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

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!

Starlette

The little ASGI framework that shines

Pydantic

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 httptools. 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.

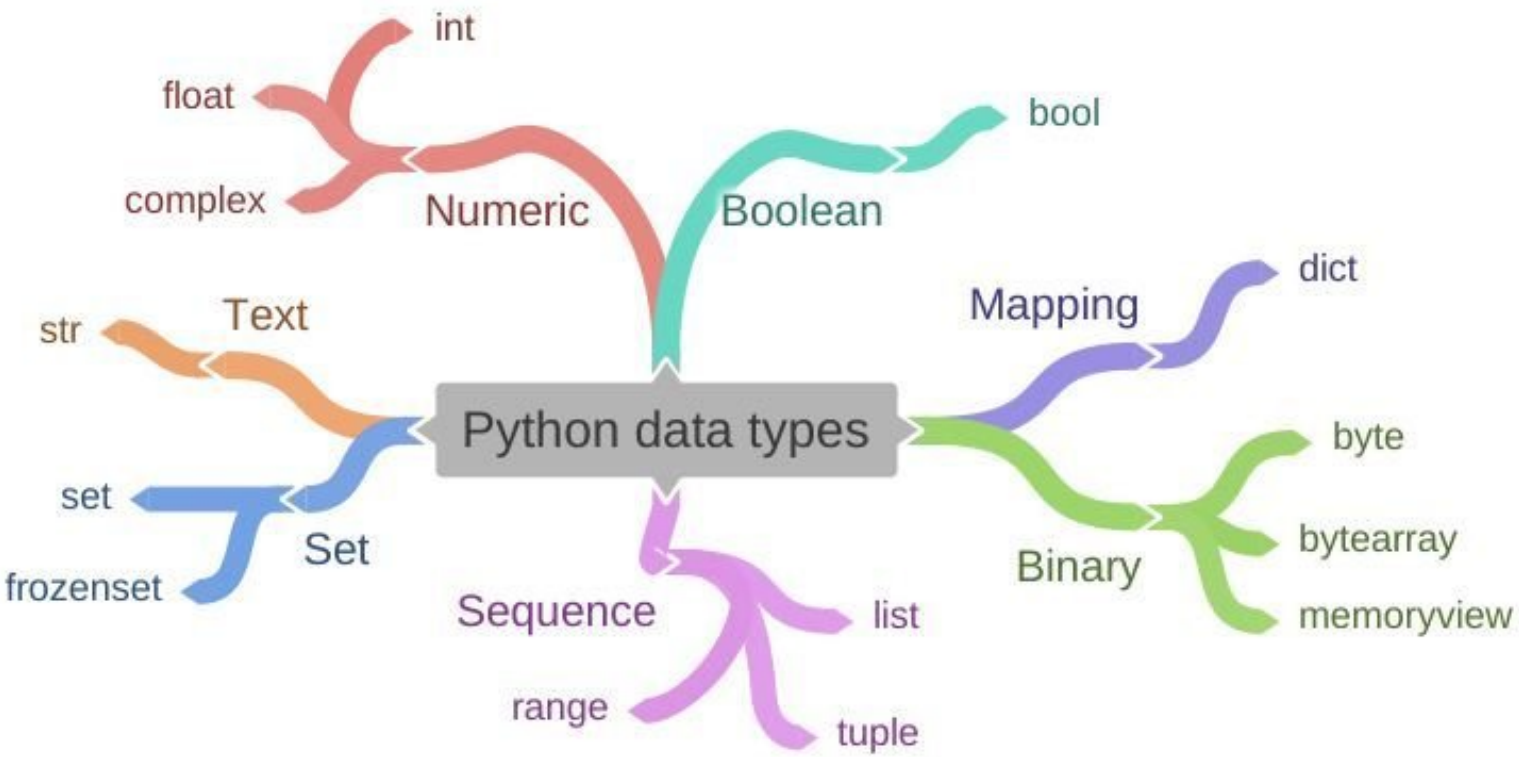
- Git: git@github.com:apigenio-GmbH/blazing-fast-fastapi.git
  - https://bit.ly/3DWDliR
- Zip: https://github.com/apigenio-GmbH/blazing-fast-fastapi/archive/refs/heads/main.zip
  - https://bit.ly/3fsTt15

3.

Python-CrashCourse

Let us talk some Python first before working with FastAPI

Basic Data Types in Python



https://cdn.hackernoon.com/images/B6I7WEwrKubf11jAWFL33iiMzR13-972831vv.jpeg

Numeric

```python
# 00_crashcourse/numeric.py

a = 5
print("Type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2 + 4j
print("\nType of c: ", type(c))
```

Text (Strings or Char-Seq)

```
# 00_crashcourse/strings.py

string1 = 'Welcome to the World'
print("String with the use of Single Quotes: ")
print(string1)

string2 = "I'm a Hero"
print("\nString with the use of Double Quotes: ")
print(string2)

string3 = '''I'm a Hero and I live in a world of "Heros"'''
print("\nString with the use of Triple Quotes: ")
print(string3)

string4 = '''Hero
            For
            Life'''
print("\nCreating a multiline String: ")
print(string4)
```

List

```python
# 00_crashcourse/lists.py

list = []
print("Initial blank List: ")
print(list)

list2 = ['HeroesAreEverywhere']
print("\nList with the use of String: ")
print(list2)

list3 = ["Hero", "For", "Heroes"]
print("\nList containing multiple values: ")
print(list3[0])
print(list3[2])

list4 = [['Hero', 'For'], ['Heroes']]
print("\nMulti-Dimensional List: ")
print(list4)
```

Tuple

```python
# 00_crashcourse/tuples.py

tuple1 = ()
print("Initial empty Tuple: ")
print(tuple1)

tuple2 = ('Heroes', 'For')
print("\nTuple with the use of String: ")
print(tuple2)

list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))

tuple3 = tuple('Heroes')
print("\nTuple with the use of function: ")
print(tuple3)

tuple4 = (0, 1, 2, 3)
tuple5 = ('python', 'hero')
tuple6 = (tuple4, tuple5)
print("\nTuple with nested tuples: ")
print(tuple6)
```

Question

What is the difference between a list and a tuple?

Lists are mutable - tuples are immutable.

Set

```python
# 00_crashcourse/sets.py

set1 = set()
print("Initial blank Set: ")
print(set1)

set2 = set("HeroForHeroes")
print("\nSet with the use of String: ")
print(set2)

set3 = set(["Hero", "For", "Heroes"])
print("\nSet with the use of List: ")
print(set3)

set4 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
print("\nSet with the use of Mixed Values")
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

```python
# 00_crashcourse/dictionaries.py

dict1 = {}
print("Empty Dictionary: ")
print(dict1)

dict2 = {1: 'Hero', 2: 'For', 3: 'Heroes'}
print("\nDictionary with the use of Integer Keys: ")
print(dict2)

dict3 = {'Name': 'Hero', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(dict3)

dict4 = dict({1: 'Hero', 2: 'For', 3: 'Heroes'})
print("\nDictionary with the use of dict(): ")
print(dict4)

dict5 = dict([(1, 'Heroes'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
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

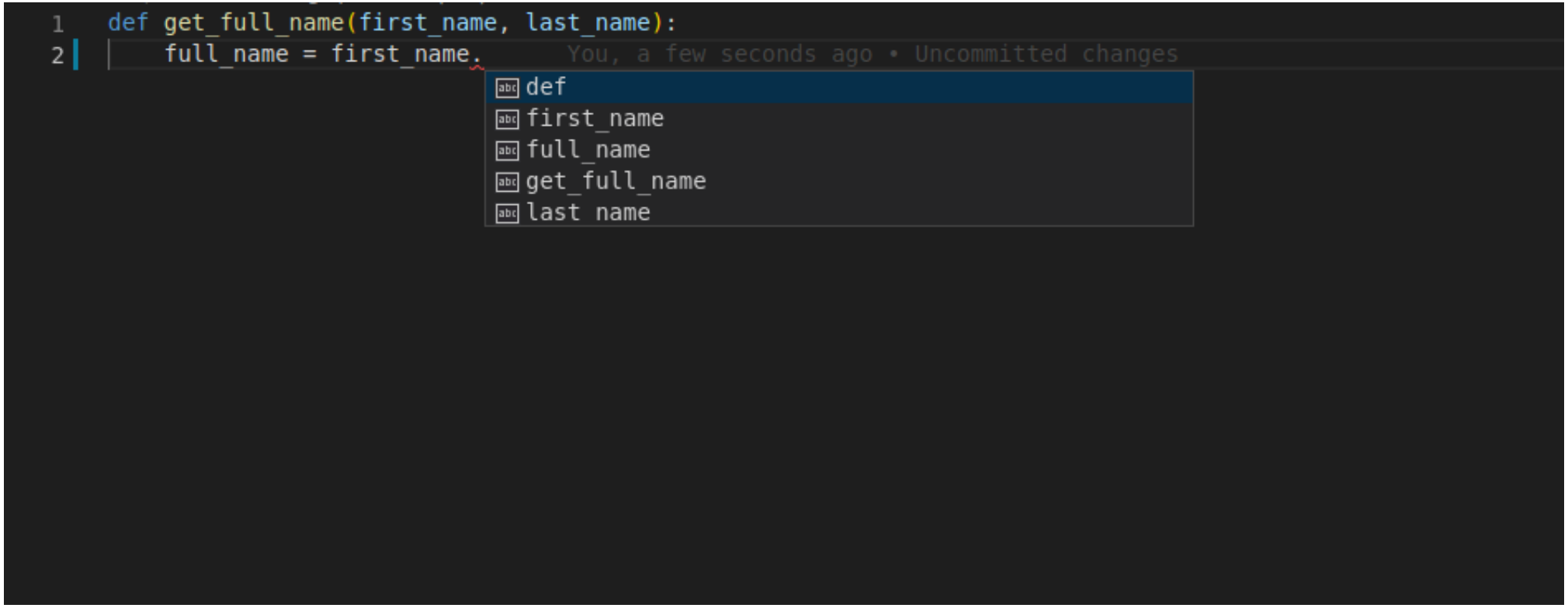
```python
def get_full_name(first_name, last_name):
    full_name = first_name.title() + " " + last_name.title()
    return full_name


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

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:



https://fastapi.tiangolo.com/python-types/

Motivation for Typing

Let us add some typing:

