KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 9 урок > Backend-разработка: что это такое. Фреймворк FastAPI для прототипирования backend-сервера

>Оглавление 9 урока

> Backend-разработка

Web-разработка – процесс создания веб-сайта или веб-приложения. Любое веб-приложение состоит из двух частей: **бэкенд** (англ. *backend*) и **фронтенд** (англ. *frontend*).

Frontend - это та часть приложения, которую видит пользователь (пользовательский интерфейс сайта - иконки, кнопки).

Backend - это та часть приложения, которая пользователю не видна (работа с сервером, обработка информации).

Backend-разработка – это часть **web-разработки**, заключающаяся в разработке сервера, к которому будут подключаться остальные участники и запрашивать из него информацию.

Например,существует сервер с информацией о пользователях – он хранит их имена, фамилии, дни рождения и так далее. Разработчик хочет организовать доступ к этим данным для пользователя, но хранить всю информацию в самом приложении невозможно - оно будет занимать очень много места. В этом случае разрабочик создает приложение, которое подключается к серверу и запрашивает из него необходимую информацию. Для этого и нужна **backend-разработка**.

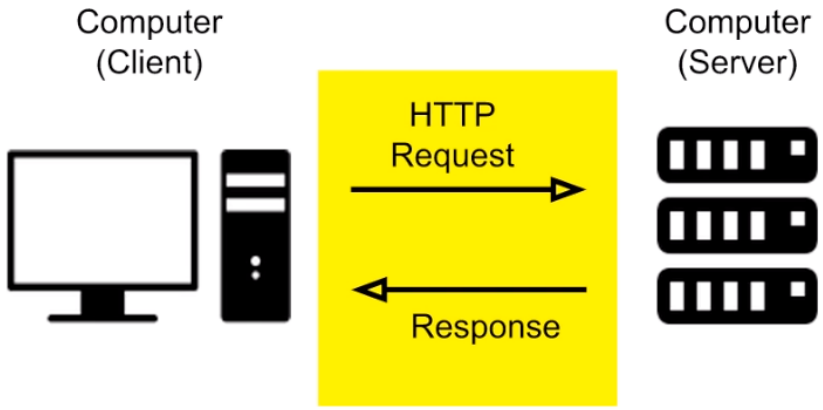
Зачем в машинном обучении нужна backend-разработка?

После того, как программист написал свою модель, он хочет сделать её доступной для пользователя. Для этих целей хорошо подойдет веб-сервис: модель сохраняется на сервер, и пользователь может запускать её когда угодно.

> Основы web-разработки

Большая часть того, что происходит в браузере, работает на механизме **веб-запросов** (англ. *requests*) и **ответов сервера** (англ. *response*).

Есть **клиент** (например, мобильное приложение), который отправляет **запрос**, и есть **сервер**, который принимает **запрос** и выдает на него **ответ**. Сервер отвечает на каждый запрос, даже если никакой информации передавать обратно клиенту не нужно.



Принцип взаимодействия клиента и сервера.

> Запрос на сервер

Запрос – это, по сути, большая текстовая строка, в которой содержится информация о том, что должен сделать сервер. Далее будет подробнее рассмотрена структура и параметры запроса.

Параметры веб-запроса:

- путь (англ. *path*)
- метод запроса (англ. *method*)
- query params
- заголовки запроса (англ. *headers*)
- тело запроса (англ. *body*)

