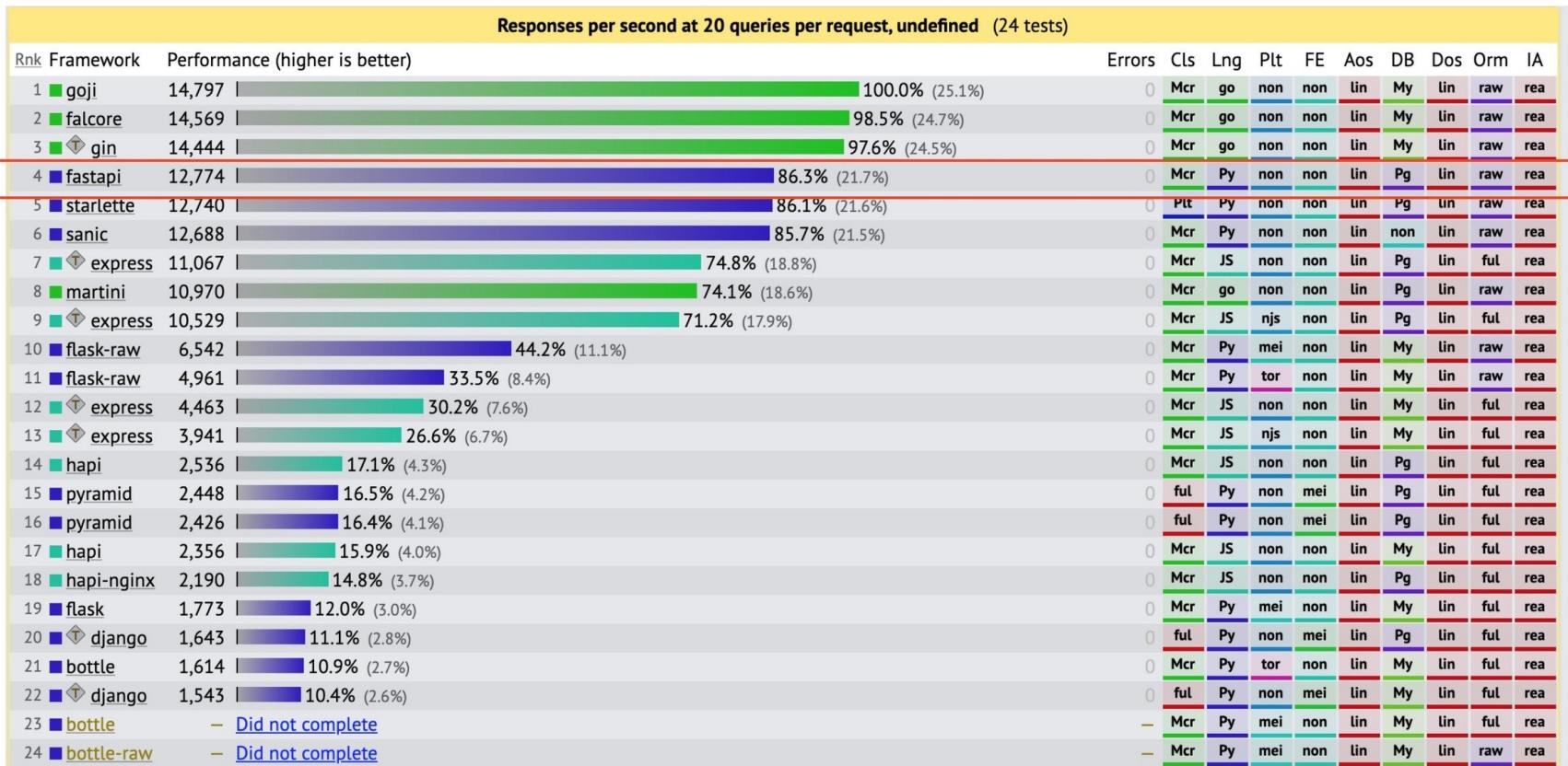
- Uvicorn: an ASGI server
 - Starlette: (uses Uvicorn) a web microframework
 - FastAPI: (uses Starlette) an API microframework with several additional features for building APIs, with data validation, etc.

Benchmarks

	Github Stars	Github Forks	Launch Year
Flask	55.3k	14.3k	2011
Django	57.5k	24.5k	2006
FastAPI	31.1k	2.2k	2019

Benchmarks

Is it really comparable to NodeJS or Go? Yes!