```python
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"))
```

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

Motivation for Typing

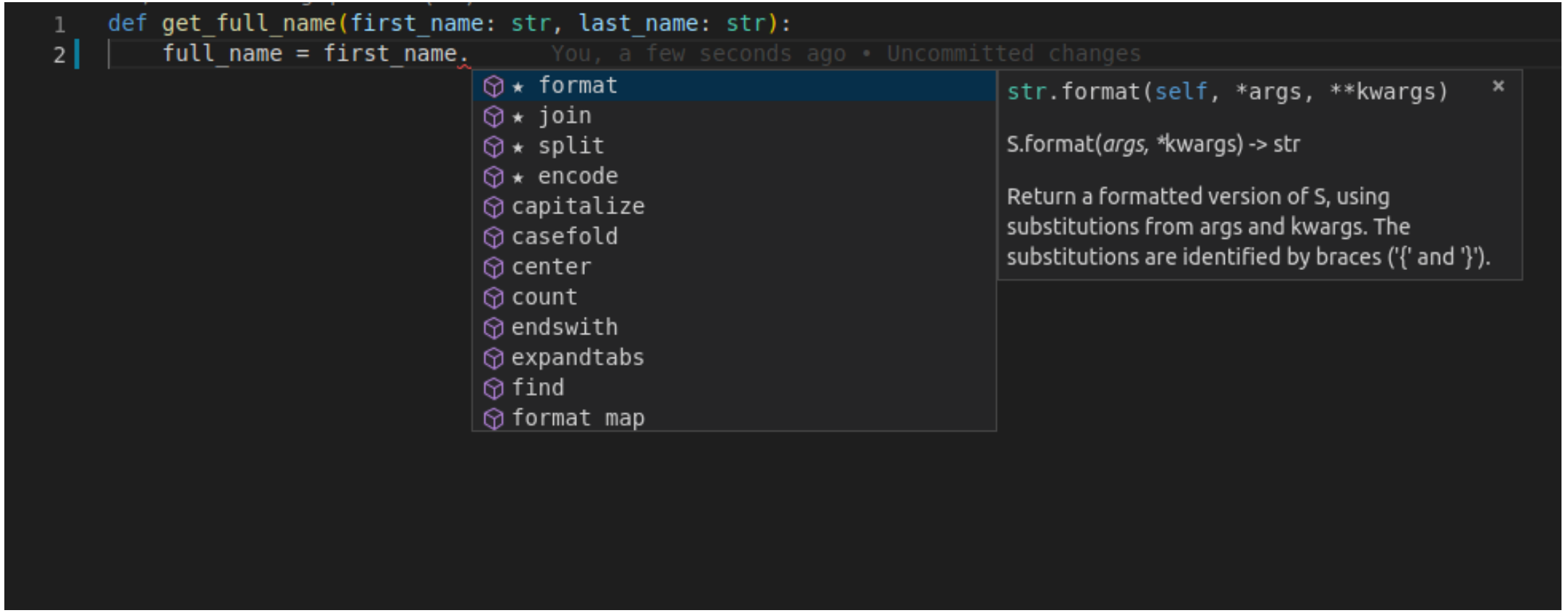
Careful, it is not the same as:

```
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

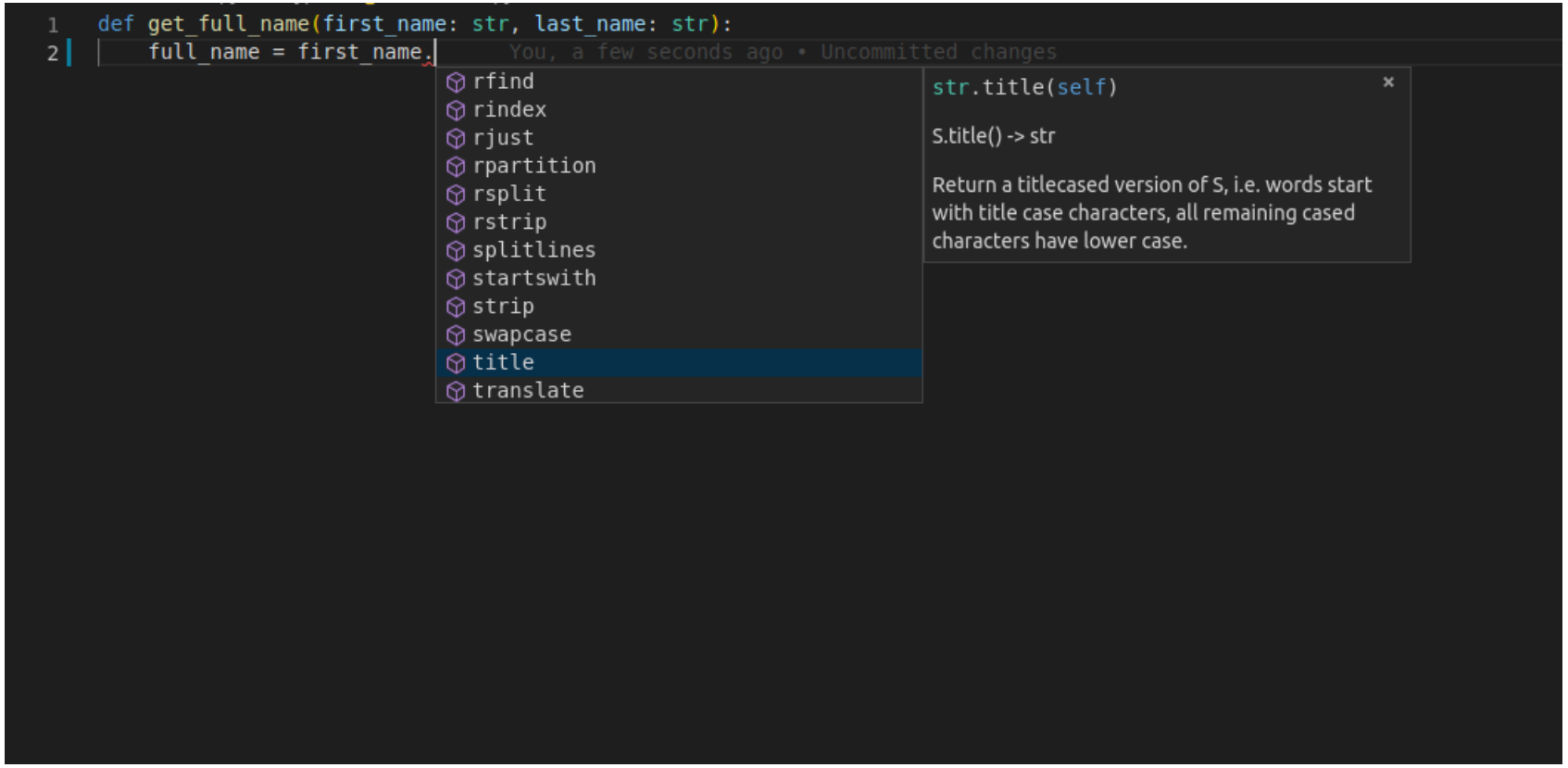
Now, the IDE magic has returned:



https://fastapi.tiangolo.com/python-types/

Motivation for Types

Now, the IDE magic has returned - title appears!



https://fastapi.tiangolo.com/python-types/

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
```

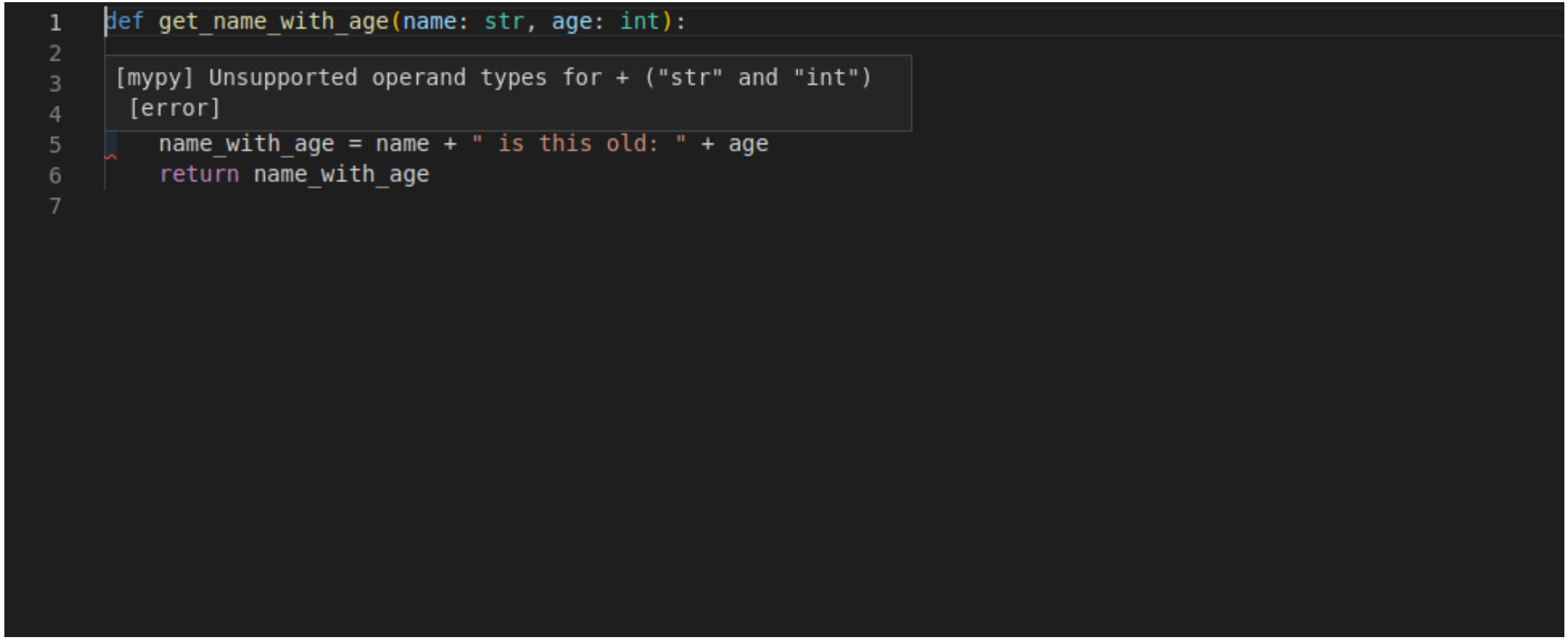
Can you spot the error?

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



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+