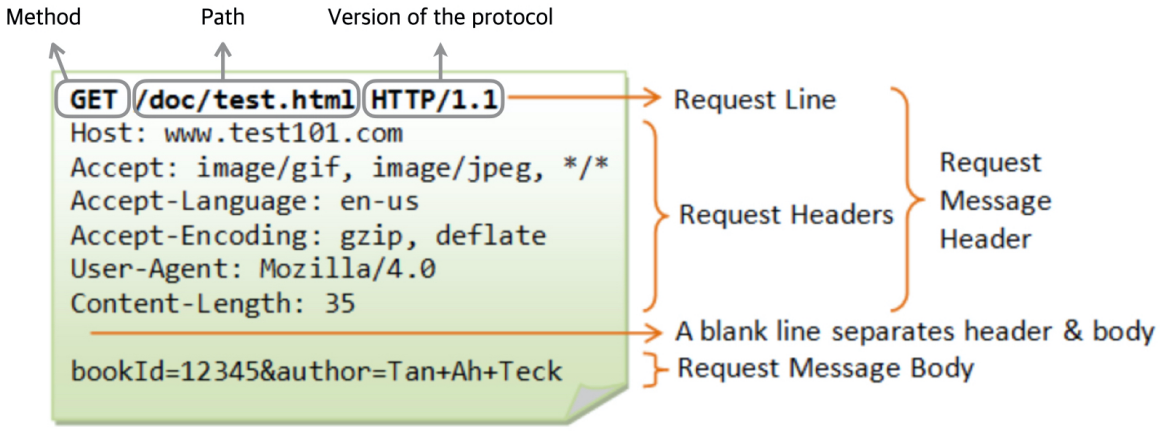


Схема и параметры запроса.

> Метод

Как уже было сказано ранее, **запрос** имеет **метод**, который сообщает серверу **цель** запроса.

Основные методы

1. **GET** - получение некоторых данных с сервера; например, при загрузке страницы пользователя в социальной сети будет отправлен GET запрос о его данных. GET запрос обычно не имеет body.
2. **PUT** - загрузка новых данных на сервер; например, при регистрации пользователя будет отправлен PUT запрос с регистрационными данными. PUT запрос обычно имеет body.
3. **POST** - редактирование информации, либо передача некоторой сущности на сервер; например, при изменении пользователем своего статуса статуса будет направлен POST запрос с новым статусом. POST запрос часто имеет body.

Методы GET, PUT и POST используются чаще всего, однако существуют и другие методы запроса.

Подробнее про методы.

> Путь

Запрос включает себя путь, указывающий серверу откуда брать информацию. Путь - это все, что идет в адресной строке после протокола и адреса сайта.

Рассмотрим следующий пример: **https://website.com/part/1/module/1/lesson1/**

- **https** - это протокол, использующийся при подключении к серверу.
- **website.com** - это адрес сайта.
- /part/1/module/1/lesson1/ - это путь. Путь к серверу схож с путем к папке на компьютере.

> Заголовки

Заголовки (жарг. *хидеры*) содержат служебную информацию: какой формат данных ожидается на выходе, какой браузер использует клиент, какой язык у клиента и т.п.

В заголовках также передаются **куки** (англ. *cookie*). Это небольшая порция данных, которая сохраняется на машине клиента и отправляется обратно серверу при каждом запросе, позволяя таким образом идентифицировать пользователя. **Cookies** хранятся в заголовке под ключом "cookie" в формате "ключ=значение".

