

## Week 7 – Основы FastAPI

### Task 1- Что такое FastAPI?

**FastAPI** представляет быстрый высокопроизводительный фреймворк для создания веб-приложений на языке Python.

Официальный сайт проекта: <https://fastapi.tiangolo.com/>. Исходный код фреймворка доступен на github по адресу: <https://github.com/tiangolo/fastapi>

На данный момент поддерживается Python версии 3.6 и выше.

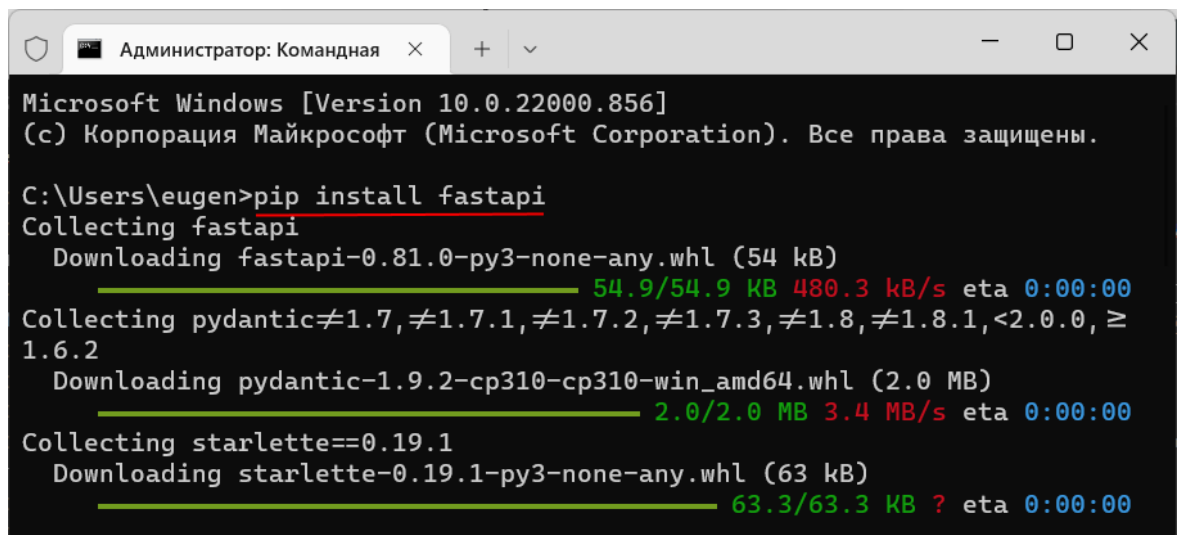
### Необходимые инструменты и установка

Для работы с FastAPI естественно потребуется интерпретатор Python.

Для установки пакетов FastAPI потребует пакетный менеджер **pip**. Менеджер **pip** позволяет загружать пакеты и управлять ими. Обычно при установке python также устанавливается и менеджер **pip**.

Для установки пакетов FastAPI откроем терминал и введем команду

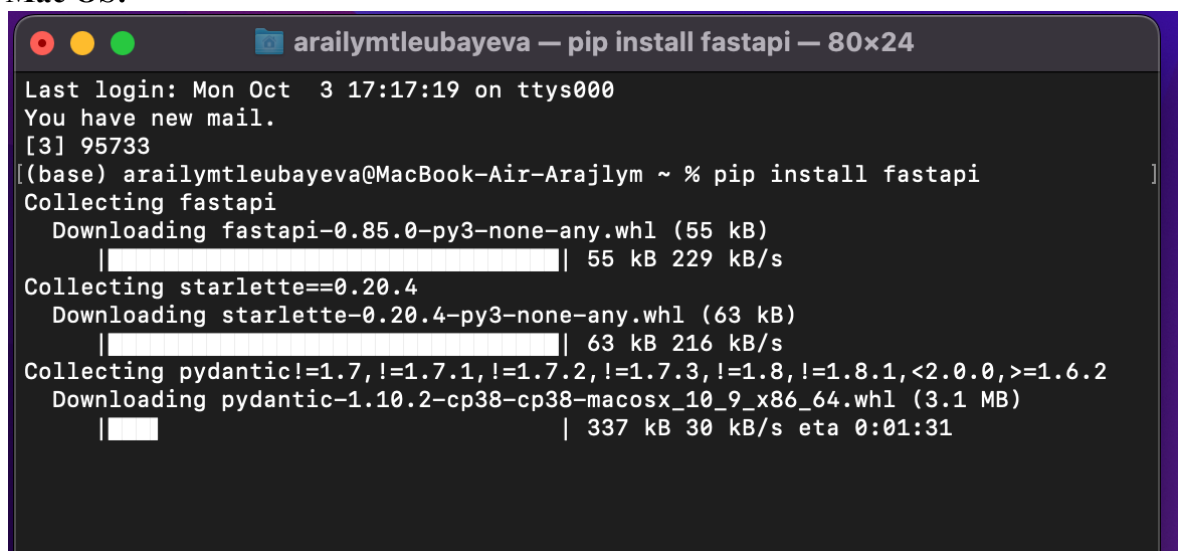
**pip install fastapi**



```
Microsoft Windows [Version 10.0.22000.856]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\eugen>pip install fastapi
Collecting fastapi
  Downloading fastapi-0.81.0-py3-none-any.whl (54 kB)
    |#####| 54.9/54.9 KB 480.3 kB/s eta 0:00:00
Collecting pydantic!=1.7,!=1.7.1,!=1.7.2,!=1.7.3,!=1.8,!=1.8.1,<2.0.0,>=1.6.2
  Downloading pydantic-1.9.2-cp310-cp310-win_amd64.whl (2.0 MB)
    |#####| 2.0/2.0 MB 3.4 MB/s eta 0:00:00
Collecting starlette==0.19.1
  Downloading starlette-0.19.1-py3-none-any.whl (63 kB)
    |#####| 63.3/63.3 KB ? eta 0:00:00
```

Mac OS:



```
Last login: Mon Oct 3 17:17:19 on ttys000
You have new mail.
[3] 95733
(base) arailymtleubayeva@MacBook-Air-Arajlym ~ % pip install fastapi
Collecting fastapi
  Downloading fastapi-0.85.0-py3-none-any.whl (55 kB)
    |#####| 55 kB 229 kB/s
Collecting starlette==0.20.4
  Downloading starlette-0.20.4-py3-none-any.whl (63 kB)
    |#####| 63 kB 216 kB/s
Collecting pydantic!=1.7,!=1.7.1,!=1.7.2,!=1.7.3,!=1.8,!=1.8.1,<2.0.0,>=1.6.2
  Downloading pydantic-1.10.2-cp38-cp38-macosx_10_9_x86_64.whl (3.1 MB)
    |#####| 337 kB 30 kB/s eta 0:01:31
```

Также для работы с FastAPI нам потребуется ASGI веб-сервер (веб сервер с поддержкой протокола Asynchronous Server Gateway Interface). В качестве такового в Python можно использовать [Univer](#) или [Hypercorn](#). В данном случае будем использовать Univer. Также установим его пакеты с помощью менеджера pip с помощью следующей команды:

```
pip install "uvicorn[standard]"
```

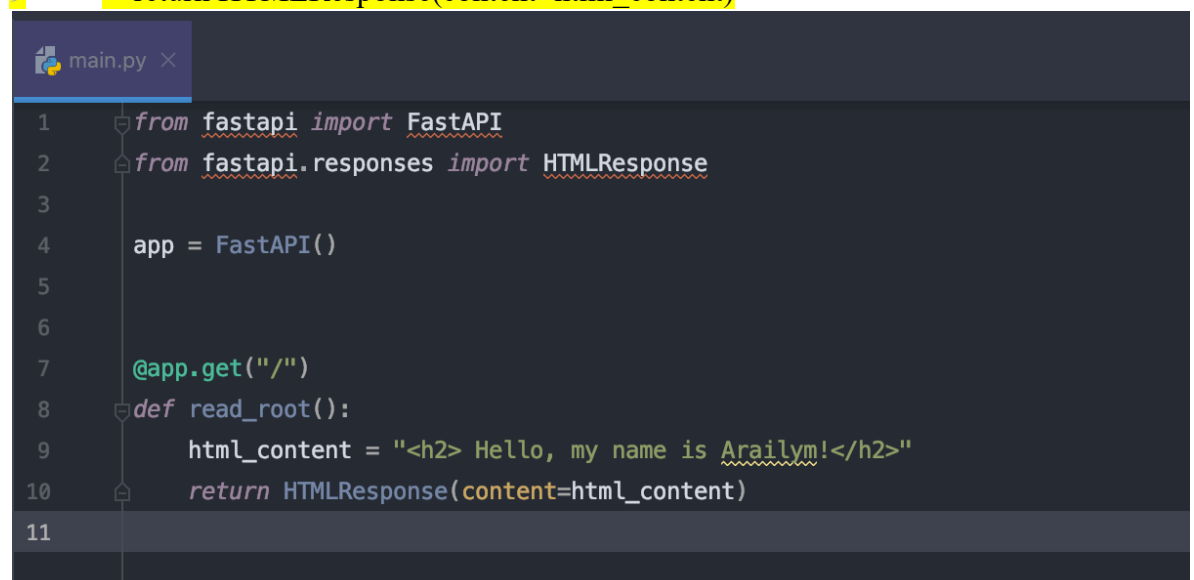
```
(base) arailymtleubayeva@MacBook-Air-Arajlym ~ % pip install "uvicorn[standard]"

Collecting uvicorn[standard]
  Downloading uvicorn-0.18.3-py3-none-any.whl (57 kB)
    |#####| 57 kB 247 kB/s
Requirement already satisfied: click>=7.0 in ./opt/anaconda3/lib/python3.8/site-packages (from uvicorn[standard]) (7.1.2)
Requirement already satisfied: h11>=0.8 in ./opt/anaconda3/lib/python3.8/site-packages (from uvicorn[standard]) (0.9.0)
Collecting uvloop!=0.15.0,!=0.15.1,>=0.14.0; sys_platform != "win32" and (sys_platform != "cygwin" and platform_python_implementation != "PyPy") and extra == "standard"
  Downloading uvloop-0.17.0-cp38-cp38-macosx_10_9_x86_64.whl (1.5 MB)
    |#####| 430 kB 33 kB/s eta 0:00:33
```

## Создание первого приложения

Определим на диске папку, где будут располагаться файлы с исходным кодом приложения. Например, в моем случае это папка **C:\fastapi**. Создадим в этой папке новый файл, который назовем **main.py** и который будет иметь следующий код:

```
1 from fastapi import FastAPI
2 from fastapi.responses import HTMLResponse
3
4 app = FastAPI()
5
6 @app.get("/")
7 def read_root():
8     html_content = "<h2> Hello, my name is Arailym!</h2>"
9     return HTMLResponse(content=html_content)
```



```
main.py X
1 from fastapi import FastAPI
2 from fastapi.responses import HTMLResponse
3
4 app = FastAPI()
5
6 @app.get("/")
7 def read_root():
8     html_content = "<h2> Hello, my name is Arailym!</h2>"
9     return HTMLResponse(content=html_content)
10
11
```

Для обработки запросов к приложения вначале необходимо создать объект приложения с помощью конструктора **FastAPI** из пакета **fastapi**

```
1 app = FastAPI()
```

Затем определяем функцию, которая будет обрабатывать запросы. К этой функции применяется специальный декоратор в виде метода **app.get()**:

```
1 @app.get("/")
```

В этот метод передается шаблон маршрута, по которому функция будет обрабатывать запросы. В данном случае это строка `"/"`, которая означает, что функция будет обрабатывать запросы по пути `"/"`, то есть запросы к корню веб-приложения.

После декоратора `app.get` идет собственно определение функции, которая обрабатывает запрос:

```
1 def read_root():
2     html_content = "<h2>Hello, my name is Arailym!</h2>"
3     return HTMLResponse(content=html_content)
```

Это обычная функция python. Она называется `read_root` (имя произвольное). Для отправки ответа она использует класс **HTMLResponse** из пакета `fastapi.responses`.