```python
from typing import List


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

Python 3.9+

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

```python
from typing import Dict
def process_items(prices: Dict[str, float]):
    for item_name, item_price in prices.items():
        print(item_name)
        print(item_price)
```

Python 3.9+

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

Declaring Types (Union)

Python 3.6+

```python
from typing import Union

def process_item(item: Union[int, str]):
    print(item)
```

Python 3.10+

```python
def process_item(item: int | str):
    print(item)
```

Declaring Types (Optional)

Python 3.10+

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

Declaring Types (Optional)

Python 3.10+

```python
from typing import Optional


def say_hi(name: Optional[str] = None):
    if name is not None:
        print(f"Hey {name}!")
    else:
        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+

```python
from datetime import datetime
from typing import List, Union
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name = "John Doe"
    signup_ts: Union[datetime, None] = None
    friends: List[int] = []


external_data = {
    "id": "123",
    "signup_ts": "2017-06-01 12:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data)
print(user)
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
print(user.id)
# > 123
```

Pydantic models

Python 3.9+

```python
from datetime import datetime
from typing import Union
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name = "John Doe"
    signup_ts: Union[datetime, None] = None
    friends: list[int] = []


external_data = {
    "id": "123",
    "signup_ts": "2017-06-01 12:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data)
print(user)
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
print(user.id)
# > 123
```

Pydantic models

Python 3.10+

```python
from datetime import datetime
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name = "John Doe"
    signup_ts: datetime | None = None
    friends: list[int] = []


external_data = {
    "id": "123",
    "signup_ts": "2017-06-01 12:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data)
print(user)
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
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.

```python
# 00_crashcourse/args.py

def adder(*num):
    sum = 0
    for n in num:
        sum = sum + n
    print("Sum:", sum)

adder(3, 5)
adder(4, 5, 6, 7)
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.

```
# 00_crashcourse/kwargs.py

def intro(**data):
    print("\nData type of argument:",type(data))

    for key, value in data.items():
        print("{} is {}".format(key,value))

intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)
intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com",
      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.

4.

Getting started

Basics are done, let us now focus on FastAPI

Hello World

Let us focus on FastAPI now!

Hello World

Hello World!

```python
# 02_getting_started/helloworld.py

from fastapi import FastAPI

app = FastAPI()

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

Run it with:

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

Hello World

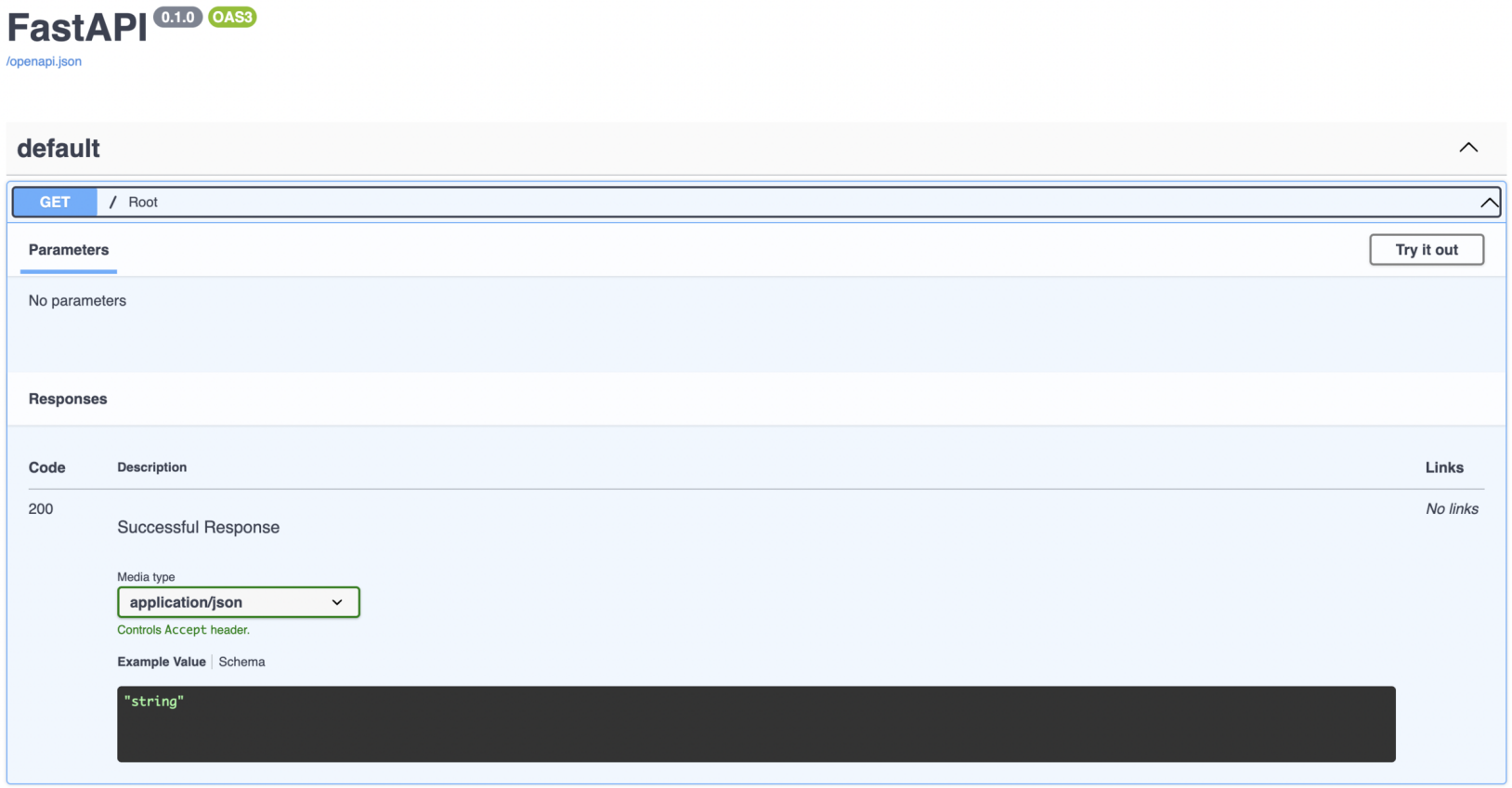
Stack-Trace:

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

Our hello world is now happily running on 127.0.0.1:8000

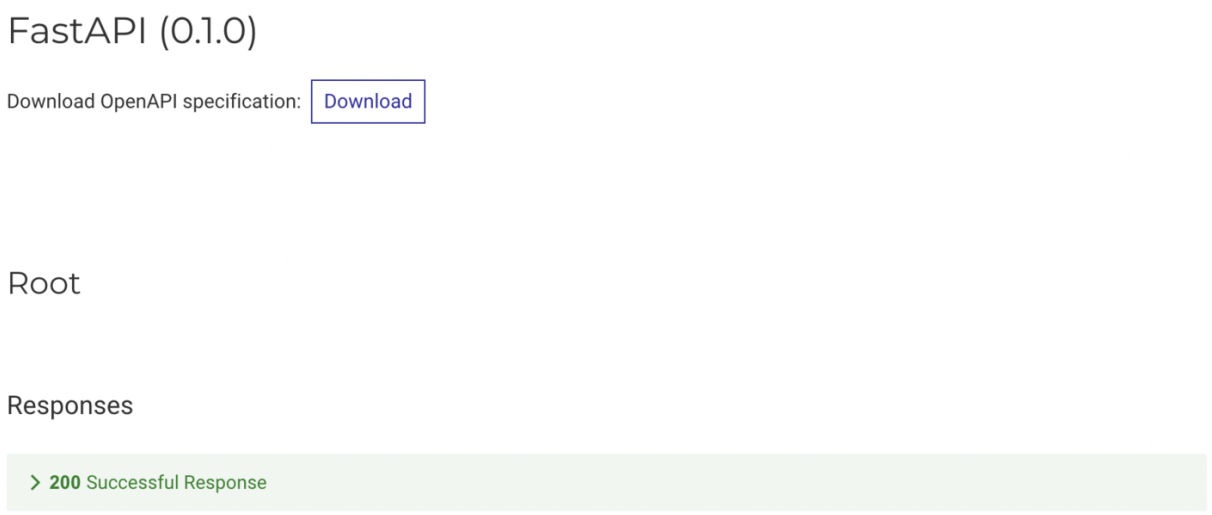
Hello World

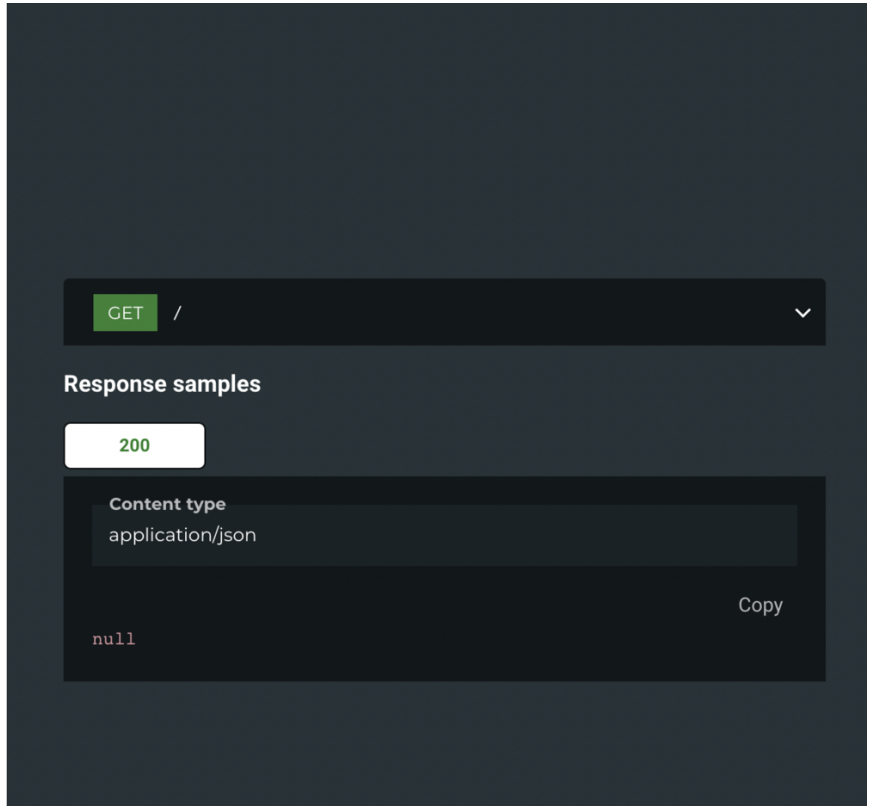
Interactive Tooling: http://127.0.0.1:8000/docs powered by SwaggerUI



Hello World (alternative)

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





Hello World (OpenAPI)

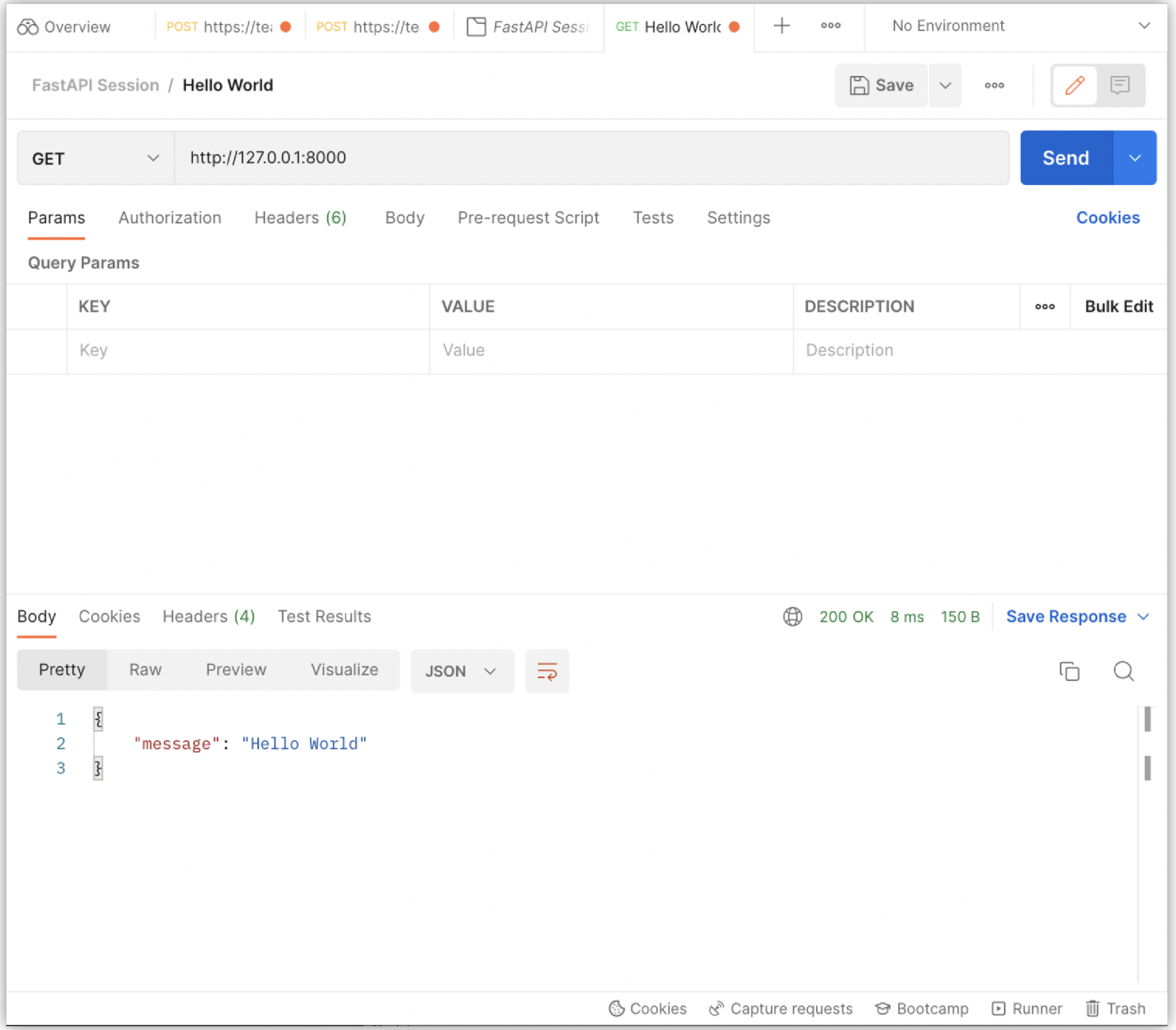
OpenAPI:

http://127.0.0.1:8000/openapi.json