Пример **заголовка** и **cookie**:

```
GET /sample_page.html HTTP/2.0
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Зачем нужны куки?

Куки имеют срок жизни и используются для **авторизации**. Когда пользователь заходит на сайт и вводит свои логин и пароль, сервер высылает ему **куки**. Проверяя **куки** данного пользователя, сервер обрабатывает его запросы без необходимости ввода логина и пароля. Это называется **Cookie-based authentication** - (устаревший) метод авторизации пользователей.

Статья про методы авторизации пользователей.

> Query params

Через **query params** в адресной строке можно передать какую-либо информацию вместе с запросом (например, id пользователя).

Синтаксис query params - вопросительный знак, затем сами query params в формате "ключ=значение"; если параметров несколько они разделяются знаком амперсанда &.

`/user?id=5&limit=10`
`/?q=hello`

> Тело запроса

Тело запроса - любая информация после заголовков (картинки, видео, файлы).

Мы будем работать со случаями, когда тело будет содержать информацию в формате **JSON**. Формат **JSON** схож со словарем в Python - он удобен и для человека, и для компьютера. Еще одно преимущество такого формата - его легко превратить в строку и
положить в тело запроса.

Пример **JSON**:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

> Ответ сервера

После получения запроса сервер его обрабатывает и возвращает (англ. *server response*). Ответ включает себя параметры запроса и некоторые дополнительные

параметры. О них будет рассказано далее.

Параметры ответа от сервера:

- параметры запроса (путь, протокол)
- status code
- информация о сервере (необязательный параметр)
- заголовки по безопасности

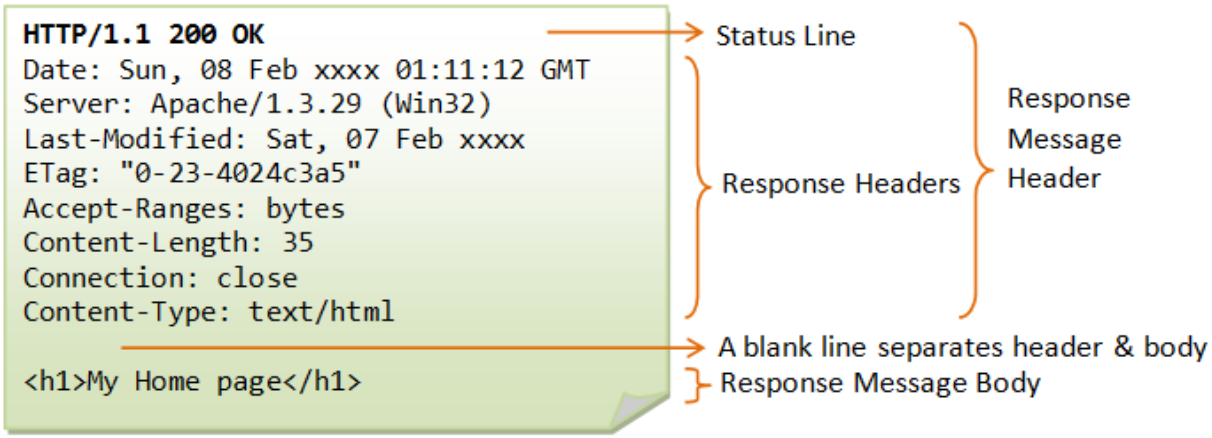


Схема и параметры ответа.

> Status code

Status code - число, несущее информацию о результатах обработки запроса. Расшифровка кодов тут.

Самые частые:

- **200** - все ок
- **30*** - будет перенаправление
- **400** - запрос некорректен и не будет обработан.
- **401** - неавторизованный запрос (нужна авторизация).
- **403** - запрещенный запрос (авторизация пройдена, но нет прав на запрос).

- **404** - не найдено (несуществующий путь или объект на сервере, либо нет прав на запрос).
- **500** - сервер упал.
- **502**, **504** - проблемы шлюза (внутри сервера существуют проблемы с внутренней коммуникацией компонент).

Расшифровка кодов:

- **20*** - все хорошо.
- **30*** технические коды по процессу соединения с сервером.
- **40*** "клиент неправ" (ошибка на стороне клиента)
- **50*** - "сам дурак" (ошибка на стороне сервера)

> Информация о сервере

Помимо кодов, ответ сервера может включать (а может и не включать) любую информацию, например:

- версию ПО
- версию ОС
- время на сервере
- и т.п.

> Заголовки по безопасности

Эти **заголовки** передаются для того, чтобы защитить пользователя от уязвимостей, например, CSRF-атак.

> Фреймворк. API

Фреймворк — программная среда специального назначения, шаблон для программной платформы, на основе которого можно писать собственный код.

API (Application programming interface) — это контракт, который предоставляет программа. **API** «говорит» о том, какие рычаги для взаимодействия открывает приложение сторонним разработчикам, и как они могут использовать приложение.

API отвечает на вопрос *«Как ко мне, к моей системе можно обратиться?»* и включает в себя:

- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,
- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

К примеру, разработчик написал программу, хранящую историю платежей пользователя. Этой программе можно добавить возможность отдавать историю конкретного пользователя, например, через веб-запрос. В таком случае говорят, что есть API для получения истории конкретного пользователя.

Статья на habr.

> FastAPI

FastAPI — это фреймворк для создания лаконичных и довольно быстрых HTTP API-серверов.

> REST API

REST API (REpresentational State Transfer) — это архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

Архитектурный стиль – это набор согласованных ограничений и принципов проектирования, позволяющий добиться определённых свойств системы.

Статья на mcs.

Свойства:

- Всё взаимодействие программ строится по модели «сервер-клиент». Т.е. кто-то выступает в роли сервера, обрабатывая запросы, а кто-то выступает в роли клиента, отправляя запросы.
- Отсутствие состояния. Между двумя запросами сервер не хранит состояния пользователя. Все необходимые для обработки запроса данных пользователь

должен передавать сам.

- Сервер выставляет наружу понятные иерархические пути (URL), объединяя их логически в блоки, и активно использует методы запроса.

К примеру, для операций со списком пользователей будут следующие URL:

- GET `/user/<id>` (выгрузить)
- GET `/user/me` (выгрузить себя)
- POST `/user/me` (обновить себя)
- POST `/user/` (добавлять произвольного пользователя)

> Создание веб-приложения

> Введение

Для того, чтобы создать обертку (веб-приложение) для нашей будущей модели, мы будем использовать фреймворк `FastAPI`.

Сначала нужно создать виртуальное окружение и файл `requirements.txt`. В файле `requirements.txt` приписывам название библиотек `fastapi` и `uvicorn`. Библиотека `fastapi` нужна для настройки работы API, `uvicorn` — для настройки сервера.

Далее создаем файл с расширением `.py` — это и будет файл нашего приложения (назовем его `app.py`) и прописываем в нем импорт и вызов класса FastAPI, метод запроса и сам запрос.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/sum")
def sum_two(a: int, b: int) -> int:
return a + b
```