Credits: 2021; <https://christophergs.com/python/2021/06/16/python-flask-fastapi/>

6. Project

Final Project

Databases

FastAPI supports multiple databases, such as

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server, and so on.

We will focus on SQLite to make it less complex.

SQLAlchemy

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

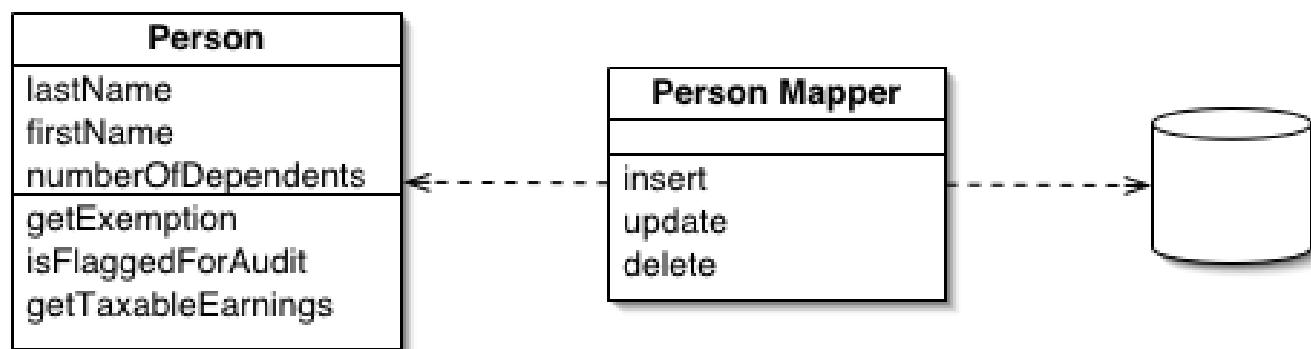
SQLAlchemy is most famous for its object-relational mapper (ORM), an optional component that provides the **data mapper pattern**.



Data Mapper Pattern

A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

See also: <https://martinfowler.com/eaaCatalog/dataMapper.html>



What we will build

Sample FastAPI Application 1.0.0 OAS3

[/openapi.json](#)

Sample FastAPI Application with Swagger and SQLAlchemy

Item

^ ▼

GET	/items	Get All Items
POST	/items	Create Item
GET	/items/{item_id}	Get Item
PUT	/items/{item_id}	Update Item
DELETE	/items/{item_id}	Delete Item

Store

^ ▼

GET	/stores	Get All Stores
POST	/stores	Create Store
GET	/stores/{store_id}	Get Store
DELETE	/stores/{store_id}	Delete Store

Prepare Database

Have a look at the db.py file.



```
1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from sqlalchemy.orm import sessionmaker
4
5 SQLALCHEMY_DATABASE_URL = "sqlite:///./data.db"
6
7
8 engine = create_engine(
9     SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}, echo=True
10 )
11 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
12
13 Base = declarative_base()
14
15
16 # Dependency
17 def get_db():
18     db = SessionLocal()
19     try:
20         yield db
21     finally:
22         db.close()
```

Prepare Database

What does it do:

- Imports packages which are required to create SQLAlchemy **engine** and database session to connect to the **SQLite** database.
- Created the SQLAlchemy **engine** using the database created above.
- **SessionLocal** class represents database session.
- The function **declarative_base()** that returns a class to create **Base** class.
- We also defined a function called **get_db()** , which can be used to create independent database session for each request.

Create the model

Have a look at models.py:



```
1 from sqlalchemy import Column, ForeignKey, Integer, String, Float
2 from sqlalchemy.orm import relationship
3 from db import Base
4
5 class Item(Base):
6     __tablename__ = "items"
7     id = Column(Integer, primary_key=True, index=True)
8     name = Column(String(80), nullable=False, unique=True, index=True)
9     price = Column(Float(precision=2), nullable=False)
10    description = Column(String(200))
11    store_id = Column(Integer, ForeignKey('stores.id'), nullable=False)
12    def __repr__(self):
13        return 'ItemModel(name=%s, price=%s, store_id=%s)' % (self.name, self.price, self.store_id)
14
15 class Store(Base):
16     __tablename__ = "stores"
17     id = Column(Integer, primary_key=True, index=True)
18     name = Column(String(80), nullable=False, unique=True)
19     items = relationship("Item", primaryjoin="Store.id == Item.store_id", cascade="all, delete-orphan")
20
21    def __repr__(self):
22        return 'Store(name=%s)' % self.name
```

Create the schema

This file will contain the **Pydantic** models for our SQLAlchemy models.