```
{
  "openapi": "3.0.2",
  "info": {
    "title": "FastAPI",
    "version": "0.1.0"
  },
  "paths": {
    "/": {
      "get": {
        "summary": "Root",
        "operationId": "root__get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {

                }
              }
            }
          }
        }
      }
```

Hello World

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



Hello World

Hello World!

```python
# 02_getting_started/helloworld.py

from fastapi import FastAPI

app = FastAPI()

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

Run it with:

```
cd examples/02_getting_started
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`

```python
from fastapi import FastAPI

app = FastAPI()


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

The five steps

2. create a `FastAPI` "instance"

```python
from fastapi import FastAPI

app = FastAPI()


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

The five steps

2. create a `FastAPI` "instance"

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
async def root():
    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

```python
from fastapi import FastAPI

app = FastAPI()


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

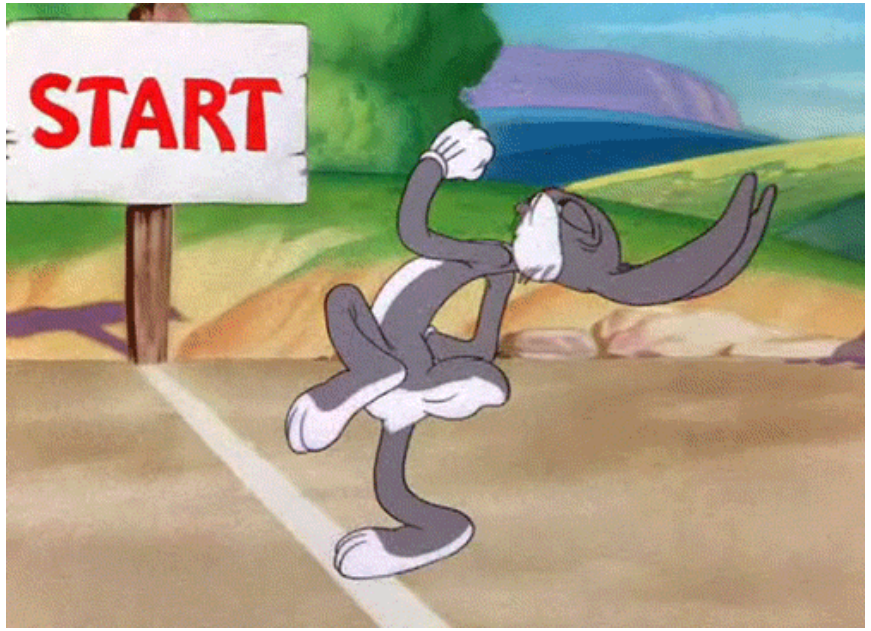
The five steps

5. Return the content

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
async def root():
    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!

```python
# 02_getting_started/withtypes.py

from fastapi import FastAPI
application = FastAPI()

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

How do you run it?

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

Items and their types

What happens if you try:

```
# 02_getting_started/helloworld.py

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

A well documented error message:

```
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

Order matters

If you have multiple routes, be careful!

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

```python
# 02_getting_started/order.py

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}
```

Predefined values

Look at this enum-rich example

```python
# 02_getting_started/enumrich.py

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"}
```

It gives you all the hints needed

Predefined values

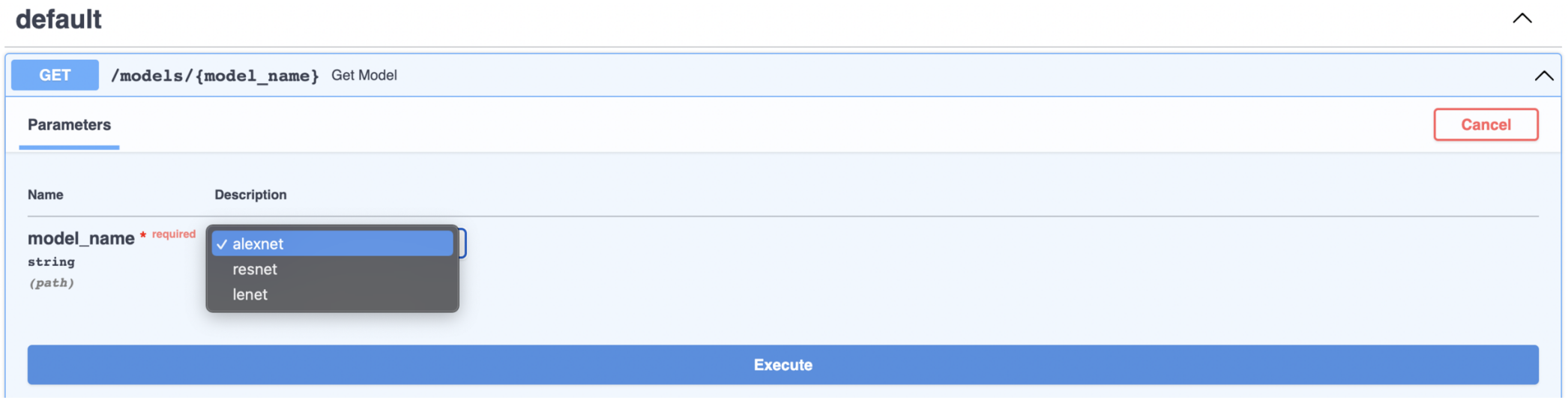
But we can do it even better:

```python
# 02_getting_started/enumrich_better.py

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"}
```

Predefined values

Let us quickly check the docs to see what happened:

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



Query Parameters

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

```python
# 02_getting_started/query.py
#
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]
```

```
uvicorn query:app --reload
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+

```python
# 02_getting_started/query_optional.py
from typing import Union
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: str, q: Union[str, None] = None):
    if q:
        return {"item_id": item_id, "q": q}
    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+

```python
# 02_getting_started/query_optional_3_10.py

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}
```

Multiple path and query parameters

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

```python
# 02_getting_started/multiple_query.py

from typing import Union
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: Union[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
```

Required query parameters

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

```python
# 02_getting_started/required.py

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
```

Required query parameters

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

```
// 20220927223512
// http://127.0.0.1:8000/items/foo-item

{
  "detail": [
    {
      "loc": [
        "query",
        "needy"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