Класс `HTMLResponse` позволяет отправить в ответ некоторое содержимое в виде кода `html`.

Для установки отправляемого содержимого в конструкторе `HTMLResponse` применяется параметр `content`, которому в данном случае передается строка `"<h2>Hello, my name is Arailym!</h2>"` со значением `"Hello, my name is Arailym!"`. То есть когда клиент обратится к веб-приложению по пути `"/"`, ему будет отправлен `html`-код `"<h2>Hello!</h2>"`.

### Запуск приложения

Теперь запустим приложение. Для этого перейдем в терминале к папке, где располагается файл **main.py** и затем выполним команду:

```
uvicorn main:app --reload
```

В данном случае мы запускаем сервер `uvicorn` и передаем ему ряд параметров:

- `main` указывает на название модуля, которое по умолчанию совпадает с названием файла - `main`
- `app` указывает на объект приложения, созданный в строке `app = FastAPI()`
- `--reload` позволяет отслеживать изменения в файлах исходного кода и автоматически перезапускать проект.

```
Администратор: Командная
Microsoft Windows [Version 10.0.22000.856]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\eugen>cd c:\fastapi ← переход к папке с исходным кодом

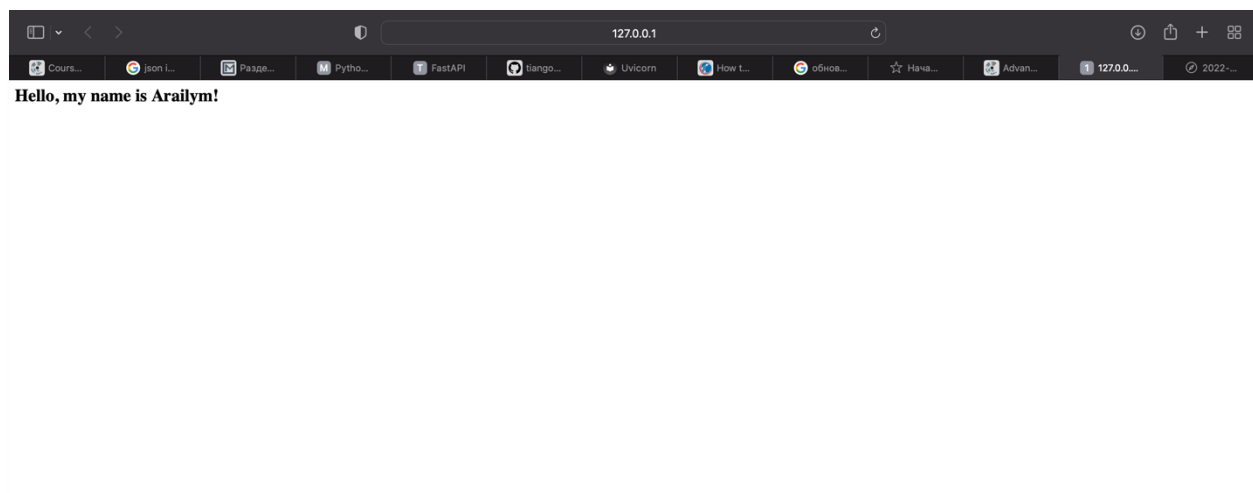
c:\fastapi>uvicorn main:app --reload ← запуск приложения
INFO: Will watch for changes in these directories: ['c:\\fastapi']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [6960] using WatchFiles
INFO: Started server process [6612]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

адрес, по которому доступно приложение

### В терминале PyCharm:

```
(venv) (base) arailymtleubayeva@MacBook-Air-Arajlym fastapi % uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['/Users/arailymtleubayeva/PycharmProjects/fastapi']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [5724] using WatchFiles
INFO: Started server process [5731]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

При запуске консоль отображает адрес, по которому запущено приложение. Обычно это адрес `http://127.0.0.1:8000/`. И если мы обратимся по этому адресу в браузере, то он отобразит нам отправленные сервером данные:



## Task 2 - Класс FastAPI и обработка запроса

В центре приложения FastAPI находится одноименный класс **FastAPI** из пакета `fastapi`. Данный класс фактически и представляет приложение FastAPI. Этот класс наследуется от класса `starlette.applications.Starlette` [Starlette](#) представляет другой легковесный ASGI-фреймворк для создания асинхронных веб-сервисов на Python. Собственно fastAPI работает поверх Starlette, используя и дополняя его функциональность. Это касается не только самого класса FastAPI, но и других классов фреймворка - многие из них используют функционал Starlette.

Конструктор класса FastAPI имеет около трех десятков различных параметров, которые позволяют настроить работу приложения. Но в общем случае для создания функционирующего объекта класса можно не передавать в конструктор никаких аргументов, тогда параметры получают значения по умолчанию:

```
from fastapi import FastAPI
app = FastAPI()
```

### Методы FastAPI

Одним из преимуществ FastAPI является то, что фреймворк позволяет быстро и легко построить веб-сервис в стиле REST. Архитектура REST предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером, где каждый тип запроса отвечает за определенное действие:

- **GET** (получение данных)
- **POST** (добавление данных)
- **PUT** (изменение данных)
- **DELETE** (удаление данных)

Кроме этих типов запросов HTTP поддерживает еще ряд, в частности:

- **OPTIONS**
- **HEAD**
- **PATCH**
- **TRACE**

В классе FastAPI для каждого из этих типов запросов определены одноименные методы:

- `get()`
- `post()`
- `put()`
- `delete()`
- `options()`
- `head()`
- `patch()`
- `trace()`

Например, если нам надо обработать HTTP-запрос типа GET, то применяется метод `get()`.

Все эти методы имеют множество параметров, но все они в качестве обязательного параметра принимают путь, запрос по которому должен обрабатываться.

Причем эти методы сам запрос не обрабатывают - они применяются в качестве декоратора к функциям, которые непосредственно обрабатывают запрос. Например:

```
from fastapi import FastAPI
```

```

app = FastAPI()

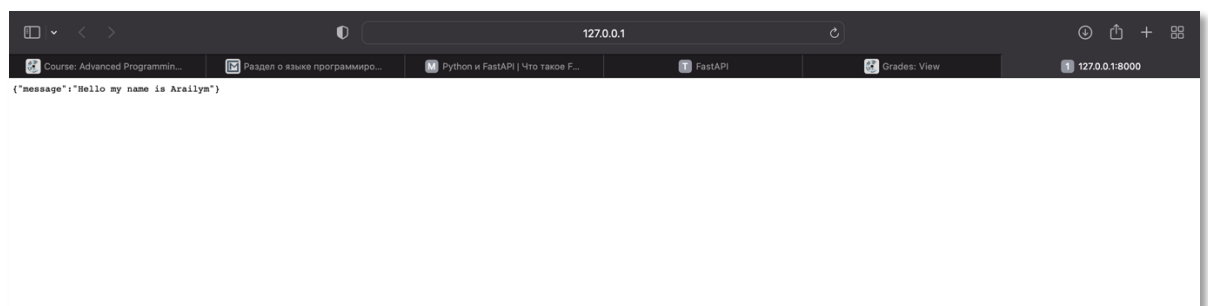
@app.get("/")
def root():
    return {"message": "Hello my name is Arailym"}

```

В данном случае метод `app.get()` применяется в качестве декоратора к функции `root()` (символ `@` указывает на определение декоратора). Этот декоратор определяет путь, запросы по которому будет обрабатывать функция `root()`. В данном случае путь представляет строку `"/"`, то есть функция будет обрабатывать запросы к корню веб-приложения (например, по адресу `http://127.0.0.1:8000/`).

Функция возвращает некоторый результат. Обычно это словарь (объект `dict`). Здесь словарь содержит один элемент `"message"`. При отправке эти данные автоматически сериализуются в формат JSON - популярный формат для взаимодействия между клиентом и сервером. А у ответа для заголовка `content-type` устанавливается значение `application/json`. Вообще функция может возвращать различные данные - словари (`dict`), списки (`list`), одиночные значения типа строк, чисел и т.д., которые затем сериализуются в json.

Соответственно, если мы запустим приложение и обратимся по адресу `http://127.0.0.1:8000/`, например, в браузере, то мы получим ответ сервера в формате json:



Подобным образом можно определять и другие функции, которые будут обрабатывать запросы по другим путям. Например:

```

from fastapi import FastAPI

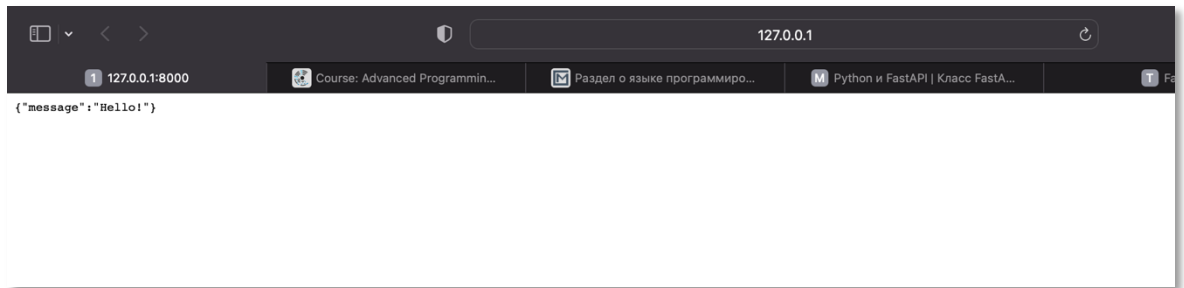
app = FastAPI()

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

@app.get("/about")
def about():
    return {"message": "О сайте"}

```

Здесь добавлена функция `about()`, которая обрабатывает запросы по пути `"/about"`:



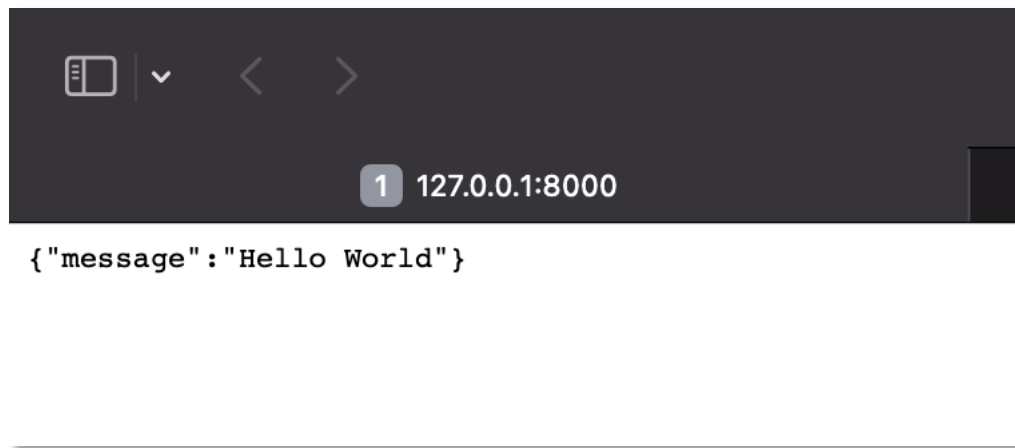
### Task 3 -Отправка ответа

В прошлой теме рассматривалось обработка запроса функциями, например, простейший код:

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

Здесь функция `root` обрабатывает запрос по пути `"/"` (то есть обращение к корню веб-приложения) и возвращает некоторый результат в виде словаря (объект `dict`), который содержит один элемент `"message"`. При отправке эти данные автоматически сериализуются в формат JSON - популярный формат для взаимодействия между клиентом и сервером. А у ответа для заголовка `content-type` устанавливается значение `application/json`. Вообще функция может возвращать различные данные - словари (`dict`), списки (`list`), одиночные значения типа строк, чисел и т.д., которые затем сериализуются в json с помощью кодировщика `fastapi.encoders.jsonable_encoder`. А для отправки ответа FastAPI по умолчанию использует класс `fastapi.responses.JSONResponse`. То есть предыдущий код в принципе будет эквивалентен следующему:

```
1 from fastapi import FastAPI
2 from fastapi.responses import JSONResponse
3 from fastapi.encoders import jsonable_encoder
4
5 app = FastAPI()
6
7 @app.get("/")
8 def root():
9     data = {"message": "Hello World"}
10    json_data = jsonable_encoder(data)
11    return JSONResponse(content=json_data)
```



