Пишем простое API: эндпоинты, параметры, валидация

Итак, постепенно повторим всё то, что мы уже умеем делать с помощью Flask, только на этот раз используя FastAPI. А именно:

- Напишем эндпоинт, принимающий GET-запрос с парочкой query-параметров. В ответ он вернёт какой-нибудь json.
- Напишем эндпоинт, который принимает POST-запрос. Сделаем валидацию входящих параметров.
- Посмотрим на некоторые особенности фреймворка, которых нет во Flask.

И как всегда, прежде всего установим парочку библиотек.

```
pip install -r requirements.txt
```

```
fastapi==0.70.0
uvicorn[standard]==0.15.0
```

FastAPI, в отличие от Flask, не поставляется со встроенным WSGI-сервером. Так что второе приложение — это именно оно. Только это не WSGI-сервер, а ASGI, наследник уже известного нам. Ссылку на подробности смотрите под видео.

И создадим файл main, где будем вести разработку.

И сразу же изобразим простой хеллоу ворлд:

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/")
async def root() -> Dict[str, str]:
   return {"message": "Hello World"}
```

По одному взгляду сразу станет понятно, что тут происходит: во Flask то же самое, даже чуточку сложнее. Созданный объект приложения имеет ряд

методов, которые отвечают за тот или иной HTTP-метод. Ими мы декорируем наши функции, в результате получаем эндпоинты. Обратим внимание, что функции в данном случае — корутины. Мы пишем асинхронный код. Функция возвращает словарик. Давайте теперь запустим этот код.

Для этого в терминале напишем следующее:

```
(venv) ➜  lesson_1 git:(master) ✗ uvicorn main:app --reload
INFO:     Will watch for changes in these directories:
['/Users/andrei/PycharmProjects/python_advanced/module_26_fastapi/
materials/lesson_1']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C
to quit)
INFO:     Started reloader process [79166] using watchgod
INFO:     Started server process [79168]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:63422 - "GET / HTTP/1.1" 200 OK
```

Консольная утилита uvicorn принимает обязательный аргумент — путь до модуля с приложением и имя объекта, который нужно запустить. Ещё одним аргументом --reload мы говорим ему обновлять запущенное приложение при изменении в файлах приложения. То же самое, что debug=True у Flask. И смело можем забыть об этой утилите, пока будем вести разработку.

Ну и проверим, что всё работает: откроем приложение в браузере.

Кстати, если мы вдруг забудем указать ключевое слово async при описании эндпоинта, то ничего криминального не произойдёт. Напишем два эндпоинта и проверим, что всё работает.

```
@app.get("/hello_async")
async def hello_async() -> Dict[str, str]:
   return {"message": "Hello World"}


@app.get("/hello_sync")
def hello_sync() -> Dict[str, str]:
   return {"message": "Hello World"}
```

// показываю

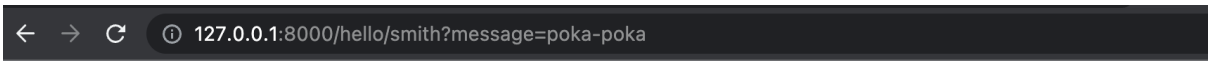
Как видим, оба эндпоинта отработали. Но почему? FastAPI внутри достаточно умён и смог понять, что мы обернули в декоратор не корутину, а обычную функцию. Вместо того чтобы вызвать await на ней напрямую, он отправляет её исполнение в отдельный threadpool и уже на нём вызывает операцию await. В итоге мы можем легко мешать друг с другом синхронные и асинхронные эндпоинты в зависимости от типа задачи. И это всё ещё будет работать достаточно быстро.

Отлично. Теперь усложним эндпоинт, пусть он будет принимать на вход парочку аргументов: из URL и как query-параметр.

```
@app.get('/hello/{who}')
async def say_hello(who: int, message: str = 'hello') -> Dict[str,
str]:
    fake_users_db = {1: 'admin', 2: 'John'}
    user = fake_users_db.get(who, 'username')
    return {'message': f'{message}, {user}'}
```

Как мы уже упоминали, FastAPI крутится вокруг аннотации типов. Именно через аннотацию типов мы указываем тип параметра в URL и query. Но и если вы уберёте аннотацию, тоже ничего страшного. Он будет того же типа, что в URL, то есть строкой. Но вот если мы укажем тип и попытаемся передать что-то не то, например в нашем случае строку, то FastAPI сам провалидирует этот параметр и выкинет понятную ошибку. Тут стоит обратить внимание на то, что хоть мы и указали возвращаемый тип из функции, эта аннотация исключительно для нас. Как правильно указать возвращаемый тип для FastAPI, мы узнаем совсем скоро.