Required query parameters

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

Required query parameters

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

```python
# 02_getting_started/not_required.py

from fastapi import FastAPI
app = FastAPI()

@app.get("/items/{item_id}")
async def read_user_item(item_id: str, needy: str = None):
    item = {"item_id": item_id, "needy": needy}
    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.

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:

```python
# 02_getting_started/base_model.py

from typing import Union
from fastapi import FastAPI
from pydantic import BaseModel

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

app = FastAPI()

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

BaseModel

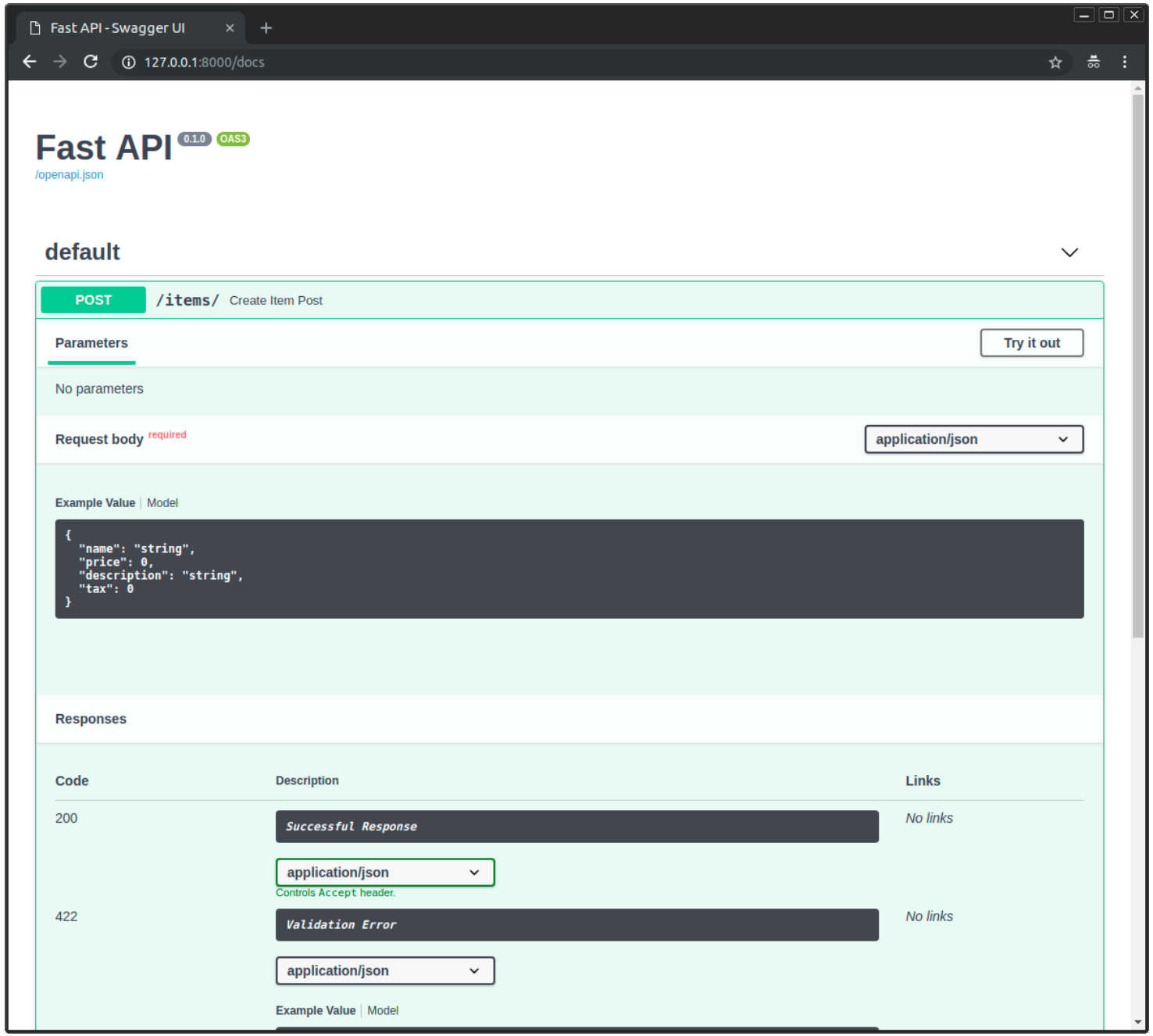
For example, this model above declares a JSON "object" (or Python dict) like:

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

or... (because of optionals)

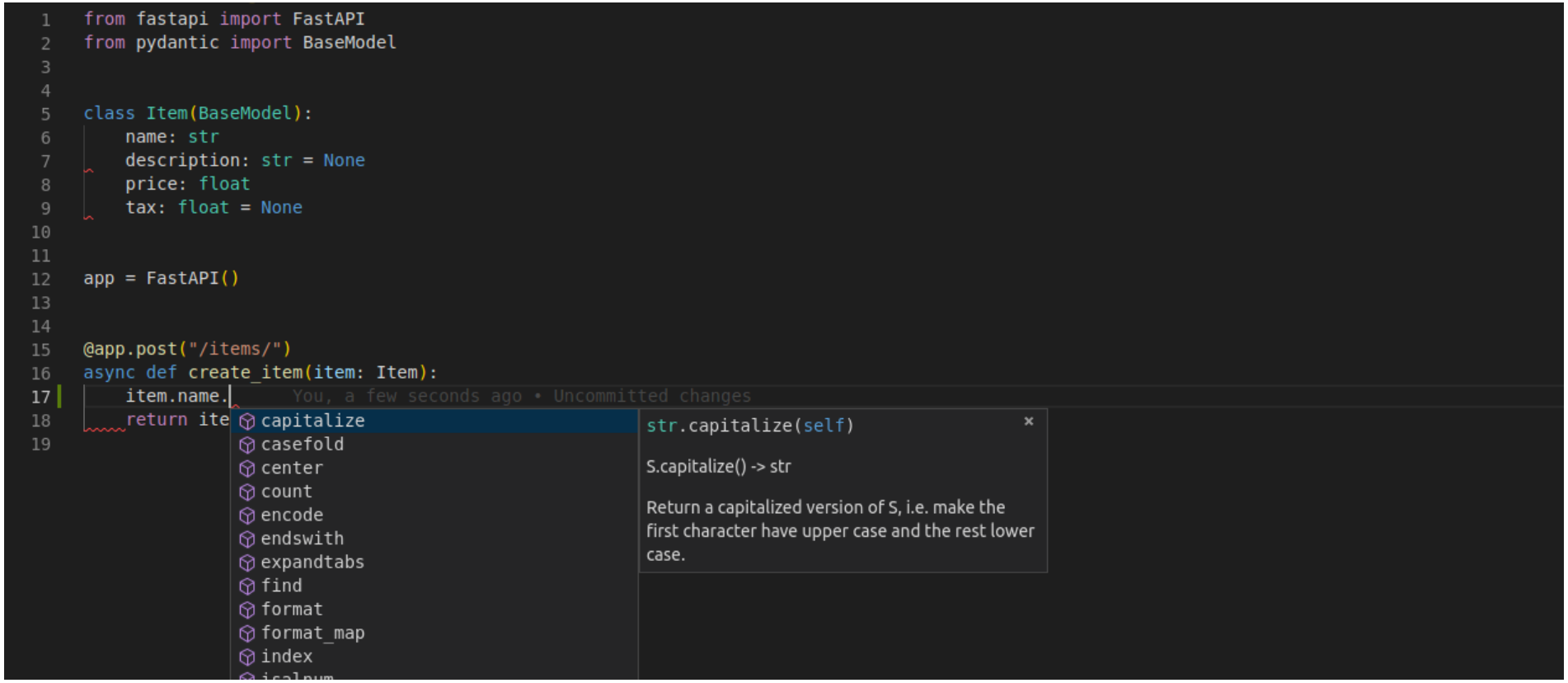
```
{
    "name": "Foo",
    "price": 45.2
}
```

BaseModel


Fast API - Swagger UI
127.0.0.1:8000/docs
Fast API
0.1.0
OAS3
/openapi.json
default
POST
/items/
Create Item Post
Parameters
Try it out
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
No links
application/json
Controls Accept header.
422
Validation Error
No links
application/json
Example Value
Model

BaseModel

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



BaseModel

Now let us use the base model:

```python
# 02_getting_started/use_base_model.py

from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

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

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    item_dict = item.dict()
    if item.tax:
        price_with_tax = item.price + item.tax
        item_dict.update({"price_with_tax": price_with_tax})
    return item_dict
```