Параметр `content` задает отправляемые данные. Либо можно без явной сериализации передать данные в `JSONResponse`:

```
from fastapi import FastAPI
from fastapi.responses import JSONResponse

app = FastAPI()

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

И при необходимости отправить какие-то нестандартные несериализуемые данные или, если нас не устраивает сериализация по умолчанию, мы можем определить свой сериализатор `json`.

Однако функция необязательно должна возвращать именно данные в формате `json`. В реальности мы можем вернуть объект класса **Response** или одного из его подклассов (коим также является `JSONResponse`), которые позволяют отправлять клиенту ответ в различных видах и формах. Рассмотрим некоторые из этих классов.

#### Response

Класс **fastapi.Response** является базовым для остальных классов ответа. Его преимуществом является то, что он позволяет также отправить ответ, который не покрывается встроенными классами, например, в каком-то нестандартном формате. Для определения ответа конструктор класса принимает следующие параметры:

**content**: задает отправляемое содержимое

**status\_code**: задает статусный код ответа

**media\_type**: задает MIME-тип ответа

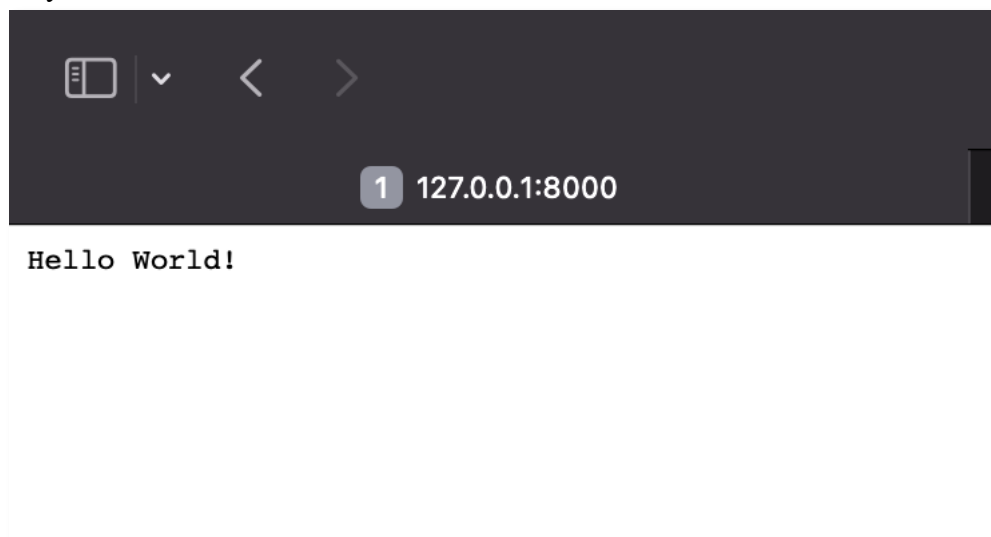
**headers**: задает заголовки ответа

Рассмотрим простейший пример:



```
main.py x
1  from fastapi import FastAPI, Response
2
3  app = FastAPI()
4
5
6  @app.get("/")
7  def root():
8      data = "Hello World!"
9      return Response(content=data, media_type="text/plain")
10
```

Результат:



В данном случае клиенту отправляет обычная строка "Hello World! ". А MIME-тип "text/plain" указывает, что тип ответа - простой текст.

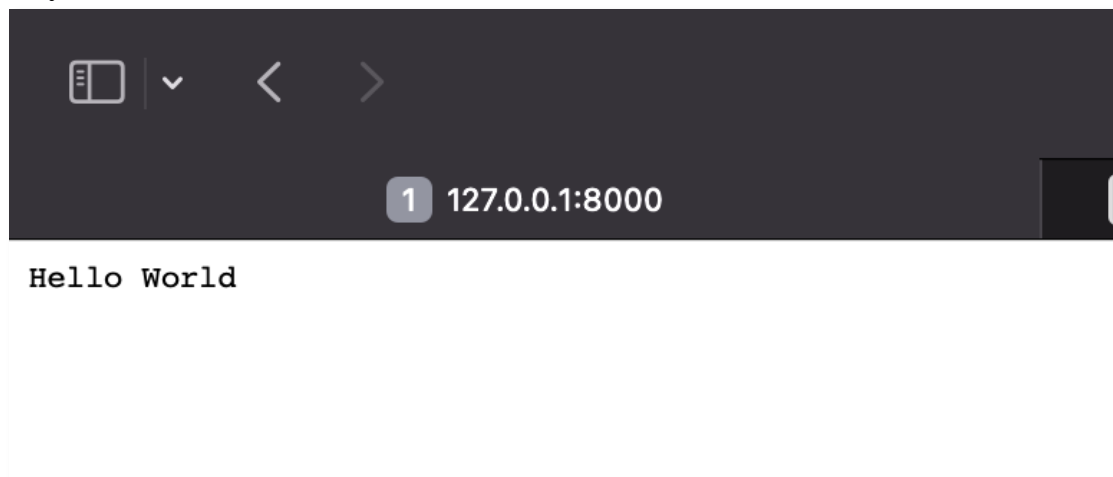
### PlainTextResponse

Для отправки простого текста также можно использовать класс-наследник **PlainTextResponse**

Исходной код main.py:

```
main.py x
1  from fastapi import FastAPI
2  from fastapi.responses import PlainTextResponse
3
4  app = FastAPI()
5
6
7  @app.get("/")
8  def root():
9      data = "Hello World"
10     return PlainTextResponse(content=data)
11
```

Результат:

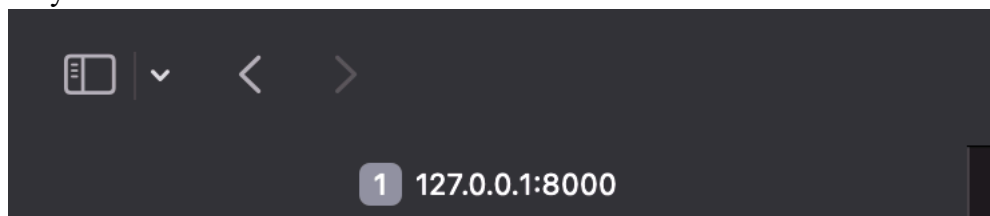


### HTMLResponse

Для упрощения отправки кода html предназначен класс **HTMLResponse**. Он устанавливает для заголовка Content-Type значение text/html:

```
main.py x
1  from fastapi import FastAPI
2  from fastapi.responses import HTMLResponse
3
4  app = FastAPI()
5
6
7  @app.get("/")
8  def root():
9      data = "<h2>Hello World!</h2>"
10     return HTMLResponse(content=data)
11
```

Результат:



# Hello World!

## Установка типа ответа через методы FastAPI

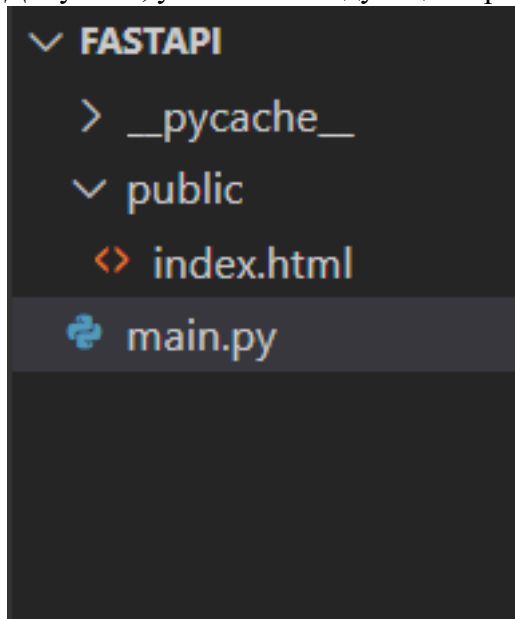
Методы FastAPI такие как `get()`, `post()` и т.д. позволяют задать тип ответа с помощью параметра **response\_class**:

```
1  from fastapi import FastAPI
2  from fastapi.responses import PlainTextResponse, JSONResponse, HTMLResponse
3
4  app = FastAPI()
5
6  @app.get("/text", response_class = PlainTextResponse)
7  def root_text():
8      return "Hello World!"
9
10 @app.get("/html", response_class = HTMLResponse)
11 def root_html():
12     return "<h2>Hello World!</h2>"
```

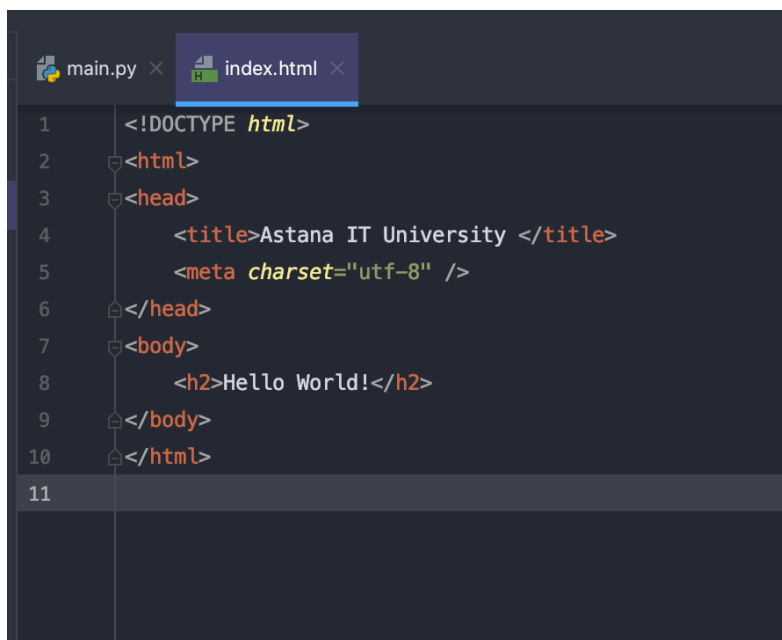
## Task 4 - Отправка файлов с сервера

Для отправки файлов из приложения FastAPI применяется класс `FileResponse` - наследник класса `Response`.

Допустим, у нас есть следующий проект:



Пусть в проекте в папке `public` есть файл `index.html` со следующим кодом:



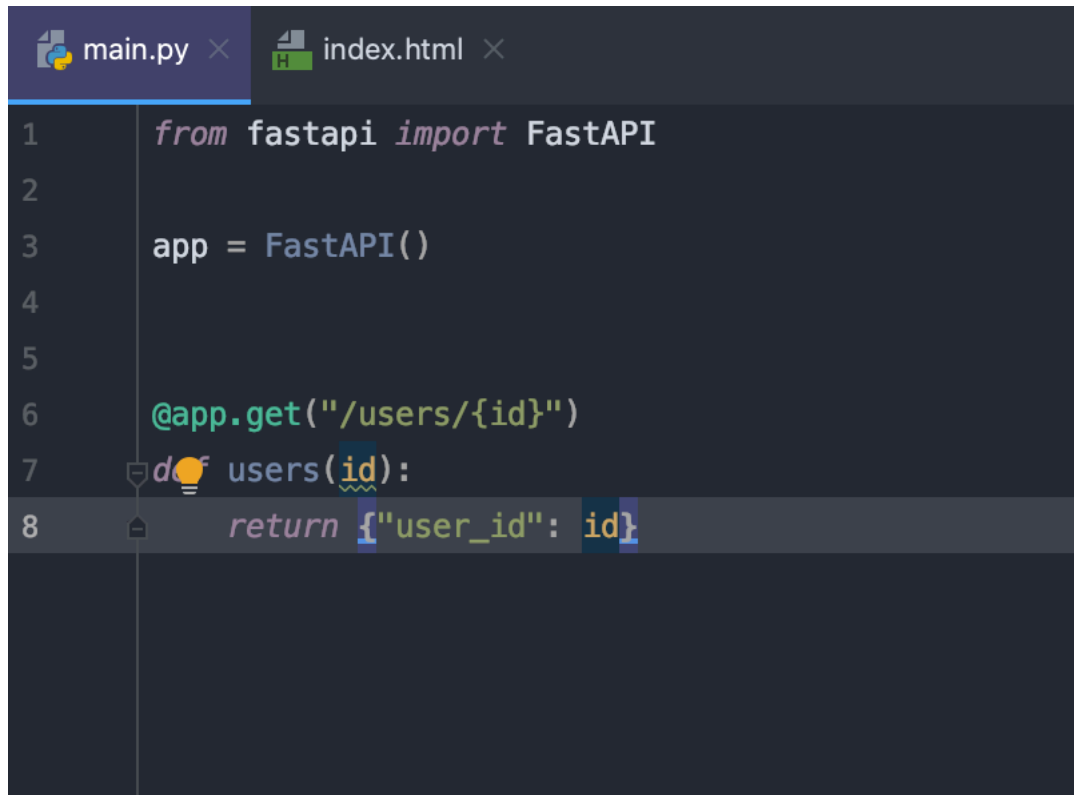
В файле `main.py` определим код для отправки этого файла клиенту:



## Task 5 - Параметры пути

Путь запроса может содержать специальные значения, которые фреймворк FastAPI может связать с параметрами функции обработчика запроса. Это так называемые параметры пути (path parameter). Благодаря параметрам пути можно передавать в приложения некоторые значения.

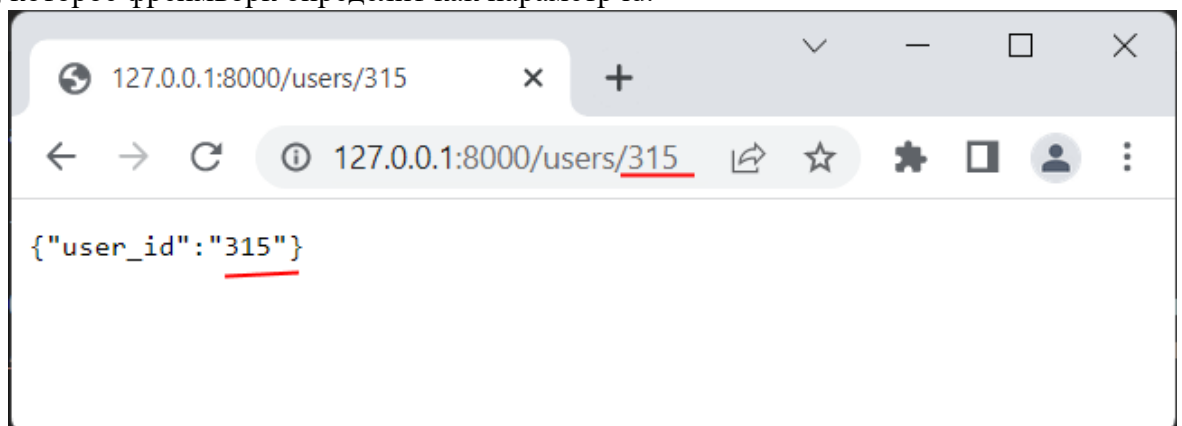
Параметры пути определяются в шаблоне пути внутри фигурных скобок: {название\_параметра}. Например, определим следующее приложение:



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5
6 @app.get("/users/{id}")
7 def users(id):
8     return {"user_id": id}
```