These Pydantic models define more or less a schema (or valid data shape).

Some extract from the official documentation:

Pydantic is primarily a parsing library, not a validation library.

Validation is a means to an end: building a model which conforms to the types and constraints provided. In other words,

`pydantic` guarantees the types and constraints of the output model, not the input data.

Create the schema

Let us have a look at this simplified example:

```
● ● ●  
1 class ItemBase(BaseModel):  
2     name: str  
3     price: float  
4     description: Optional[str] = None  
5     store_id: int  
6  
7 class ItemCreate(ItemBase):  
8     pass  
9  
10 class Item(ItemBase):  
11    id: int  
12    class Config:  
13        orm_mode = True
```

Create the schema



```
1 class ItemBase(BaseModel):
2     name: str
3     price: float
4     description: Optional[str] = None
5     store_id: int
6
7 class ItemCreate(ItemBase):
8     pass
9
10 class Item(ItemBase):
11     id: int
12     class Config:
13         orm_mode = True
```

First: We create `ItemBase` derived from `BaseModel` (we know already). This class contain the common attributes, which we need while creating or reading data.

Create the schema

● ● ●

```
1 class ItemBase(BaseModel):
2     name: str
3     price: float
4     description: Optional[str] = None
5     store_id: int
6
7 class ItemCreate(ItemBase):
8     pass
9
10 class Item(ItemBase):
11     id: int
12     class Config:
13         orm_mode = True
```

ItemCreate (or StoreCreate) inherit from ItemBase (and StoreBase).

Thus, they will have all the attributes of the Parent class, plus any additional data (attributes) needed for creation. None here for simplicity.

Create the schema



```
1 class ItemBase(BaseModel):
2     name: str
3     price: float
4     description: Optional[str] = None
5     store_id: int
6
7 class ItemCreate(ItemBase):
8     pass
9
10 class Item(ItemBase):
11     id: int
12     class Config:
13         orm_mode = True
```

Finally, we created Pydantic models (schemas) Item and Store that will be used to read the data from the database and returning it from the API including an added internal Config class.

Main

Now we head into the main.py file to create our HTTP-routes.

Main

The includes have already been done, and the app has a bit more description:



```
1 # Some configuration added
2 app = FastAPI(title="Sample FastAPI Application for Demo",
3                 description="Nice to show how easy an API is created!",
4                 version="1.0.0", )
```

Main

In this line, we create all the tables in the database during the application startup using the SQLAlchemy models defined.



```
1 models.Base.metadata.create_all(bind=engine)
```

Main - Exception

Some exception-handling because we know how to do it:



```
1 @app.exception_handler(Exception)
2 def validation_exception_handler(request, err):
3     base_error_message = f"Failed to execute: {request.method}: {request.url}"
4     return JSONResponse(status_code=400, content={"message": f"{base_error_message}. Detail: {err}"})
```

Middleware

You can add middleware to FastAPI applications.

A "middleware" is a function that works with every request before it is processed by any specific path operation. And also with every response before returning it.

Here we are going to use `http`.



```
1 @app.middleware("http")
```

Middleware

Some middleware magic!



```
1 @app.middleware("http")
2 async def add_process_time_header(request, call_next):
3     print('inside middleware!')
4     start_time = time.time()
5     response = await call_next(request)
6     process_time = time.time() - start_time
7     response.headers["X-Process-Time"] = str(f'{process_time:.4f} sec')
8     return response
```

Adds a processing time for the request.

Main - Item

POST-Request



```
1 @app.post('/items', tags=["Item"], response_model=schemas.Item, status_code=201)
2 async def create_item(item_request: schemas.ItemCreate, db: Session = Depends(get_db)):
3     """
4         Create an Item and store it in the database
5     """
6     db_item = ItemRepo.fetch_by_name(db, name=item_request.name)
7     if db_item:
8         raise HTTPException(status_code=400, detail="Item already exists!")
9
10    return await ItemRepo.create(db=db, item=item_request)
```