Path & Body

Now a combination of request body and path parameters

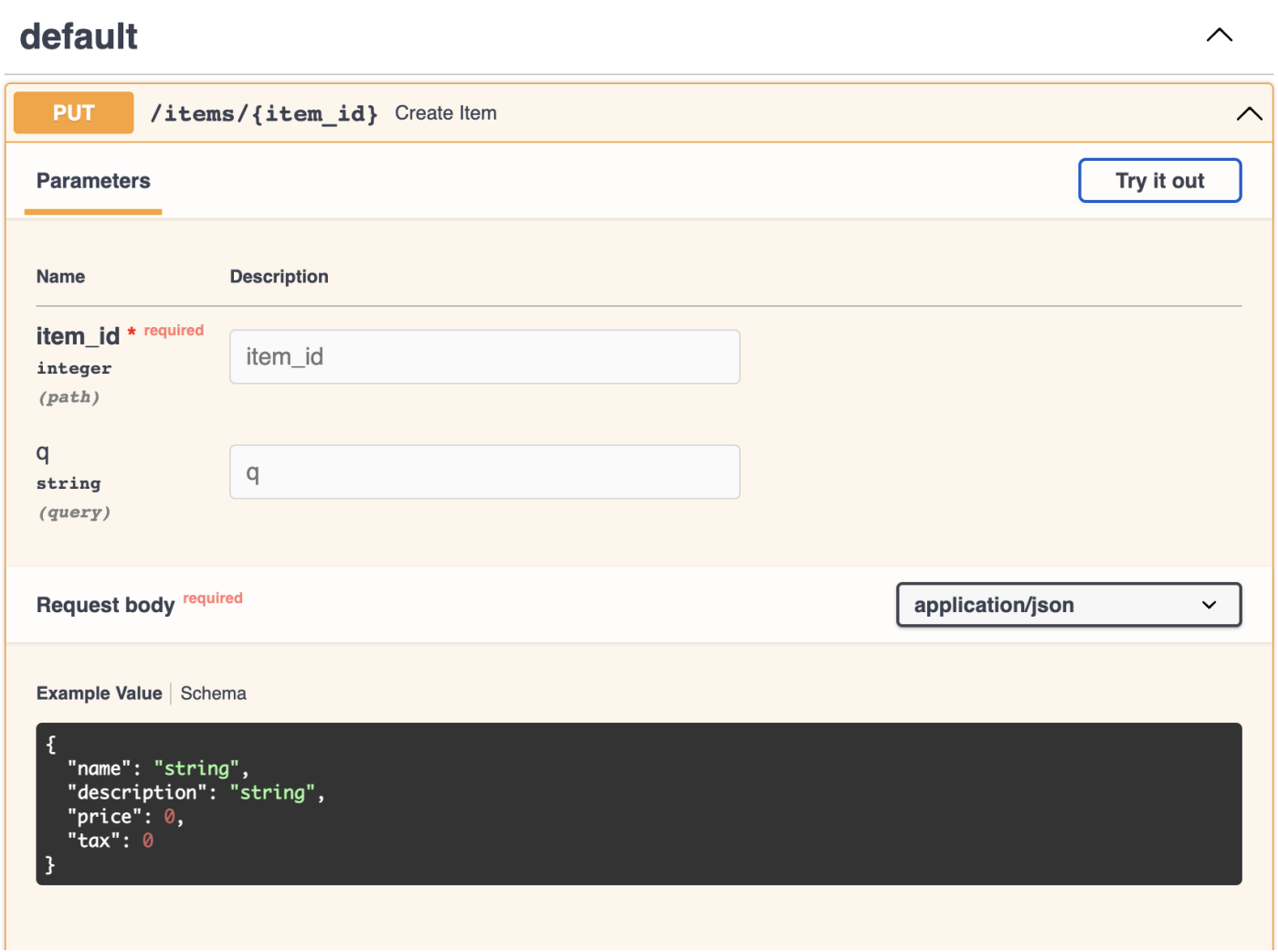
```python
# 02_getting_started/request_body_path.py

from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

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

app = FastAPI()

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

Path & Body & Query

Now a combination of request body, path parameters and query

```python
# 02_getting_started/request_body_path_query.py

from typing import Union

from fastapi import FastAPI
from pydantic import BaseModel

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

app = FastAPI()

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

Path & Body & Query



Mix Path, Query and body parameters

```python
# 02_getting_started/multiple_parameters.py

from typing import Union
from fastapi import FastAPI, Path
from pydantic import BaseModel
app = FastAPI()

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

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

Multiple Parameters

```python
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
```

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.

```python
# 02_getting_started/body_fields.py

from typing import Union
from fastapi import Body, FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

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

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

Body - Fields

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

```python
# 02_getting_started/body_fields.py

from typing import Union
from fastapi import Body, FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

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

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

Body - Nested Models

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

```python
# 02_getting_started/nested_models.py

from typing import 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 = []

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

```python
# 02_getting_started/typed_nested_models.py

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
```

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

```python
# 02_getting_started/extra_data_types.py

from datetime import datetime, time, timedelta
from typing import Union
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: Union[datetime, None] = Body(default=None),
    end_datetime: Union[datetime, None] = Body(default=None),
    repeat_at: Union[time, None] = Body(default=None),
    process_after: Union[timedelta, None] = Body(default=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,
        "repeat_at": repeat_at,
        "process_after": process_after,
        "start_process": start_process,
        "duration": duration,
    }
```

Cookies & Header

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

```python
from typing import Union
from fastapi import Cookie, FastAPI

app = FastAPI()

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

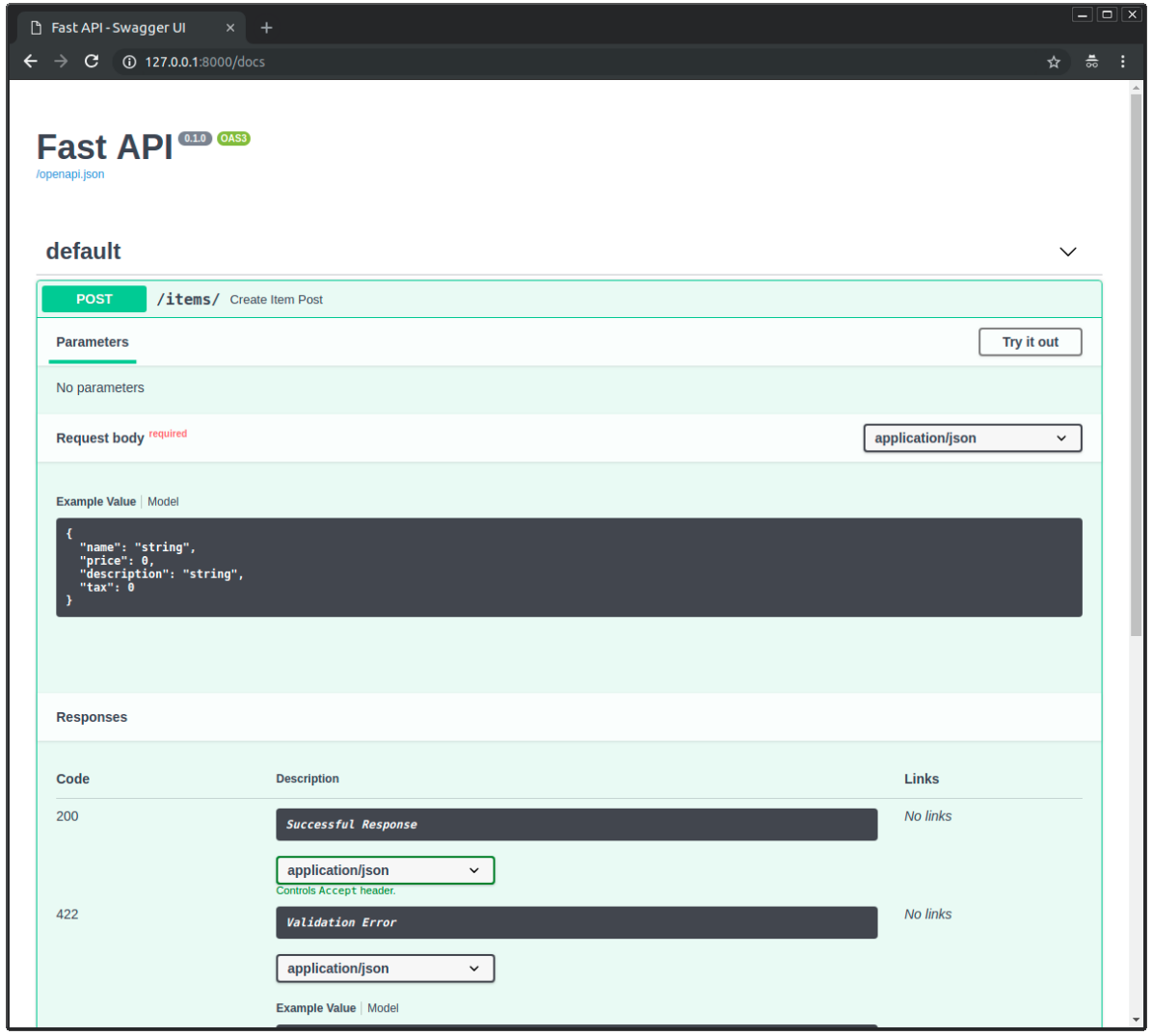
```python
from typing import Union
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")
async def read_items(user_agent: Union[str, None] = Header(default=None)):
    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:

```python
from fastapi import FastAPI

app = FastAPI()

@app.post("/items/", status_code=201)
async def create_item(name: str):
    return {"name": name}
```

Response Status Code



Form Data

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

```python
from fastapi import FastAPI, Form

app = FastAPI()

@app.post("/login/")
async def login(username: str = Form(), password: str = Form()):
    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

```
cat = Animal(
    name="Cat",
    description="This is a small cat",
    height="0.2",
    weight=3.5,
    abilities=["fast", "cute", "small"]
)
```

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`.

```python
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]}
```

Custom Errors

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

```python
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}
```

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...

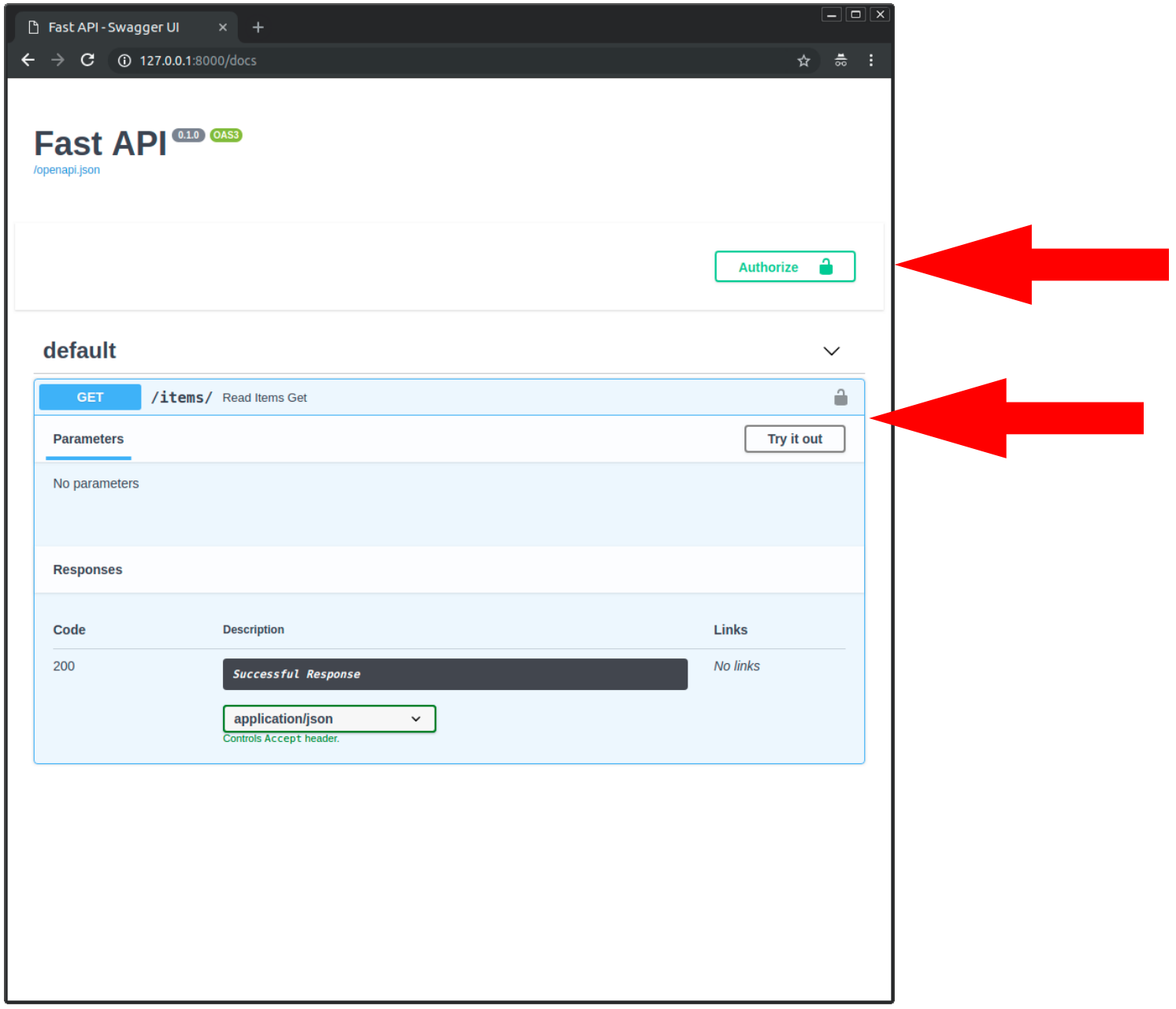
Security

Here a small example:

```python
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: str = Depends(oauth2_scheme)):
    return {"token": token}
```

Easy right?

Security


Fast API - Swagger UI
127.0.0.1:8000/docs
Fast API
0.1.0
OAS3
/openapi.json
Authorize
default
GET
/items/
Read Items Get
Parameters
Try it out
No parameters
Responses
Code
Description
Links
200
Successful Response
No links
application/json
Controls Accept header.

Security

```python
from typing import Union

from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer
from pydantic import BaseModel

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):
    username: str
    email: Union[str, None] = None
    full_name: Union[str, None] = None
    disabled: Union[bool, None] = None


def fake_decode_token(token):
    return User(
        username=token + "fakedecoded", email="john@example.com", full_name="John Doe"
    )

async def get_current_user(token: str = Depends(oauth2_scheme)):
    user = fake_decode_token(token)
    return user

@app.get("/users/me")
async def read_users_me(current_user: User = Depends(get_current_user)):
    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.

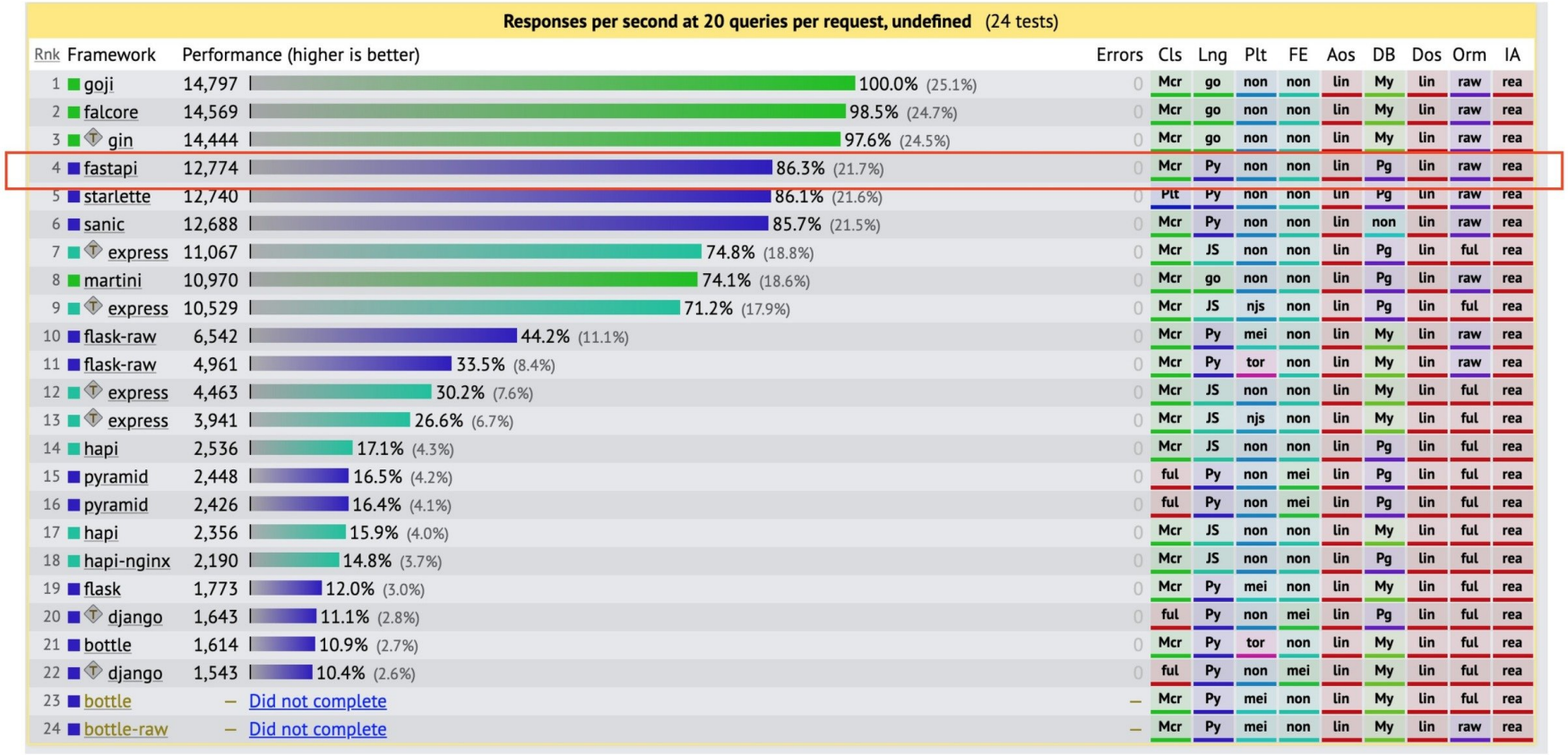
Benchmarks

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

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

Benchmarks

Is it really comparable to NodeJS or Go? Yes!



Responses per second at 20 queries per request, undefined (24 tests)

Rnk	Framework	Performance (higher is better)	Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1	goji	14,797 100.0% (25.1%)	0	Mcr	go	non	non	lin	My	lin	raw	rea
2	falcore	14,569 98.5% (24.7%)	0	Mcr	go	non	non	lin	My	lin	raw	rea
3	gin	14,444 97.6% (24.5%)	0	Mcr	go	non	non	lin	My	lin	raw	rea
4	fastapi	12,774 86.3% (21.7%)	0	Mcr	Py	non	non	lin	Pg	lin	raw	rea
5	starlette	12,740 86.1% (21.6%)	0	Plt	Py	non	non	lin	Pg	lin	raw	rea
6	sanic	12,688 85.7% (21.5%)	0	Mcr	Py	non	non	lin	non	lin	raw	rea
7	express	11,067 74.8% (18.8%)	0	Mcr	JS	non	non	lin	Pg	lin	ful	rea
8	martini	10,970 74.1% (18.6%)	0	Mcr	go	non	non	lin	Pg	lin	raw	rea
9	express	10,529 71.2% (17.9%)	0	Mcr	JS	njs	non	lin	Pg	lin	ful	rea
10	flask-raw	6,542 44.2% (11.1%)	0	Mcr	Py	mei	non	lin	My	lin	raw	rea
11	flask-raw	4,961 33.5% (8.4%)	0	Mcr	Py	tor	non	lin	My	lin	raw	rea
12	express	4,463 30.2% (7.6%)	0	Mcr	JS	non	non	lin	My	lin	ful	rea
13	express	3,941 26.6% (6.7%)	0	Mcr	JS	njs	non	lin	My	lin	ful	rea
14	hapi	2,536 17.1% (4.3%)	0	Mcr	JS	non	non	lin	Pg	lin	ful	rea
15	pyramid	2,448 16.5% (4.2%)	0	ful	Py	non	mei	lin	Pg	lin	ful	rea
16	pyramid	2,426 16.4% (4.1%)	0	ful	Py	non	mei	lin	Pg	lin	ful	rea
17	hapi	2,356 15.9% (4.0%)	0	Mcr	JS	non	non	lin	My	lin	ful	rea
18	hapi-nginx	2,190 14.8% (3.7%)	0	Mcr	JS	non	non	lin	Pg	lin	ful	rea
19	flask	1,773 12.0% (3.0%)	0	Mcr	Py	mei	non	lin	My	lin	ful	rea
20	django	1,643 11.1% (2.8%)	0	ful	Py	non	mei	lin	Pg	lin	ful	rea
21	bottle	1,614 10.9% (2.7%)	0	Mcr	Py	tor	non	lin	My	lin	ful	rea
22	django	1,543 10.4% (2.6%)	0	ful	Py	non	mei	lin	My	lin	ful	rea
23	bottle	– Did not complete	–	Mcr	Py	mei	non	lin	My	lin	ful	rea
24	bottle-raw	– Did not complete	–	Mcr	Py	mei	non	lin	My	lin	raw	rea

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

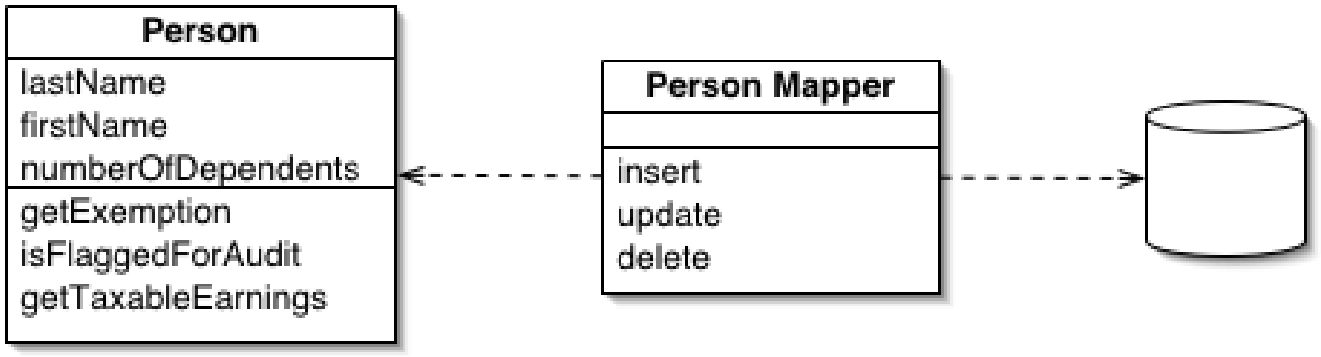
SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