в данном случае декоратор `app.get` использует шаблон пути `"/users/{id}"`. Здесь `id` представляет параметр пути.

Функция `users()`, которая обрабатывает запрос по этому пути, принимает одноименный параметр. Через этот параметр в функцию будет передаваться значения из пути, которое фреймворк определит как параметр `id`.



Так, на скриншоте видно, что браузер отправляет запрос по адресу `http://127.0.0.1:8000/users/315`, то есть путь запроса в данном случае

представляет users/315. Фреймворк FastAPI видит, что этот путь соответствует шаблону "/users/{id}", поэтому данный запрос будет обрабатываться функцией users.

Параметр id в шаблоне пути составляет второй сегмент адреса. Соответственно фреймворк сможет сопоставить сегмент "315" с параметром id. Поэтому при обращении по адресу http://127.0.0.1:8000/users/315 параметр id в функции users получит значение 315.

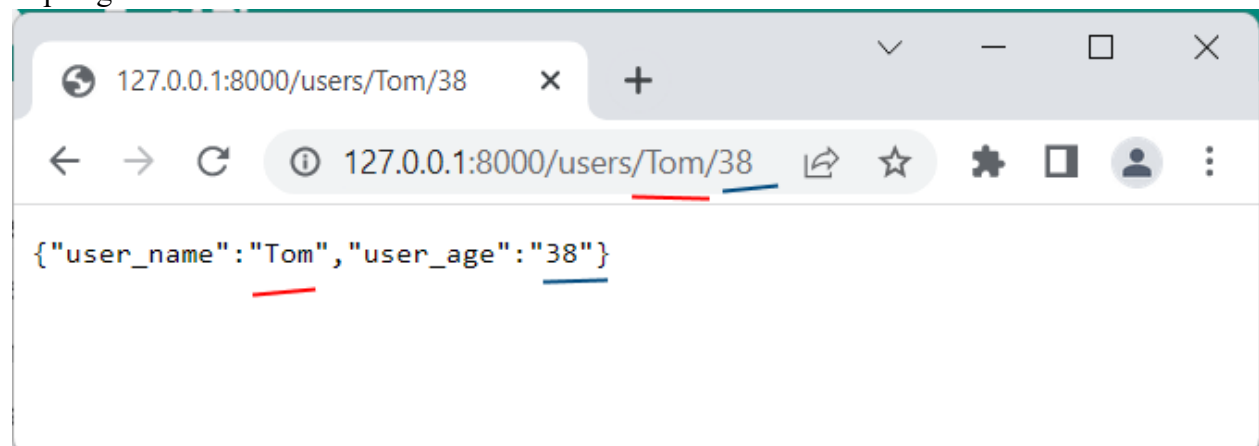
Подобным образом можно использовать и большее количество параметров:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{name}/{age}")
def users(name, age):
    return {"user_name": name, "user_age": age}
```

В данном случае функция users обрабатывает запросы, путь которых соответствует шаблону "/users/{name}/{age}". Подобный путь должен состоять из трех сегментов, где второй сегмент представляет значение параметра name, а третий сегмент - значение параметра age.



Обычно если в пути передается несколько параметров, то разделителем между ними, как правило, служит слеш, который закрывает сегмент, как в примере выше. Однако это необязательно, например, мы можем разделить параметры с помощью дефиса:

```
@app.get("/users/{name}-{age}")
def users(name, age):
    return {"user_name": name, "user_age": age}
```

Такому шаблону соответствовал бы запрос по пути http://127.0.0.1:8000/users/Tom-38, и фреймворк FastAPI автоматически распарсил бы запрошенный путь и разделил бы его последний сегмент на параметры name и age.

Очередность определения путей

Однако при определении шаблонов путей следует учитывать, что между различными шаблонами может возникнуть двойственность, когда запрос соответствует нескольким определенным шаблонам. И в этой связи следует учитывать очередность определения шаблонов путей. Например:

```
from fastapi import FastAPI

app = FastAPI()

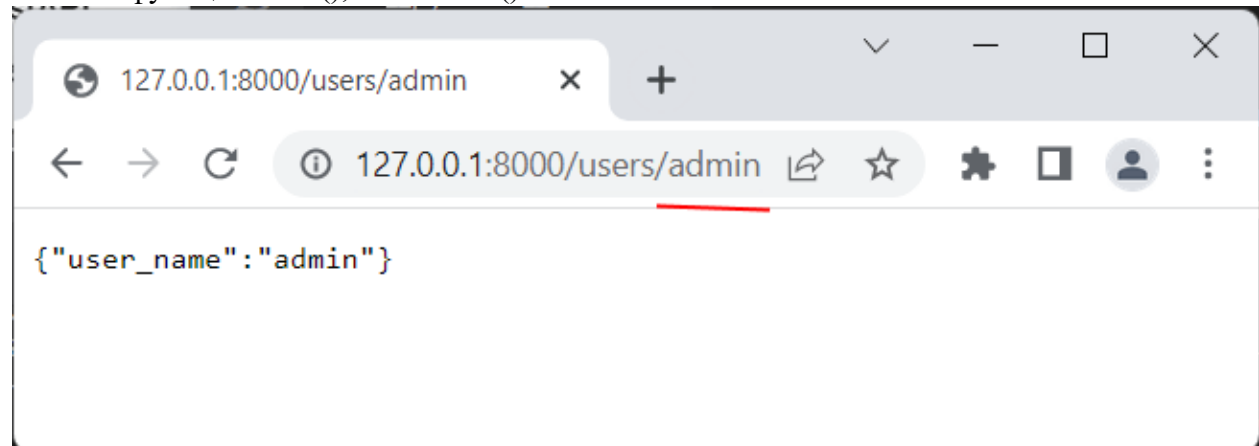
@app.get("/users/{name}")
def users(name):
    return {"user_name": name}
```

```

10 @app.get("/users/admin")
11 def admin():
12     return {"message": "Hello admin"}

```

В данном случае мы хотим, чтобы запросы по пути `/users/admin` обрабатывала функция `admin()`. А остальные пути по шаблону `/users/{name}`, где второй сегмент представляет параметр `name`, обрабатывала бы функция `users()`. Однако если мы обратимся к приложению с запросом `http://127.0.0.1:8000/users/admin`, то мы увидим, что запрос обрабатывает функция `users()`, а не `admin()`:



Потому что функция `users` определена до функции `admin`, соответственно функция `users` и будет обрабатывать данный запрос. Чтобы добиться нужного результата, нам надо поменять определение функций местами:

```

1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/users/admin")
6 def admin():
7     return {"message": "Hello admin"}
8
9
10 @app.get("/users/{name}")
11 def users(name):
12     return {"user_name": name}

```

Ограничения типа параметров

FastAPI позволяет ограничить тип параметров и соответственно набор используемых значений. Например, мы хотим, чтобы параметр `id` представлял только целое число:

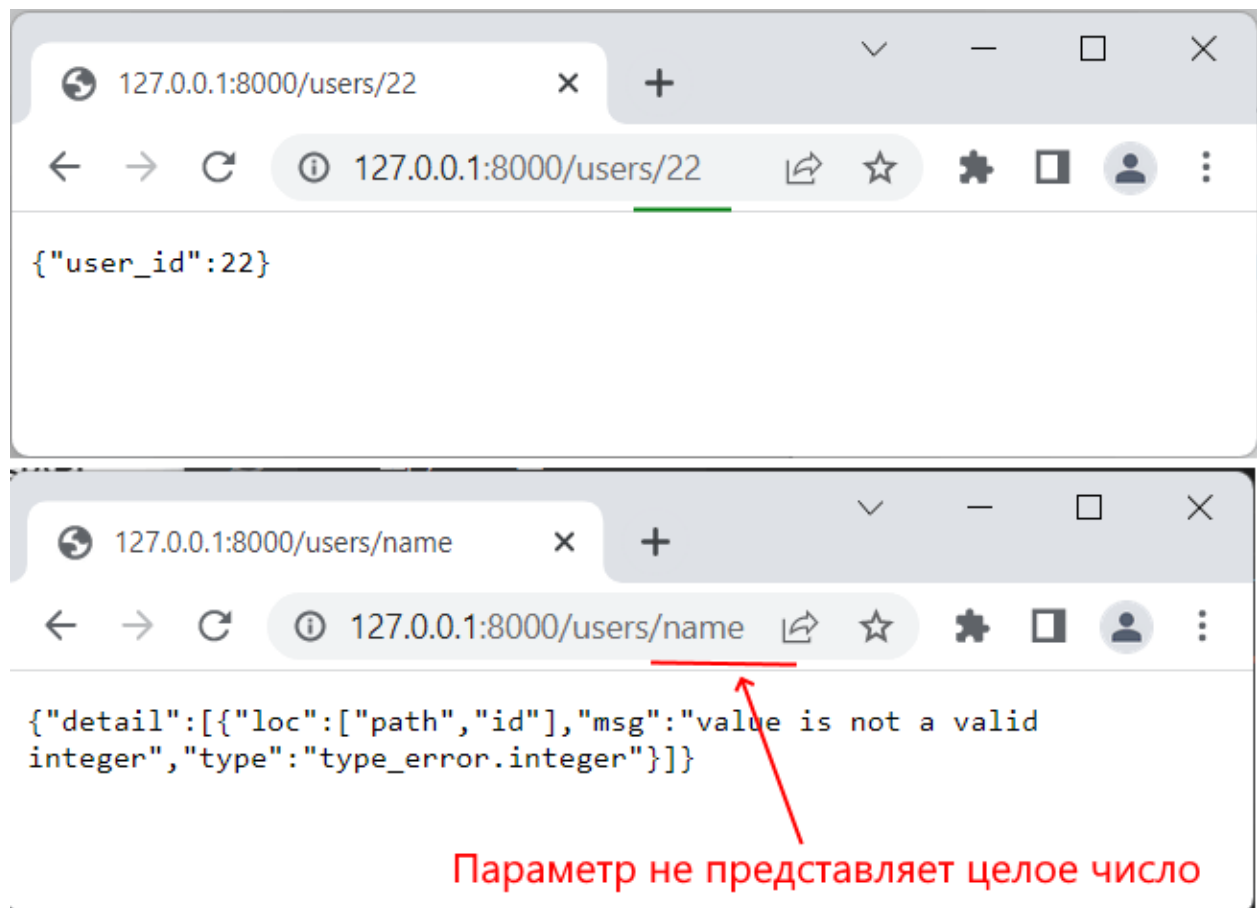
```

1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/users/{id}")
6 def users(id: int):
7     return {"user_id": id}

```

Чтобы указать, что параметр - целое число, у параметра функции явным образом задается тип `int`. И если мы попробуем передать этому параметру не целочисленное значение, то сервер отправит сообщение об ошибке:





Подобным образом в качестве ограничения можно использовать и другие типы: str, float, bool и ряд других.

Path

Дополнительно для работы с параметрами пути фреймворк FastAPI предоставляет класс **Path** из пакета fastapi. Класс Path позволяет валидировать значения параметров. В частности, через конструктор Path можно установить следующие параметры для валидации значений:

**min\_length**: устанавливает минимальное количество символов в значении параметра

**max\_length**: устанавливает максимальное количество символов в значении параметра

**regex**: устанавливает регулярное выражение, которому должно соответствовать значение параметра

**lt**: значение параметра должно быть меньше определенного значения

**le**: значение параметра должно быть меньше или равно определенному значению

**gt**: значение параметра должно быть больше определенного значения

**ge**: значение параметра должно быть больше или равно определенному значению

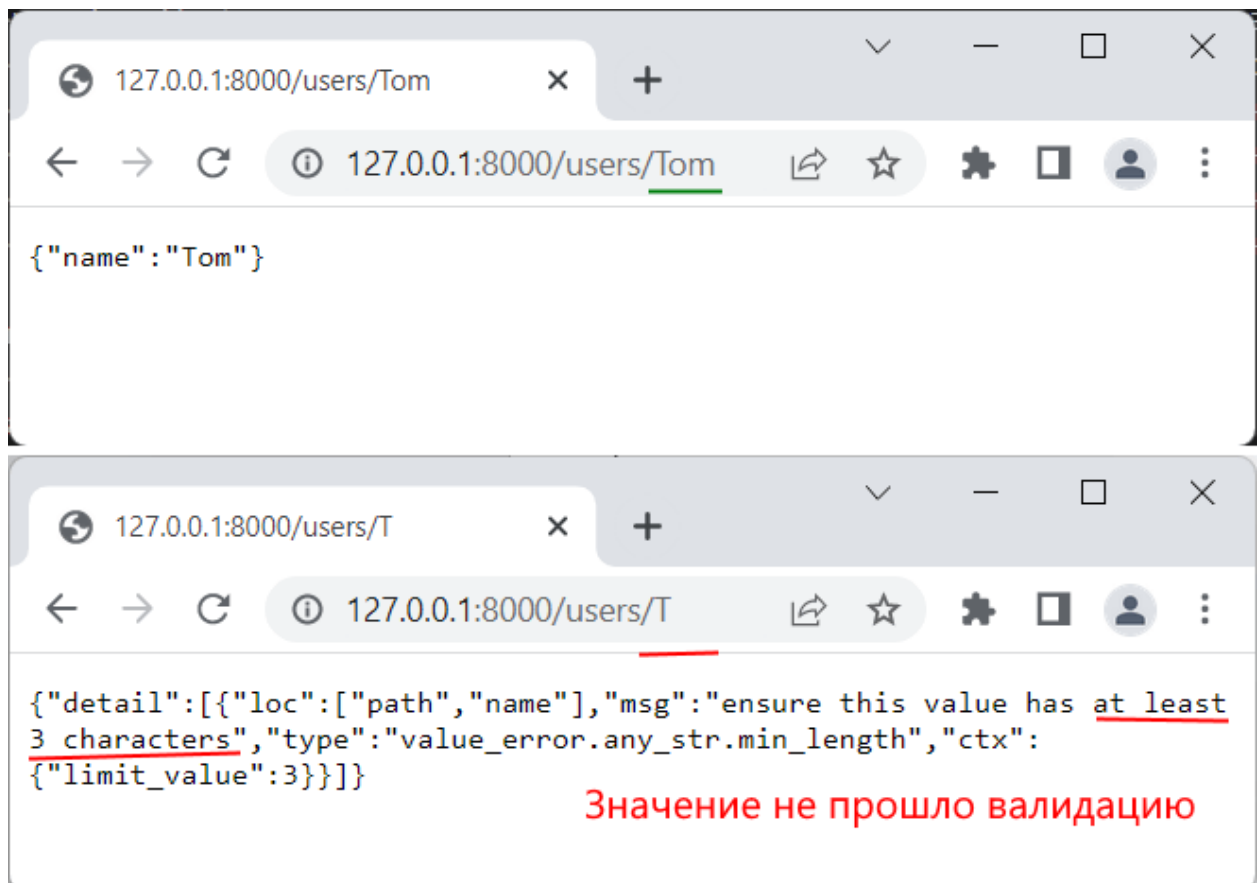
Применим некоторые параметры:

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/users/{name}")
def users(name:str = Path(min_length=3, max_length=20)):
    return {"name": name}
```

В данном случае получаем параметр name. Причем его значение должно иметь не меньше 3 и не больше 20 символов.



Подобным образом можно использовать другие параметры валидации:

```
1 from fastapi import FastAPI, Path
2
3 app = FastAPI()
4
5 @app.get("/users/{name}/{age}")
6 def users(name:str = Path(min_length=3, max_length=20),
7           age: int = Path(ge=18, lt=111)):
8     return {"name": name, "age": age}
```

В данном случае добавляется параметр "age", который должен представлять число в диапазоне от 18 (включительно) до 111 (не включая)

Валидация с помощью регулярного значения:

```
1 from fastapi import FastAPI, Path
2
3 app = FastAPI()
4
5 @app.get("/users/{phone}")
6 def users(phone:str = Path(regex="^\d{11}$")):
7     return {"phone": phone}
```

Здесь параметр phone должен состоять из 11 цифр.

## Task 6 - Параметры строки запроса

Параметры строки запроса представляют еще один способ передать в приложение некоторые значения в запросе типа GET. Для начала надо понимать, что такое **строка запроса**. Например, возьмем следующий адрес

```
1 http://127.0.0.1:8000/users/add?name=Tom&age=38
```

Здесь та часть, которая идет после адреса сервера и порта и до вопросительного знака ?, то есть users/add, представляет путь запроса (path). А та часть, которая идет **после** вопросительного знака, то есть name=Tom&age=38, представляет строку запроса (query string). В данной статье нас будет интересовать прежде всего строка запроса.

Строка запроса состоит из параметров. Каждый параметр определяется в форме

```
1 имя_параметра=значение_параметра
```

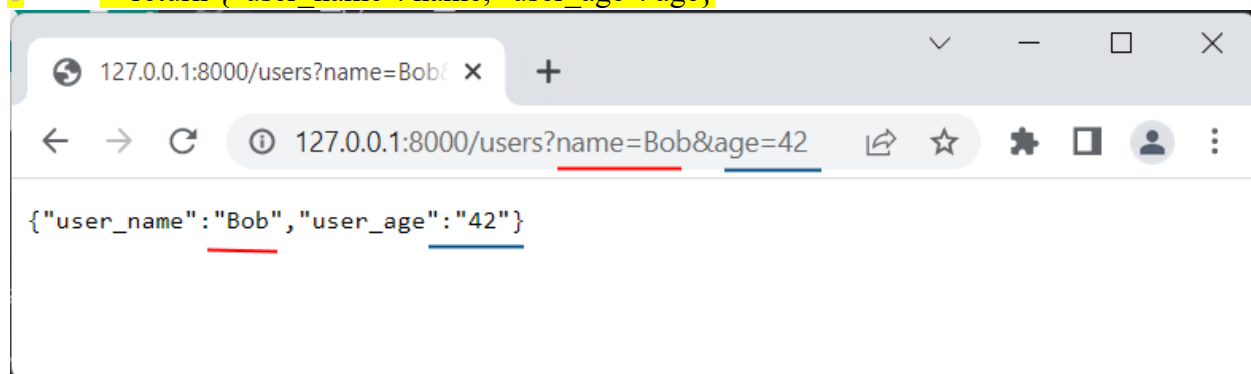
Если строка запроса содержит несколько параметров, то они отделяются друг от друга знаком амперсанда &. Так, в примере в адресом http://127.0.0.1:8000/users/add?name=Tom&age=38 строка запроса состоит из двух параметров: параметр name имеет значение "Tom", а параметр age имеет значение 38.

Для получения значений параметров строки запроса мы можем в функции определить одноименные параметры:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users")
def get_model(name, age):
    return {"user_name": name, "user_age": age}
```



### Значения по умолчанию

Вполне может быть, что при обращении к приложению пользователь не передаст значения для какого-либо параметра или даже для всех параметров строки запроса. В примере выше все параметры строки запроса являются обязательными. И если мы не передадим хотя бы один из параметров, то мы получим ошибку.

Чтобы ошибки не было, можно задать для параметров значения по умолчанию:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users")
def get_model(name = "Undefined", age = 18):
    return {"user_name": name, "user_age": age}
```

Параметры со значению по умолчанию должны идти после обязательных параметров.