```
http://127.0.0.1:8000/hello/smith?message=poka-poka
```



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

И заметьте, мы не писали никаких валидаторов, всё это сделал за нас сам фреймворк.

И ещё один нюанс аннотаций: если мы уберём дефолтное значение для query-параметра и попробуем вызвать эндпоинт без него, то опять поймаем ошибку валидатора. На этот раз нам любезно сообщат, что не хватает обязательного query-параметра.

async def say_hello(who: int, message: str):

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

Но то всё была валидация из коробки. Как бы нам написать свою кастомную валидацию? Добавить сообщения понятнее, например. И проверять, что ID не более того, что мы ожидаем. Довольно просто.

```
@app.get('/hello/{who}')
async def say_hello(
    who: int = Path(
        ...,
        title='Id of the user to whom to send the message.',
        ge=0,
        le=4,
    ),
    message: Optional[str] = Query(
        None,
        title="Say something pleasant to the user. Don't be
arrogant (do not use uppercase)",
        regex='^[a-z0-9_\-]+$'
    )
) -> Dict[str, str]:
    fake_users_db = {1: 'admin', 2: 'John'}
    user = fake_users_db.get(who, 'username')
    return {'message': f'{message}, {user}'}
```

Для подробного описания параметров URL и query есть два специальных класса: Path и Query. Они не слишком сильно различаются по способу использования: мы их используем как дефолтные аргументы при описании эндпоинта. Параметры же этих объектов, по сути, то поведение, которое мы хотим добавить в валидатор. Обратим внимание на первый аргумент этих классов: туда мы передаём тип аргумента, который хотим получить в эндпоинте. В первом случае мы передали объект типа элипсис — вот эти три точки. Это объект, который используют вместо ключевого слова pass или строки документации в пустом теле функции, чтобы подчеркнуть, что значение тут неважно. Плюс разные библиотеки, NumPy или pydantic с FastAPI используют этот тип, чтобы маркировать некоторое специальное поведение. Таким образом, вы говорим использовать тип из аннотации. Во втором случае мы говорим, что тип будет None. И обратите внимание, что мы добавили в аннотацию optional.

Тут же мы можем добавить описание параметров — это нам потом пригодится в документации. А также разные проверки. Например, мы хотим, чтобы ID был не меньше нуля и не больше четырёх. Если переданный параметр отличается от этих требований — будет ошибка. В случае message мы указали regex, который проверит, что в строке нет uppercase-символов.
Всего несколько дополнительных строк, а сколько логики по валидации мы написали. В случае Flask нам бы пришлось всё это писать самим.

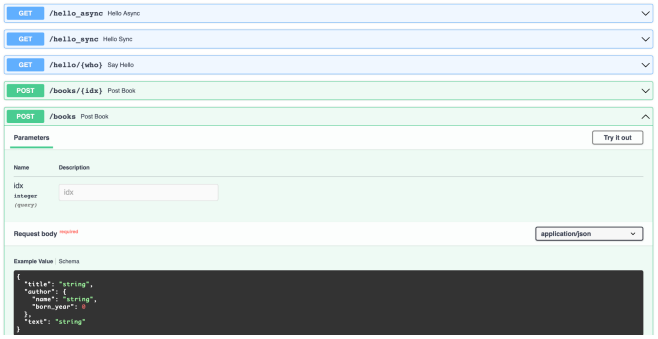
Хорошо, а теперь давайте опишем эндпоинт, который по методу post будет принимать какой-нибудь json.