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

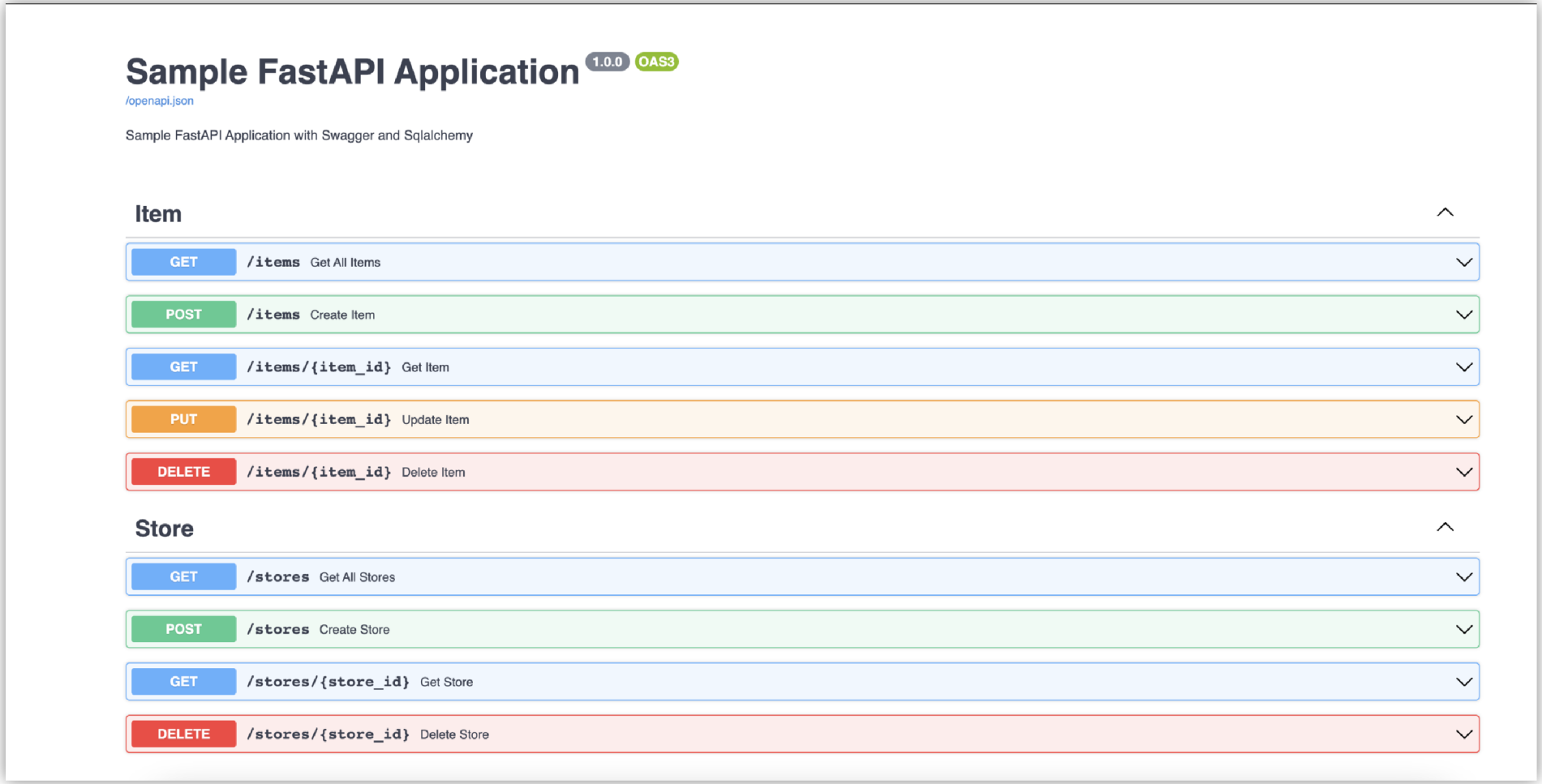
Data Mapper Pattern

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



Prepare Database

Have a look at the db.py file.

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./data.db"


engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False},echo=True
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()


# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        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 used to create independent database session for each request.

Create the model

Have a look at models.py:

```python
from sqlalchemy import Column, ForeignKey, Integer, String, Float
from sqlalchemy.orm import relationship
from db import Base

class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True,index=True)
    name = Column(String(80), nullable=False, unique=True,index=True)
    price = Column(Float(precision=2), nullable=False)
    description = Column(String(200))
    store_id = Column(Integer,ForeignKey('stores.id'),nullable=False)
    def __repr__(self):
        return 'ItemModel(name=%s, price=%s,store_id=%s)' % (self.name, self.price,self.store_id)

class Store(Base):
    __tablename__ = "stores"
    id = Column(Integer, primary_key=True,index=True)
    name = Column(String(80), nullable=False, unique=True)
    items = relationship("Item",primaryjoin="Store.id == Item.store_id",cascade="all, delete-orphan")

    def __repr__(self):
        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:

```python
class ItemBase(BaseModel):
    name: str
    price: float
    description: Optional[str] = None
    store_id: int

class ItemCreate(ItemBase):
    pass

class Item(ItemBase):
    id: int
    class Config:
        orm_mode = True
```

Create the schema

```python
class ItemBase(BaseModel):
    name: str
    price: float
    description: Optional[str] = None
    store_id: int

class ItemCreate(ItemBase):
    pass

class Item(ItemBase):
    id: int
    class Config:
        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

```python
class ItemBase(BaseModel):
    name: str
    price: float
    description: Optional[str] = None
    store_id: int

class ItemCreate(ItemBase):
    pass

class Item(ItemBase):
    id: int
    class Config:
        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

```python
class ItemBase(BaseModel):
    name: str
    price: float
    description: Optional[str] = None
    store_id: int

class ItemCreate(ItemBase):
    pass

class Item(ItemBase):
    id: int
    class Config:
        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:

```
# Some configuration added
app = FastAPI(title="Sample FastAPI Application for Demo",
              description="Nice to show how easy an API is created!",
              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:

```python
@app.exception_handler(Exception)
def validation_exception_handler(request, err):
    base_error_message = f"Failed to execute: {request.method}: {request.url}"
    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!

```python
@app.middleware("http")
async def add_process_time_header(request, call_next):
    print('inside middleware!')
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(f'{process_time:0.4f} sec')
    return response
```

Adds a processing time for the request.

Main - Item

POST-Request

```python
@app.post('/items', tags=["Item"], response_model=schemas.Item, status_code=201)
async def create_item(item_request: schemas.ItemCreate, db: Session = Depends(get_db)):
    """
    Create an Item and store it in the database
    """
    db_item = ItemRepo.fetch_by_name(db, name=item_request.name)
    if db_item:
        raise HTTPException(status_code=400, detail="Item already exists!")

    return await ItemRepo.create(db=db, item=item_request)
```

Main - Item

GET-Request (all items)

```python
@app.get('/items', tags=["Item"], response_model=List[schemas.Item])
def get_all_items(name: Optional[str] = None, db: Session = Depends(get_db)):
    """
    Get all the Items stored in database
    """
    if name:
        items = []
        db_item = ItemRepo.fetch_by_name(db, name)
        items.append(db_item)
        return items
    else:
        return ItemRepo.fetch_all(db)
```

Main - Item

GET-Request (single item)

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

Main - Item

PUT-Request

```python
@app.put('/items/{item_id}', tags=["Item"], response_model=schemas.Item)
async def update_item(item_id: int, item_request: schemas.Item, db: Session = Depends(get_db)):
    """
    Update an Item stored in the database
    """
    db_item = ItemRepo.fetch_by_id(db, item_id)
    if db_item:
        update_item_encoded = jsonable_encoder(item_request)
        db_item.name = update_item_encoded['name']
        db_item.price = update_item_encoded['price']
        db_item.description = update_item_encoded['description']
        db_item.store_id = update_item_encoded['store_id']
        return await ItemRepo.update(db=db, item_data=db_item)
    else:
        raise HTTPException(status_code=400, detail="Item not found with the given ID")
```

Main - Item

DELETE-Request

```python
@app.delete('/items/{item_id}', tags=["Item"])
async def delete_item(item_id: int, db: Session = Depends(get_db)):
    """
    Delete the Item with the given ID provided by User stored in database
    """
    db_item = ItemRepo.fetch_by_id(db, item_id)
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found with the given ID")
    await ItemRepo.delete(db, item_id)
    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)

```python
@app.get('/stores', tags=["Store"], response_model=List[schemas.Store])
def get_all_stores(name: Optional[str] = None, db: Session = Depends(get_db)):
    """
    Get all the Stores stored in database
    """
    if name:
        stores = []
        db_store = StoreRepo.fetch_by_name(db, name)
        print(db_store)
        stores.append(db_store)
        return stores
    else:
        return StoreRepo.fetch_all(db)
```

Main - Store

GET-Request (single items)

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

Main - Store

POST-Request

```python
@app.post('/stores', tags=["Store"], response_model=schemas.Store, status_code=201)
async def create_store(store_request: schemas.StoreCreate, db: Session = Depends(get_db)):
    """
    Create a Store and save it in the database
    """
    db_store = StoreRepo.fetch_by_name(db, name=store_request.name)
    print(db_store)
    if db_store:
        raise HTTPException(status_code=400, detail="Store already exists!")

    return await StoreRepo.create(db=db, store=store_request)
```

Main - Store

DELETE-Request

```python
@app.delete('/stores/{store_id}', tags=["Store"])
async def delete_store(store_id: int, db: Session = Depends(get_db)):
    """
    Delete the Item with the given ID provided by User stored in database
    """
    db_store = StoreRepo.fetch_by_id(db, store_id)
    if db_store is None:
        raise HTTPException(status_code=404, detail="Store not found with the given ID")
    await StoreRepo.delete(db, store_id)
    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