Запускаем файл:

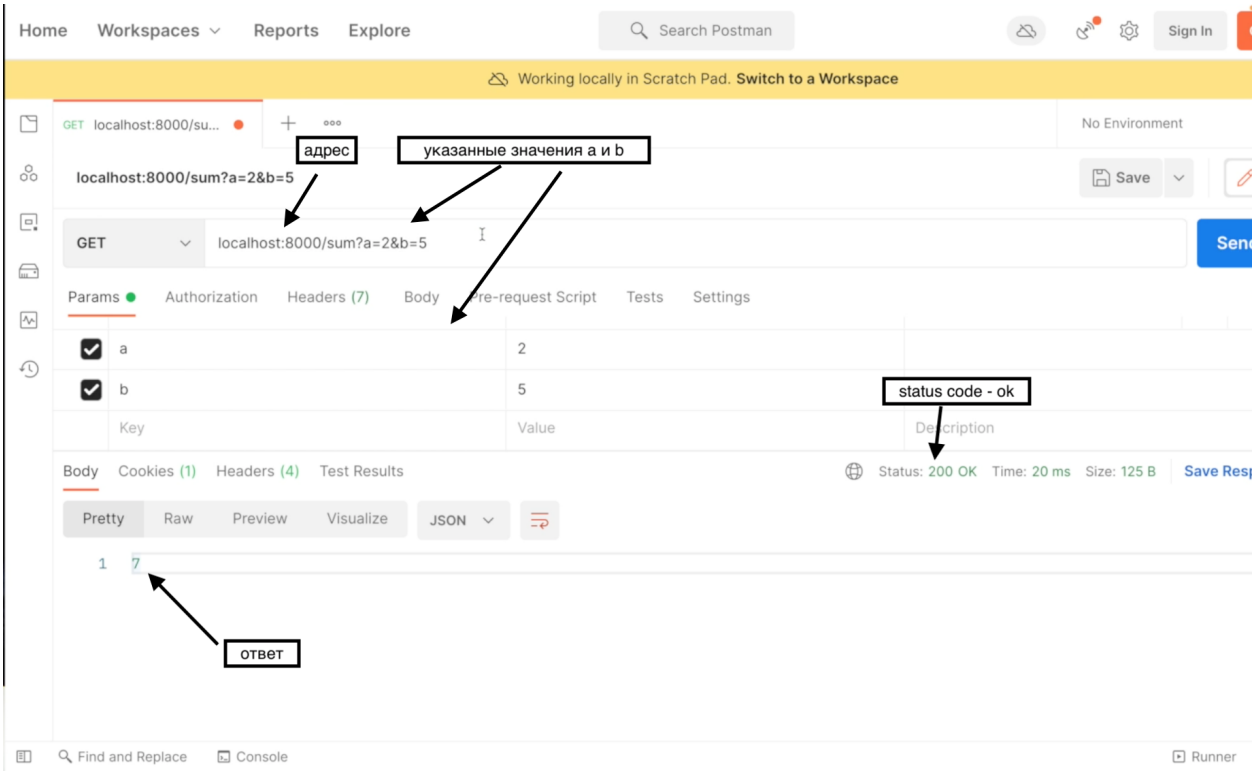
`uvicorn app:app --reload`

После запуска выведутся логи — адрес, порт.

Если пройти по ссылке (пути), который вывелся в логах, используя браузер, то увидим, какой запрос обработал сервер.

Чтобы более наглядно показать как работает наша программа, мы будем использовать софт Postman — это продвинутый браузер, который показывает детали общения клиент-сервер.