## Ограничения по типу

Также для параметров строки запроса можно задать ограничения по типу:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users")
def get_model(name: str, age: int = 18):
    return {"user_name": name, "user_age": age}
```

В данном случае параметр `name` должен представлять тип `str`, то есть строку, а параметр `age` - целое число. Если параметру `age` передать нечисловое значение, то мы получим ошибку.

## Query

Дополнительно для работы с параметрами строки запроса фреймворк предоставляет класс **Query** из пакета `fastapi`. Класс **Query** позволяет прежде всего валидировать значения параметров строки запроса. В частности, через конструктор **Query** можно установить следующие параметры для валидации значений:

**min\_length**: устанавливает минимальное количество символов в значении параметра

**max\_length**: устанавливает максимальное количество символов в значении параметра

**regex**: устанавливает регулярное выражение, которому должно соответствовать

значение параметра

**lt**: значение параметра должно быть меньше определенного значения

**le**: значение параметра должно быть меньше или равно определенному значению

**gt**: значение параметра должно быть больше определенного значения

**ge**: значение параметра должно быть больше или равно определенному значению

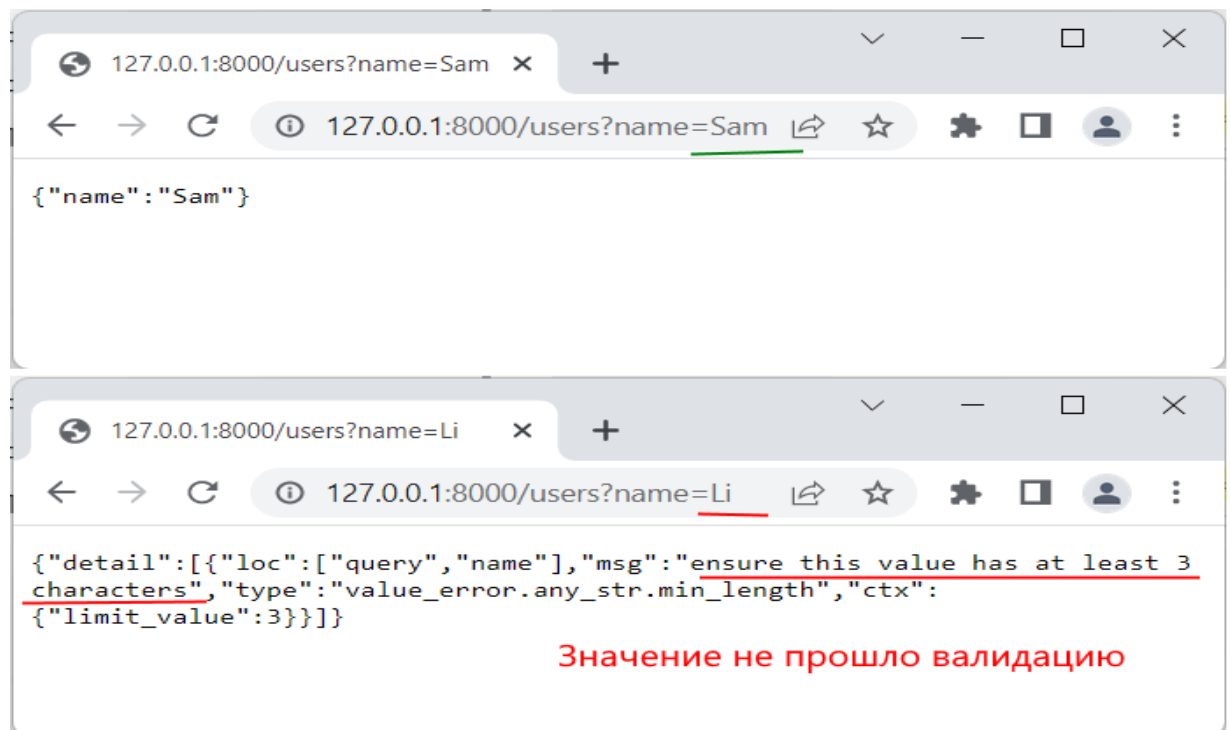
Применим некоторые параметры:

```
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/users")
def users(name: str = Query(min_length=3, max_length=20)):
    return {"name": name}
```

В данном случае через строку запроса получаем параметр `name`. Причем его значение должно иметь не меньше 3 и не больше 20 символов.



Подобным образом можно использовать другие параметры валидации:

```
1 from fastapi import FastAPI, Query
2
3 app = FastAPI()
4
5 @app.get("/users")
6 def users(name: str = Query(min_length=3, max_length=20),
7     age: int = Query(ge=18, lt=111)):
8     return {"name": name, "age": age}
```

В данном случае добавляется параметр "age", который должен представлять число в диапазоне от 18 (включительно) до 111 (не включая)

Валидация с помощью регулярного значения:

```
1 from fastapi import FastAPI, Query
2
3 app = FastAPI()
4
5 @app.get("/users")
6 def users(phone: str = Query(regex="^\d{11}$")):
7     return {"phone": phone}
```

Здесь параметр phone должен состоять из 11 цифр.

Query позволяет установить значение по умолчанию с помощью параметра default:

```
1 from fastapi import FastAPI, Query
2
3 app = FastAPI()
4
5 @app.get("/users")
6 def users(name: str = Query(default="Undefined", min_length=2)):
7     return {"name": name}
```

Здесь, если в запрошенном адресе отсутствует параметр name, то по умолчанию он будет равен строке "Undefined"

Если параметры должны быть необязательными, то параметру default передается значение None:

```

1 from fastapi import FastAPI, Query
2
3 app = FastAPI()
4
5 @app.get("/users")
6 def users(name: str | None = Query(default=None, min_length=2)):
7     if name==None:
8         return {"name": "Undefined"}
9     else:
10        return {"name": name}

```

Получение списков значений

Использование класса Query позволяет получать через строку запроса списки. В общем случае списки значений передаются, когда в строке запроса одному параметру несколько раз передаются разные значения. Например, как в запросе по следующему адресу:

1 <http://127.0.0.1:8000/users?people=tom&people=Sam&people=Bob>

Здесь параметру people передаются три разных значения, соответственно мы ожидаем, что список people будет содержать три элемента.

Определим следующее приложение:

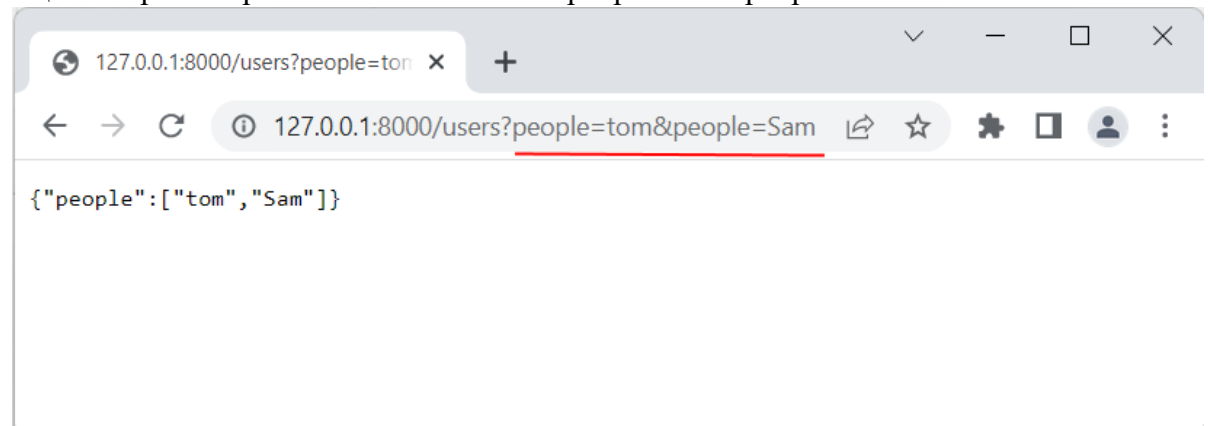
```

1 from fastapi import FastAPI, Query
2
3 app = FastAPI()
4
5 @app.get("/users")
6 def users(people: list[str] = Query()):
7     return {"people": people}

```

Здесь функция users имеет один параметр - people, который должен представлять список строк - тип list[str].

Передадим через строку запроса список значений в виде параметра people, например, с помощью запроса <http://127.0.0.1:8000/users?people=tom&people=Sam>:



Сочетание параметров пути и строки запроса

При необходимости можно сочетать параметры пути и строки запроса:

```

1 from fastapi import FastAPI, Path, Query
2
3 app = FastAPI()
4
5 @app.get("/users/{name}")
6 def users(name: str = Path(min_length=3, max_length=20),
7           age: int = Query(ge=18, lt=111)):
8     return {"name": name, "age": age}

```

В данном случае параметр `name` представляет параметр пути, а `age` - параметр строки запроса. И в данном случае мы могли бы обратиться к функции `users`, например, посредством адреса <http://127.0.0.1:8000/users/Tom?age=38>

### Task 7 - Отправка статусных кодов

Одной из распространенных задач в веб-приложении является отправка статусных кодов, которые указывают на статус выполнения операции на сервере.

**1xx:** предназначены для информации. Ответ с таким кодом не может иметь содержимого

**2xx:** указывает на успешное выполнение операции

**3xx:** предназначены для переадресации

**4xx:** предназначены для отправки информации об ошибках клиента

**5xx:** предназначены для информации об ошибках сервера

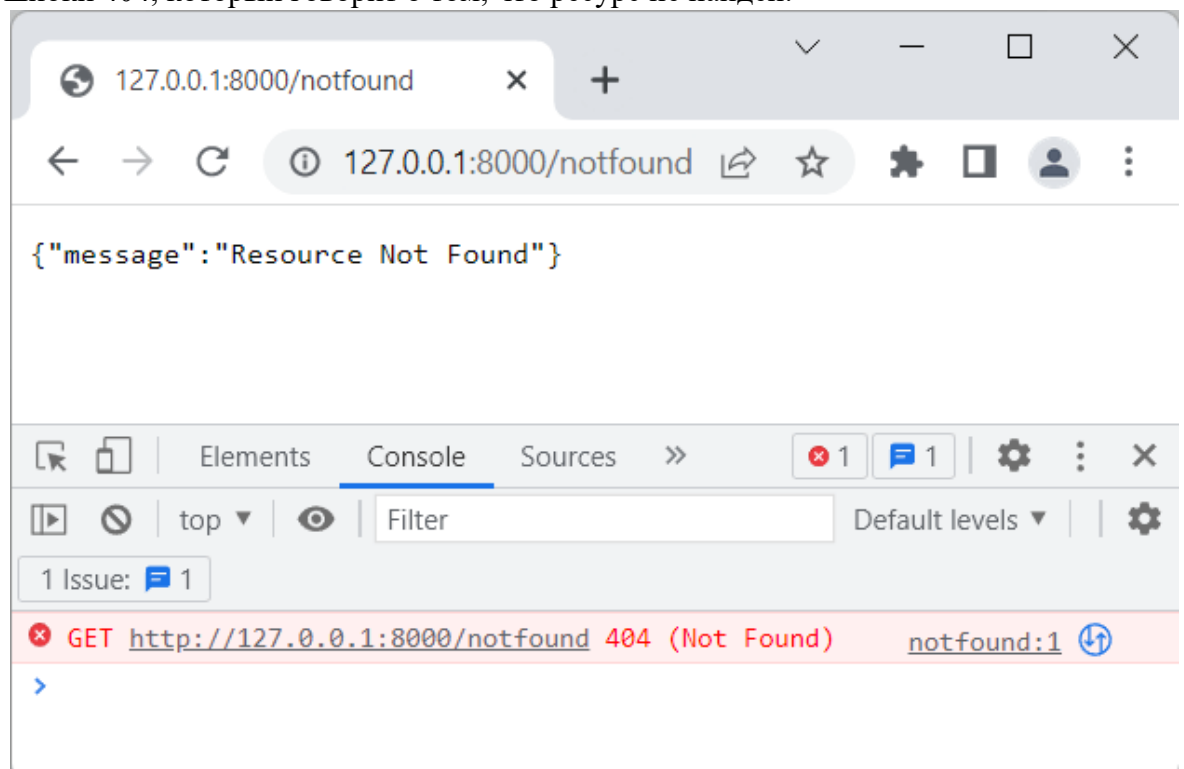
По умолчанию функции обработки отправляют статусный код 200, но при необходимости мы можем отправить любой статусный код. Для этого у методов `get()`, `post()`, `put()`, `delete()`, `options()`, `head()`, `patch()`, `trace()` в классе `FastAPI` применяется параметр `status_code`, который принимает числовой код статуса HTTP. Например:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/notfound", status_code=404)
def notfound():
    return {"message": "Resource Not Found"}
```

В данном случае при обращении по пути `"/notfound"` клиенту отправляется статусный код ошибки 404, который говорит о том, что ресурс не найден.