Main - Item

GET-Request (all items)



```
1 @app.get('/items', tags=["Item"], response_model=List[schemas.Item])
2 def get_all_items(name: Optional[str] = None, db: Session = Depends(get_db)):
3     """
4         Get all the Items stored in database
5     """
6     if name:
7         items = []
8         db_item = ItemRepo.fetch_by_name(db, name)
9         items.append(db_item)
10        return items
11    else:
12        return ItemRepo.fetch_all(db)
```

Main - Item

GET-Request (single item)



```
1 @app.get('/items/{item_id}', tags=["Item"], response_model=schemas.Item)
2 def get_item(item_id: int, db: Session = Depends(get_db)):
3     """
4         Get the Item with the given ID provided by User stored in database
5     """
6     db_item = ItemRepo.fetch_by_id(db, item_id)
7     if db_item is None:
8         raise HTTPException(status_code=404, detail="Item not found with the given ID")
9     return db_item
```

Main - Item

PUT-Request



```
1 @app.put('/items/{item_id}', tags=["Item"], response_model=schemas.Item)
2 async def update_item(item_id: int, item_request: schemas.Item, db: Session = Depends(get_db)):
3     """
4         Update an Item stored in the database
5     """
6     db_item = ItemRepo.fetch_by_id(db, item_id)
7     if db_item:
8         update_item_encoded = jsonable_encoder(item_request)
9         db_item.name = update_item_encoded['name']
10        db_item.price = update_item_encoded['price']
11        db_item.description = update_item_encoded['description']
12        db_item.store_id = update_item_encoded['store_id']
13        return await ItemRepo.update(db=db, item_data=db_item)
14    else:
15        raise HTTPException(status_code=400, detail="Item not found with the given ID")
```

Main - Item

DELETE-Request



```
1 @app.delete('/items/{item_id}', tags=["Item"])
2 async def delete_item(item_id: int, db: Session = Depends(get_db)):
3     """
4         Delete the Item with the given ID provided by User stored in database
5     """
6     db_item = ItemRepo.fetch_by_id(db, item_id)
7     if db_item is None:
8         raise HTTPException(status_code=404, detail="Item not found with the given ID")
9     await ItemRepo.delete(db, item_id)
10    return "Item deleted successfully!"
```

Main - Store

Now, it is your turn; turn in the exercises and fill out the missing "store" functions (they are similar to items, do not worry).

If you struggle, wait for my explanation or have a peek into the solutions.

Main - Store

GET-Request (all items)



```
1 @app.get('/stores', tags=["Store"], response_model=List[schemas.Store])
2 def get_all_stores(name: Optional[str] = None, db: Session = Depends(get_db)):
3     """
4         Get all the Stores stored in database
5     """
6     if name:
7         stores = []
8         db_store = StoreRepo.fetch_by_name(db, name)
9         print(db_store)
10        stores.append(db_store)
11    return stores
12 else:
13     return StoreRepo.fetch_all(db)
```

Main - Store

GET-Request (single items)



```
1 @app.get('/stores/{store_id}', tags=["Store"], response_model=schemas.Store)
2 def get_store(store_id: int, db: Session = Depends(get_db)):
3     """
4         Get the Store with the given ID provided by User stored in database
5     """
6     db_store = StoreRepo.fetch_by_id(db, store_id)
7     if db_store is None:
8         raise HTTPException(status_code=404, detail="Store not found with the given ID")
9     return db_store
```

Main - Store

POST-Request



```
1 @app.post('/stores', tags=["Store"], response_model=schemas.Store, status_code=201)
2 async def create_store(store_request: schemas.StoreCreate, db: Session = Depends(get_db)):
3     """
4     Create a Store and save it in the database
5     """
6     db_store = StoreRepo.fetch_by_name(db, name=store_request.name)
7     print(db_store)
8     if db_store:
9         raise HTTPException(status_code=400, detail="Store already exists!")
10
11    return await StoreRepo.create(db=db, store=store_request)
```

Main - Store

DELETE-Request



```
1 @app.delete('/stores/{store_id}', tags=["Store"])
2 async def delete_store(store_id: int, db: Session = Depends(get_db)):
3     """
4         Delete the Item with the given ID provided by User stored in database
5     """
6     db_store = StoreRepo.fetch_by_id(db, store_id)
7     if db_store is None:
8         raise HTTPException(status_code=404, detail="Store not found with the given ID")
9     await StoreRepo.delete(db, store_id)
10    return "Store deleted successfully!"
```

Thank you!

That was all for the moment

I love to discuss about Vue, Python and more

You can find me at [@dpinezich](https://twitter.com/dpinezich)