Вот как отработает наш код `app.py` в Postman.



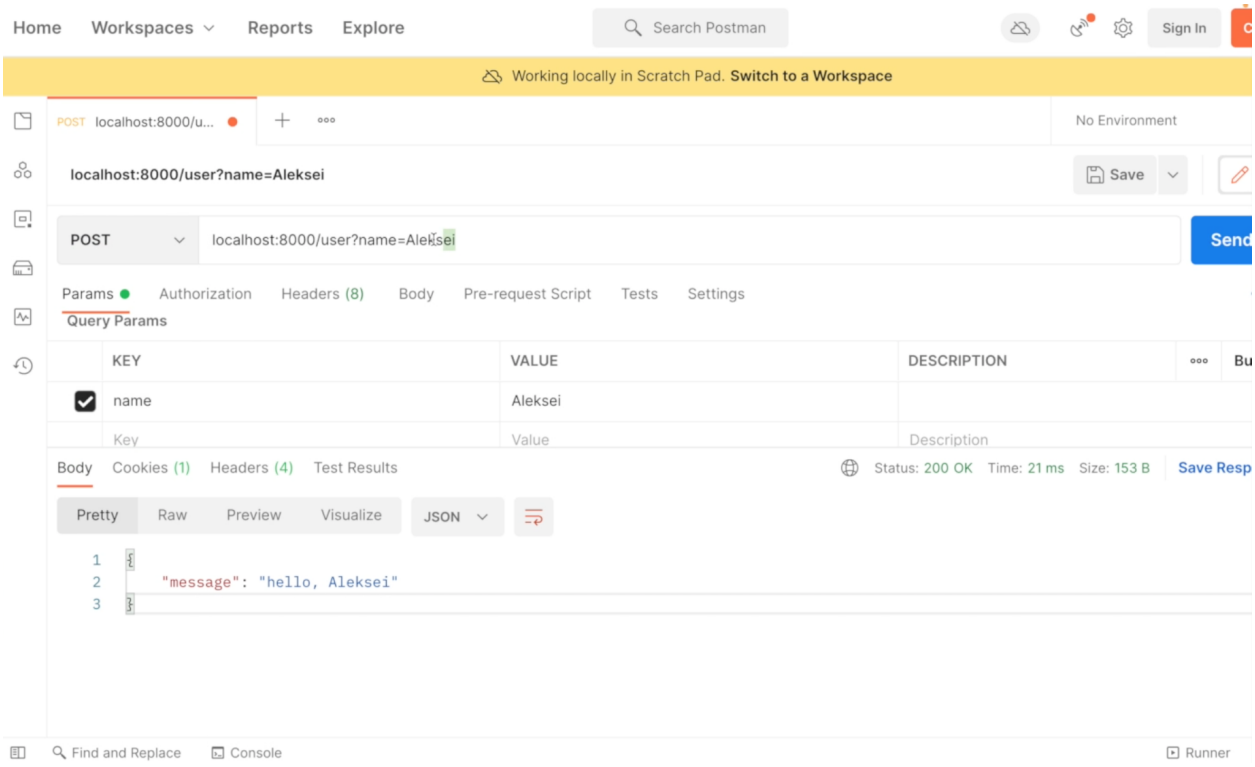
В `fastapi` есть еще один способ создания запросов — написание параметра в фигурных скобках.

```
@app.get("/print/{number}")
def print_num(num: int):
    return num*2

А вот пример для другого метода – `post`:

@app.post("/user")
def print_name(name: str):
    return {"message": "hello, {name}"}
```