```
from pydantic import BaseModel


class Author(BaseModel):
   name: str
   born_year: int


class Book(BaseModel):
   title: str
   author: Author
   text: str


@app.post('/books')
@app.post('/books/{idx}')
async def detect_hello(book: Book, idx: Optional[int] = None) ->
Dict[str, str]:
   return {
       'message': f"{book.author.name} wrote a great book! I
definitely will read {book.title}!"
   }
```

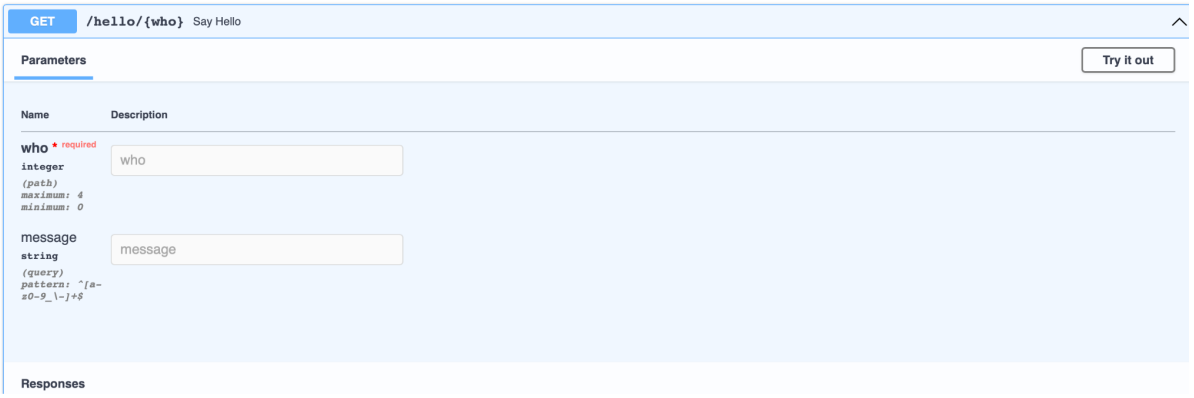
Как и в прошлый раз, всё строится вокруг аннотации типов. Но в этот раз нам нужно указать не просто базовый тип, а целую структуру. Ей послужит модель, которую мы унаследуем от базового класса, — basemodel из модуля pydantic. По смыслу это очень похоже на то, как мы с вами использовали dataclass. Кстати, если у вас уже есть датаклассы в проекте, но вы хотите их использовать в эндпоинтах FastAPI, то можете смело это делать. FastAPI умеет с ними работать и будет использовать методы библиотеки pydantic для валидации объектов.

Как видим, все атрибуты моделей проаннотированны, и даже есть вложенность. В самом эндпоинте нам нужно, как и в случае с URL и query-параметрами, указать в аргументах функции тип аргумента, который мы ожидаем. И ещё одна немаловажная деталь: мы можем декорировать дважды одну функцию, так она будет работать с двумя URL. В данном случае у нас будет один обработчик на URL с ID и без.