Результат:



> Подключение базы данных

Подключение базы данных осуществляется с использованием библиотеки `psycopg2`.

```
@app.get("/booking/all")
def all_bookings():
    conn = psycopg2.connect("/PATH/")
    cursor = connect.cursor()
    cursor.execute(
        """
        SELECT *
        FROM cd.bookings
        """
)
    return cursor.fetchall()
```

> Dependency injection

Dependency Injection - способ прописать заранее объекты, которые необходимы для дальнейшей работы (dependencies - зависимости). После этого система (в нашем случае FastAPI) сможет использовать (inject) эти зависимости. Данный метод позволяет избежать повторения кода.

Для имплементации `dependency injection` используется класс `Depends()`, которому в качестве аргумента передается функция. В свою очередь, в этой функции создается объект, используемый системой.

```
from fastapi import Depends, FastAPI

app = FastAPI()

async def common_parameters(q: str | None = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
async def read_items(commons: dict = Depends(common_parameters)):
    return commons
```

Подробнее про это можно почитать в документации.

> Валидация

Чтобы валидировать (тестировать) работу кода, используется библиотека `pydentic` — для этого нужно написать класс, наследующийся от класса `BaseModel` с необходимыми инструкциями. Затем в методе запроса нужно передать название класса в переменную `response_model=<имя класса>`. Если класс для валидации возвращает несколько элементов, то нужно использовать функцию `List` из библиотеки `typing` — `response_model=List[<имя класса>]`.

```
class BookingGet(BaseModel):
    id : str # инструкция
    class Config:
        orm_mode = True
```

Если не удается понять, в чем ошибка, можно логгировать выполнение функции, используя библиотеку `loguru`.

> Status codes

Чтобы вернуть пользователю статус коды в нормальном виде, используется `HTTPException` из библиотеки `fastapi`.

```
from fastapi import HTTPException

@app.get("/error")

def show_error():
    if a == 5:
        raise HTTPException(304)
    return 'ok'
```

> Значение @

@ перед названием методов обозначает вызов декоратора.

Декоратор — это «обёртка», которая дает возможность изменить поведение функции, не изменяя её код.

Когда мы описываем логику работы методов API, мы настраиваем эти методы, не меняя их. Таким образом, мы переиспользуем код и добавляем новую функциональность методу.

Статья на habr.

Еще одна статья на habr.

> Сетевые запросы в Python

Для того, чтобы делать сетевые запросы не в Postman, а в самом Python понадобится библиотека `requests`.

```
import requests
```

```
r = requests.get("/PATH/") # метод get
r = requests.post("/PATH/", json = {'name': 'Aleksei'}) # метод post
```