Давайте попробуем что-нибудь отправить на этот эндпоинт. Кстати, в этом нам отлично поможет уже подключённый swagger. Помните, мы говорили, что у FastAPI есть документация из коробки? Посмотрим на неё.

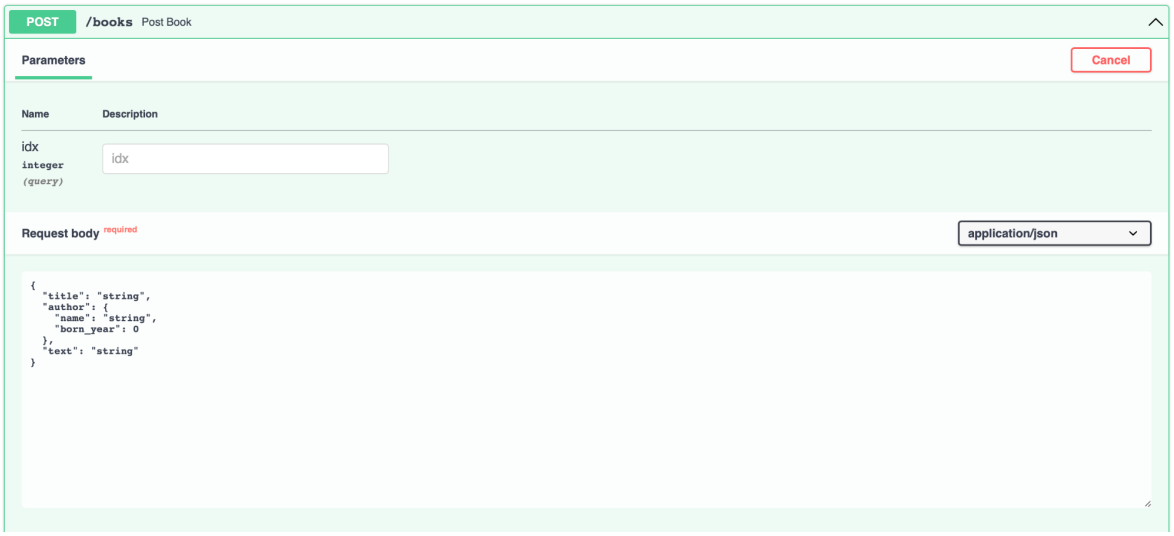


Для этого нам нужно перейти по ссылке host/docs, и мы увидим привычный интерфейс swagger. Быстро взглянем, что нам даёт аннотация типов. Например, посмотрим на предпоследний гет эндпоинт:



Тут мы сразу видим ограничения, которые мы сами на себя наложили. ID не длиннее, чем сколько-то, и вот такой регес.

Также у каждого эндпоинта есть кнопка try it out.



Если нажать на неё, swagger предложит сделать запрос на запущенный бэкенд. Давайте это сделаем.

Формируем тело запроса и жмём «Отправить».

Видим, в какую команду это будет преобразовано и что сервер успешно вернул ответ.

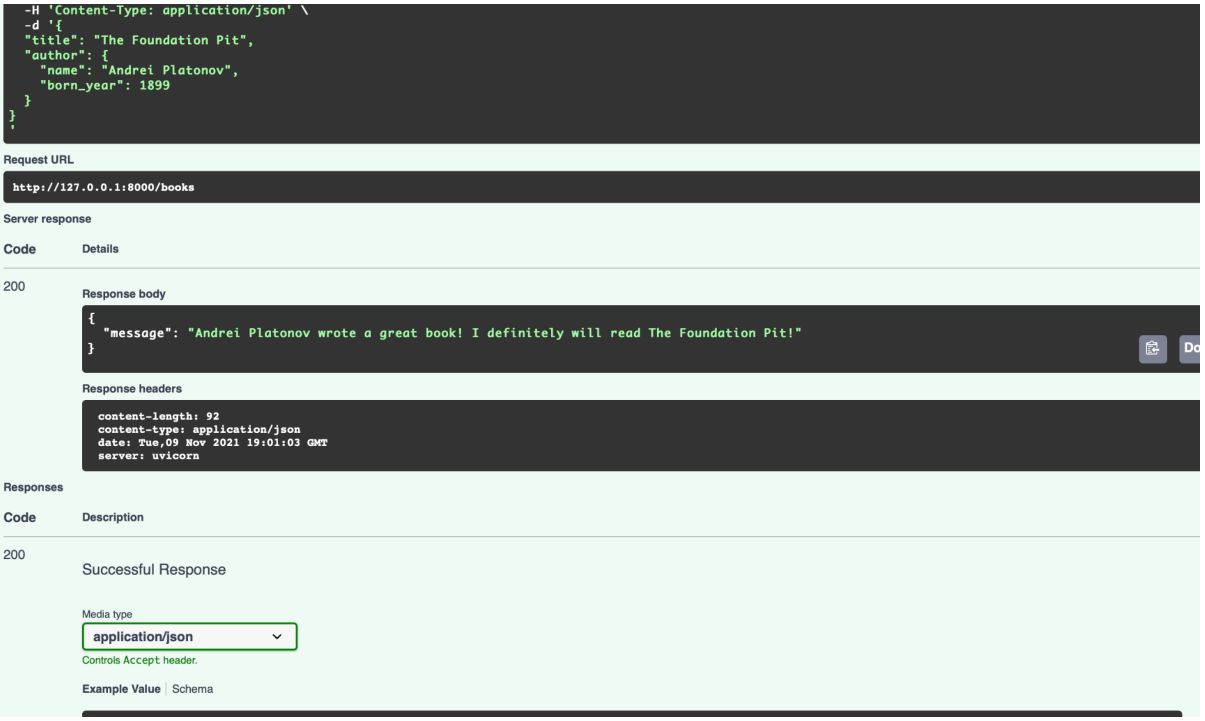
Делаем запрос, не забываем сделать вложенную структуру. Работает.
Но как будет работать валидация? Что, если мы забудем указать один из атрибутов? Давайте удалим параметр «Текст».

Удалили и получаем вполне понятную ошибку. Но если вдуматься, то, может быть, этот атрибут всё же опционален? Не всякий же раз просить передавать нам в json простыню текста. Давайте сделаем его опциональным и проверим, как изменится поведение валидатора.

Для этого в модели напишем следующее:

```
class Book(BaseModel):
    title: str
    author: Author
    text: Optional[str] = None
```

И попробуем сделать запрос ещё раз.



Отлично, работает! Но что насчёт чуть более продвинутой кастомизации валидации параметров, чем присутствие/отсутствие или подходящий тип? Да, кстати, валидация типа тут тоже подразумевается, хоть мы и не упомянули о ней явно.

Итак, чтобы добавить больше логики в валидацию параметров модели, как и в прошлые разы, нужно воспользоваться специальным классом.

```
from pydantic import BaseModel, Field
...

class Author(BaseModel):
    name: str
    born_year: int = Field(..., lt=2015)


class Book(BaseModel):
    title: str = Field(
        ...,
        title='Full title of the book.',
        min_length=3,
        max_length=100
    )
    author: Author
    text: Optional[str] = None
```

Для этого проимпортируем объект Field из модуля pydantic. Работает он точно так же, как и объект query или path. Тут мы описываем дефолтное значение, если это нужно, описываем ограничения. Например, мы не принимаем авторов младше шести лет. Или мы хотим, чтобы длина названия книги была не более 100 символов, но не менее трёх.

Также мы можем добавить дополнительную информацию о поле, она пригодится нам потом, в документации.

Все эти поля попадут в соответствующие атрибуты в модели swagger:

```
Book ⌄ {
   title*          string
                   title: Full title of the book.
                   maxLength: 100
                   minLength: 3
   author*         Author ⌄ {
                      name*          string
                                     title: Name
                      born_year*     integer
                                     title: Born Year
                                     exclusiveMaximum: 2015

                   }
   text            string
                   title: Text

}
```

Теперь посмотрим, как будет выглядеть ошибка валидации, если мы где-то ошибёмся.

```
  -d '{
  "title": "The Foundation Pit",
  "author": {
    "name": "Andrei Platonov",
    "born_year": 2020
  }
}
'
```

Request URL

```
http://127.0.0.1:8000/books
```

Server response

Code	Details
422	Error: Unprocessable Entity

Response body

```
{
  "detail": [
    {
      "loc": [
        "body",
        "author",
        "born_year"
      ],
      "msg": "ensure this value is less than 2015",
      "type": "value_error.number.not_lt",
      "ctx": {
        "limit_value": 2015
      }
    }
  ]
}
```

В результате опять придёт человекочитаемая ошибка, и нам сразу станет понятно, где мы ошиблись.

А что, если нам в одном из эндпоинтов потребуется получить параметр, которого нет в модели? Пусть у нас будет уникальный эндпоинт, где помимо самой книги мы будем ожидать ещё один параметр, например название издательства.

```
@app.post('/books')
@app.post('/books/{idx}')
async def detect_hello(
   book: Book,
   publisher: Optional[str] = Body(...) ,
   idx: Optional[int] = None,
) -> Dict[str, str]:
   return {
       'message': f"{book.author.name} wrote a great book! I
definitely will read {book.title}!"
   }
```

Для этого обновим наш эндпоинт следующим образом. Он всё так же будет ожидать объект книги на вход, но вместе с ним ещё один дополнительный атрибут.

```
@app.post('/books')
@app.post('/books/{idx}')
async def detect_hello(
   book: Book,
   publisher: Optional[str] = Body(...) ,
   idx: Optional[int] = None,
) -> Dict[str, str]:
   publisher_message = f'It was published by {publisher}' if
publisher else ''
   return {
       'message': f"{book.author.name} wrote a great book!"
                  + publisher_message
                  + f" I definitely will read {book.title}!"
   }
```

Перейдём в postman и посмотрим, как нам нужно изменить тело запроса.

Как видим, нам нужно было явно указать ключ, где будет находиться объект книги, и чуть ниже указать новый атрибут, который мы с вами только что добавили в обработчик. FastAPI, опять же, достаточно умён, чтобы понять, как правильно десериализовать этот json в python-объект. Ну и конечно, валидация тут всё ещё работает.

Итак, в этом уроке мы с вами познакомились с базовыми возможностями библиотеки FastAPI, научились описывать эндпоинты и их параметры. Знаем, как работают валидаторы и что пользоваться ими очень-очень просто. В следующем уроке мы посмотрим, что можно делать с провалидированными данными дальше, а именно как работать с базой асинхронно, используя ОРМ SQLAlchemy.