Для упрощения в FastAPI есть модуль **status**, в котором определены константы для представления статусных кодов:

- HTTP\_100\_CONTINUE (код 100)
- HTTP\_101\_SWITCHING\_PROTOCOLS (код 101)
- HTTP\_102\_PROCESSING (код 102)
- HTTP\_103\_EARLY\_HINTS (код 103)
- HTTP\_200\_OK (код 200)
- HTTP\_201\_CREATED (код 201)
- HTTP\_202\_ACCEPTED (код 202)
- HTTP\_203\_NON\_AUTHORITATIVE\_INFORMATION (код 203)
- HTTP\_204\_NO\_CONTENT (код 204)
- HTTP\_205\_RESET\_CONTENT (код 205)
- HTTP\_206\_PARTIAL\_CONTENT (код 206)
- HTTP\_207\_MULTI\_STATUS (код 207)
- HTTP\_208\_ALREADY\_REPORTED (код 208)
- HTTP\_226\_IM\_USED (код 226)
- HTTP\_300\_MULTIPLE\_CHOICES (код 300)
- HTTP\_301\_MOVED\_PERMANENTLY (код 301)
- HTTP\_302\_FOUND (код 302)
- HTTP\_303\_SEE\_OTHER (код 303)
- HTTP\_304\_NOT\_MODIFIED (код 304)
- HTTP\_305\_USE\_PROXY (код 305)
- HTTP\_306\_RESERVED (код 306)
- HTTP\_307\_TEMPORARY\_REDIRECT (код 307)
- HTTP\_308\_PERMANENT\_REDIRECT (код 308)
- HTTP\_400\_BAD\_REQUEST (код 400)
- HTTP\_401\_UNAUTHORIZED (код 401)
- HTTP\_402\_PAYMENT\_REQUIRED (код 402)
- HTTP\_403\_FORBIDDEN (код 403)
- HTTP\_404\_NOT\_FOUND (код 404)
- HTTP\_405\_METHOD\_NOT\_ALLOWED (код 405)
- HTTP\_406\_NOT\_ACCEPTABLE (код 406)
- HTTP\_407\_PROXY\_AUTHENTICATION\_REQUIRED (код 407)
- HTTP\_408\_REQUEST\_TIMEOUT (код 408)
- HTTP\_409\_CONFLICT (код 409)
- HTTP\_410\_GONE (код 410)
- HTTP\_411\_LENGTH\_REQUIRED (код 411)
- HTTP\_412\_PRECONDITION\_FAILED (код 412)
- HTTP\_413\_REQUEST\_ENTITY\_TOO\_LARGE (код 413)
- HTTP\_414\_REQUEST\_URI\_TOO\_LONG (код 414)
- HTTP\_415\_UNSUPPORTED\_MEDIA\_TYPE (код 415)
- HTTP\_416\_REQUESTED\_RANGE\_NOT\_SATISFIABLE (код 416)
- HTTP\_417\_EXPECTATION\_FAILED (код 417)
- HTTP\_418\_IM\_A\_TEAPOT (код 418)
- HTTP\_421\_MISDIRECTED\_REQUEST (код 421)
- HTTP\_422\_UNPROCESSABLE\_ENTITY (код 422)
- HTTP\_423\_LOCKED (код 423)
- HTTP\_424\_FAILED\_DEPENDENCY (код 424)
- HTTP\_425\_TOO\_EARLY (код 425)
- HTTP\_426\_UPGRADE\_REQUIRED (код 426)
- HTTP\_428\_PRECONDITION\_REQUIRED (код 428)
- HTTP\_429\_TOO\_MANY\_REQUESTS (код 429)



HTTP\_431\_REQUEST\_HEADER\_FIELDS\_TOO\_LARGE (код 431)  
HTTP\_451\_UNAVAILABLE\_FOR\_LEGAL\_REASONS (код 451)  
HTTP\_500\_INTERNAL\_SERVER\_ERROR (код 500)  
HTTP\_501\_NOT\_IMPLEMENTED (код 501)  
HTTP\_502\_BAD\_GATEWAY (код 502)  
HTTP\_503\_SERVICE\_UNAVAILABLE (код 503)  
HTTP\_504\_GATEWAY\_TIMEOUT (код 504)  
HTTP\_505\_  
HTTP\_VERSION\_NOT\_SUPPORTED (код 505)  
HTTP\_506\_VARIANT\_ALSO\_NEGOTIATES (код 506)  
HTTP\_507\_INSUFFICIENT\_STORAGE (код 507)  
HTTP\_508\_LOOP\_DETECTED (код 508)  
HTTP\_510\_NOT\_EXTENDED (код 510)  
HTTP\_511\_NETWORK\_AUTHENTICATION\_REQUIRED (код 511)

#### Пример использования

```
1 from fastapi import FastAPI
2
3
4 app = FastAPI()
5
6 @app.get("/notfound", status_code=status.HTTP_404_NOT_FOUND)
7 def notfound():
8     return {"message": "Resource Not Found"}
```

Определение статусного кода в ответе

В примере выше функция вне зависимости от данных запроса или каких-то других условий в любом случае возвращала статусный код 404. Однако чаще бывает необходимо возвращать статусный код в зависимости от некоторых условий. В этом случае мы можем использовать параметр `status_code` конструктора класса `Response` или его наследников:

```
1 from fastapi import FastAPI
2 from fastapi.responses import JSONResponse
3
4 app = FastAPI()
5
6 @app.get("/notfound")
7 def notfound():
8     return JSONResponse(content={"message": "Resource Not Found"}, status_code=404)
```

Изменение статусного кода

Можно комбинировать оба подхода:

```
1 from fastapi import FastAPI, Response, Path
2
3 app = FastAPI()
4
5 @app.get("/users/{id}", status_code=200)
6 def users(response: Response, id: int = Path()):
7     if id < 1:
8         response.status_code = 400
9         return {"message": "Incorrect Data"}
10    return {"message": f"Id = {id}"}
```

В данном случае если параметр пути меньше 1, то условно считаем, что переданные некорректные данные, и отправляем в ответ статусный код 400 (Bad Request)

## Task 8 - Переадресация

Для переадресации в приложении FastAPI применяется класс **RedirectResponse** (класс-наследник от **Response**). В качестве обязательного параметра конструктор **RedirectResponse** принимает адрес для перенаправления:

```
1 import mimetypes
2 from fastapi import FastAPI
3 from fastapi.responses import RedirectResponse, PlainTextResponse
4
5 app = FastAPI()
6
7 @app.get("/old")
8 def old():
9     return RedirectResponse("/new")
10
11 @app.get("/new")
12 def new():
13     return PlainTextResponse("Новая страница")
```

В данном случае при обращении по пути `/old` происходит перенаправление по пути `/new`. Альтернативный вариант:

```
1 @app.get("/old", response_class= RedirectResponse)
2 def old():
3     return "/new"
```

Также можно перенаправлять по абсолютному адресу:

```
1 @app.get("/old")
2 def old():
3     return RedirectResponse("https://metanit.com/python/fastapi")
```

По умолчанию **RedirectResponse** отправляет статусный код 307 (временная переадресация). Если такое положение не устраивает, то можно задать статусный код переадресации с помощью параметра `status_code`:

```
1 @app.get("/old")
2 def old():
3     return RedirectResponse("/new", status_code=302)
```

или так

```
1 @app.get("/old", response_class= RedirectResponse, status_code=302)
2 def old():
3     return "/new"
```

## Task 9 - Статические файлы

Для определения интерфейса для работы с сервером нередко используются html-страницы, то есть статические файлы с кодом html, которые могут использовать какие-то другие статические файлы - файлы стилей css, изображений, скриптов javascript и т.д. Для работы со статическими файлами FastAPI предоставляет удобный и компактный функционал, который располагается в пакете `fastapi.staticfiles`. В частности, для обслуживания статических файлов в определенном каталоге применяется класс `StaticFiles`, конструктор которого имеет следующую сигнатуру:

```
StaticFiles(directory=None, packages=None, html=False, check_dir=True)
```

Используемые параметры:

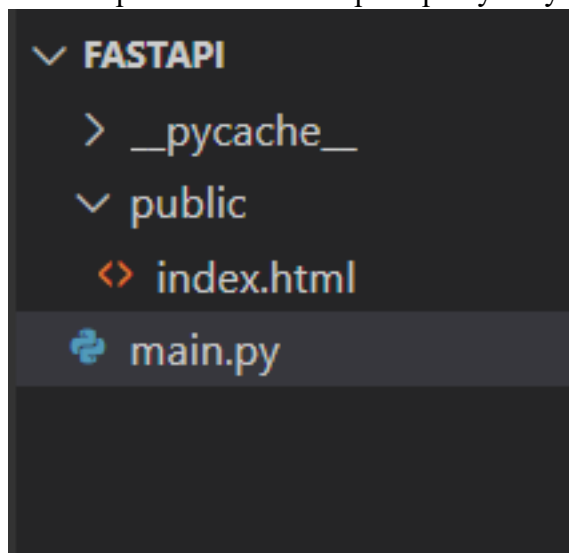
`directory`: путь к каталогу со статическими файлами

`packages`: список пакетов python в виде списка строк или кортежей строк

`html`: устанавливает запуск в HTML-режиме, когда при обращении к корню каталога автоматически загружается файл `index.html` (при наличии такого файла)

`check_dir`: гарантирует, что каталог со статическими файлами существует

Рассмотрим небольшой пример. Пусть у нас будет следующий проект:



В проекте определен каталог `public`, который предназначен для хранения статических файлов. И определим в этом каталоге простейший файл `index.html` со следующим кодом:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Astana IT University</title>
5   <meta charset="utf-8" />
6 </head>
7 <body>
8   <h2>Hello Students!</h2>
9 </body>
10 </html>
```

В файле `main.py` для обслуживания статических файлов из каталога `public` определим следующий код:

```
1 from fastapi import FastAPI
2 from fastapi.staticfiles import StaticFiles
3
4 app = FastAPI()
5
6 app.mount("/static", StaticFiles(directory="public"))
```

Для работы со статическими файлами вначале импортируем класс StaticFiles. Затем создаем объект приложения FastAPI и вызываем у него метод mount().

Метод mount() устанавливает объект ASGIApp0 - обработчик запросов по определенному пути. В данном случае для запросов по пути "/static" в качестве обработчика запросов выступает объект StaticFiles, в котором с помощью параметра directory в качестве каталога статических файлов устанавливается каталог "/public" (название каталога произвольное).

То есть при обращении по пути "/static" приложение будет посылать в ответ файлы из каталога "public".

Запустим приложение и обратимся по пути `http://127.0.0.1:8000/static/index.html`, и приложение в ответ пришлет нам файл `index.html`.

Подобным образом мы можем добавлять в каталог public и другие статические файлы.

### **Установка главной страницы**

В примере выше для обращения к файлу `index.html` Нередко веб-приложение имеет некоторую главную страницу. Например, когда мы обращаемся к корню некоторых сайтов или к корню их отдельных каталогов, веб-сервер присылает главную страницу этого сайта или каталога. Класс StaticFiles также позволяет сделать подобное с помощью передачи параметру `html` значения `True` (значение по умолчанию - `False`). В этом случае, если в пути не указывается имя файла, то сервер автоматически отправляет файл `index.html` (при его наличии). Например, изменим код main.py следующим образом:

```
1 from fastapi import FastAPI
2 from fastapi.staticfiles import StaticFiles
3
4 app = FastAPI()
5
6 app.mount("/static", StaticFiles(directory="public", html=True))
```

Теперь мы можем обратиться по пути `http://127.0.0.1:8000/static/`, и сервер также пришлет нам страницу `index.html`.

Подобным образом можно установить главную страницу и для всего веб-приложения:

```
1 from fastapi import FastAPI
2 from fastapi.staticfiles import StaticFiles
3
4 app = FastAPI()
5
6 app.mount("/", StaticFiles(directory="public", html=True))
```

## Task 10- Получение данных запроса

В запросе могут передаваться различные данные, например, через отправку каких-то значений в формате json. Рассмотрим, как получать подобные данные.

### Body

Для получения данных из тела запроса можно использовать класс Body из пакета fastapi. Данный класс позволяет связать с параметром функции-обработчика запроса либо все тело запроса, либо какие-то отдельные его значения. Для примера для упрощения отправки данных определим в проекте папку public, в которой создадим новый файл index.html со следующим кодом:

```
<!DOCTYPE html>
<html>
<head>
  <title>Astana IT University</title>
  <meta charset="utf-8" />
</head>
<body>
  <div id="message"></div>
  <p>
    Введите имя: <br />
    <input name="username" id="username" />
  </p>
  <p>
    Введите возраст: <br />
    <input name="userage" id="userage" type="number" />
  </p>
  <button onclick="send()">Отправить</button>
<script>
  async function send(){
    // получаем введенное в поле имя и возраст
    const username = document.getElementById("username").value;
    const userage = document.getElementById("userage").value;
    // отправляем запрос
    const response = await fetch("/hello", {
      method: "POST",
      headers: { "Accept": "application/json", "Content-Type": "application/json" },
      body: JSON.stringify({
        name: username,
        age: userage
      })
    });
    if (response.ok) {
      const data = await response.json();
      document.getElementById("message").textContent = data.message;
    }
  }
```

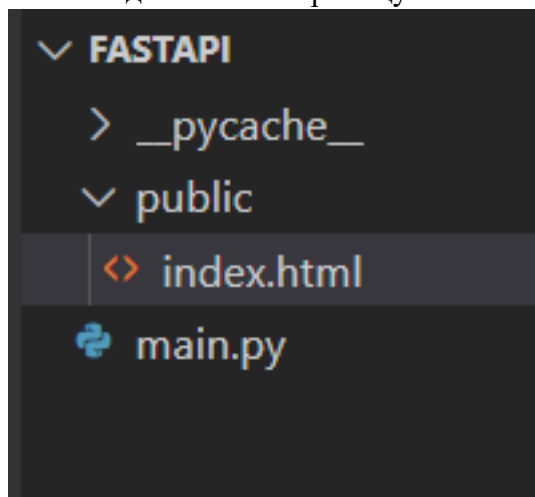
```

else
    console.log(response);
}
</script>
</body>
</html>

```

В данном случае для ввода данных на веб-странице определено текстовое поле, в которое пользователь вводит свое имя. По нажатию на кнопку срабатывает функция `send()`, определенная в коде `javascript`. Эта функция получает введенное значение и с помощью функции `fetch` отправляет его по адресу `"/hello"` в запросе типа `POST`. При этом в теле запроса посылаются сериализованные в `JSON` данные. В частности, посылается объект, в котором свойство `name` хранит введенное имя.

После отправки запроса мы ожидаем, что сервер в ответ пришлет нам некоторые данные - некоторый объект `JSON`, в котором будет свойство `"message"`. И значение этого свойства выводим на веб-страницу в элемент с `id=message`.



Затем определим основной файл приложения `main.py`, который будет получать и обрабатывать запросы:

```

from fastapi import FastAPI, Body
from fastapi.responses import FileResponse

app = FastAPI()

@app.get("/")
def root():
    return FileResponse("public/index.html")

@app.post("/hello")
#def hello(name = Body(embed=True)):
def hello(data = Body()):
    name = data["name"]
    age = data["age"]
    return {"message": f"{name}, ваш возраст - {age}"}

```

Здесь при обращении по пути "/" клиенту будет отправляться страница **index.html** для ввода данных.

Для обработки полученных в POST-запросе данных по адресу "/hello" определена функция hello(). Эта функция имеет один параметр - data, который получает содержимое тела запроса:

```
data = Body()
```

То есть здесь data будет представлять весь объект, который отправляется с веб-страницы и который имеет свойства "name" и "age". Этот объект в python будет представлять словарь. Соответственно, чтобы получить значения свойства "name", обращаемся по одноименному ключу:

```
name = data["name"]
```

Подобным образом получаем значение свойства "age". Затем в ответ клиенту посылается словарь с элементом "message".



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000". The page content includes a form with two input fields. The first field is labeled "Введите имя:" and contains the text "Tom". The second field is labeled "Введите возраст:" and contains the number "22". Below these fields is a button labeled "Отправить".

Получение отдельных значений

В примере выше мы получали все данные из тела запроса в один параметр. Однако, установив параметр `embed=True`, можно получать отдельные значения:

```
1 from fastapi import FastAPI, Body
2 from fastapi.responses import FileResponse
3
4 app = FastAPI()
5
6 @app.get("/")
7 def root():
8     return FileResponse("public/index.html")
9
10 @app.post("/hello")
11 def hello(name = Body(embed=True), age = Body(embed=True)):
12     return {"message": f"{name}, ваш возраст - {age}"}
```

### Валидация

Класс `Body` позволяет валидировать значения из тела запроса. В частности, через конструктор `Body` можно установить следующие параметры для валидации значений:

**min\_length:** устанавливает минимальное количество символов в значении параметра

**max\_length:** устанавливает максимальное количество символов в значении параметра

**regex:** устанавливает регулярное выражение, которому должно соответствовать значение параметра

**lt:** значение параметра должно быть меньше определенного значения

**le:** значение параметра должно быть меньше или равно определенному значению

**gt:** значение параметра должно быть больше определенного значения

**ge:** значение параметра должно быть больше или равно определенному значению  
Применим некоторые параметры:

```
1 from fastapi import FastAPI, Body
2 from fastapi.responses import FileResponse
3
4 app = FastAPI()
5
6 @app.get("/")
7 def root():
8     return FileResponse("public/index.html")
9
10 @app.post("/hello")
11 def hello(name: str = Body(embed=True, min_length=3, max_length=20),
12          age: int = Body(embed=True, ge=18, lt=111)):
13     return {"message": f'{name}, ваш возраст - {age}'}
```

В данном случае значение параметра name должно иметь не меньше 3 и не больше 20 символов, а параметр "age" должен представлять число в диапазоне от 18 (включительно) до 111 (не включая)

### Task 11 - Получение данных запроса в виде объекта класса

В задании 10 рассматривалось получение данных из тела запроса с помощью класса **fastapi.Body** в виде словаря или отдельных его значений. Однако FastAPI также позволяет получать данные в виде объектов своих классов. Такие классы должны быть унаследованы от класса **pydantic.BaseModel**. Такие классы определяются специально под запрос, данные которого необходимо получить.

Например, определим в файле приложения следующий код:

```
1 from fastapi import FastAPI
2 from fastapi.responses import FileResponse
3 from pydantic import BaseModel
4
5 class Person(BaseModel):
6     name: str
7     age: int
8
9 app = FastAPI()
10
11 @app.get("/")
12 def root():
13     return FileResponse("public/index.html")
14
15 @app.post("/hello")
16 def hello(person: Person):
17     return {"message": f'Привет, {person.name}, твой возраст - {person.age}'}
```

Здесь в функции hello получаем данные из тела запроса в объект класса Person. Данный класс унаследован от **BaseModel**. Класс Person определяет два атрибута, которые соответствуют данным из тела запроса, которые мы собираемся получить. В данном случае это атрибут name, который представляет строку, и атрибут age, который представляет целое число.



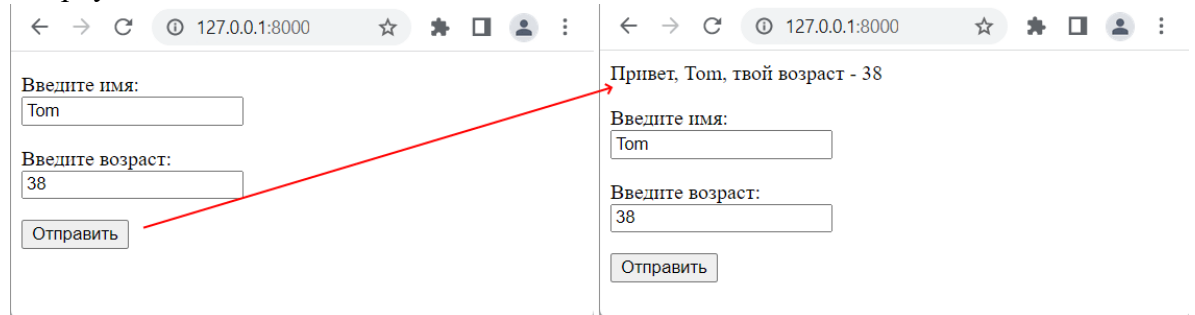
Получив данные, мы сможем работать с ними как в данными объекта Person, например, обратиться к его атрибутам name и age. В частности, в данном случае, используя эти атрибуты, формируем и отправляем клиенту некоторое сообщение.

В проекте определим папку **public**, в которой определим для тестирования веб-страницу **index.html** со следующим кодом:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Astana IT university</title>
5   <meta charset="utf-8" />
6 </head>
7 <body>
8   <div id="message"></div>
9   <p>
10     Введите имя: <br />
11     <input name="username" id="username" />
12   </p>
13   <p>
14     Введите возраст: <br />
15     <input name="userage" id="userage" type="number" />
16   </p>
17   <button onclick="send()">Отправить</button>
18 <script>
19   async function send(){
20     // получаем введенные в поля имя и возраст
21     const username = document.getElementById("username").value;
22     const userage = document.getElementById("userage").value;
23
24     // отправляем запрос
25     const response = await fetch("/hello", {
26       method: "POST",
27       headers: { "Accept": "application/json", "Content-Type": "application/json"
28     },
29     body: JSON.stringify({
30       name: username,
31       age: userage
32     })
33   });
34   if (response.ok) {
35     const data = await response.json();
36     document.getElementById("message").textContent = data.message;
37   }
38   else
39     console.log(response);
40   }
41 </script>
42 </body>
43 </html>
```

Здесь на сервер по адресу `"/hello"` в запросе типа POST отправляются введенные в поля значения. Эти значения в теле запроса имеют ключи `"name"` и `"age"` - как и атрибуты модели Person: между ключами отправляемых данных и атрибутами класса должно быть

соответствие по имени. Полученное от сервера сообщение отображается на веб-странице в блоке сверху:



### Необязательные атрибуты

В примере выше оба атрибута: и name, и age являются обязательными. А это значит, что если в запросе не будет значения хотя бы для одного из этих атрибутов, то приложение пришлет клиенту ошибку. Однако мы также можем сделать некоторые атрибуты необязательными, присвоив им значение **None**:

```
1 from fastapi import FastAPI, Body
2 from fastapi.responses import FileResponse
3 from pydantic import BaseModel
4
5 class Person(BaseModel):
6     name: str
7     age: int | None = None
8
9 app = FastAPI()
10
11 @app.get("/")
12 def root():
13     return FileResponse("public/index.html")
14
15 @app.post("/hello")
16 def hello(person: Person):
17     if person.age == None:
18         return {"message": f"Привет, {person.name}"}
19     else:
20         return {"message": f"Привет, {person.name}, твой возраст - {person.age}"}
```

В данном случае атрибут name остается обязательным, а атрибут age - необязательным, поэтому в запросе необязательно для него передавать значение.

Детальная настройка атрибутов и класс Field

Для более детальной настройки атрибутов модели применяется класс **pydantic.Field**. Например, он позволяет задать значение по умолчанию и правила валидации значений с помощью следующих параметров конструктора:

**default:** устанавливает значение по умолчанию

**min\_length:** устанавливает минимальное количество символов в значении параметра

**max\_length:** устанавливает максимальное количество символов в значении параметра

**regex:** устанавливает регулярное выражение, которому должно соответствовать значение параметра

**lt:** значение параметра должно быть меньше определенного значения

**le:** значение параметра должно быть меньше или равно определенному значению

**gt:** значение параметра должно быть больше определенного значения

**ge:** значение параметра должно быть больше или равно определенному значению

Применим некоторые параметры:

```
1 from fastapi import FastAPI
2 from fastapi.responses import FileResponse
3 from pydantic import BaseModel, Field
4
5 class Person(BaseModel):
6     name: str = Field(default="Undefined", min_length=3, max_length=20)
7     age: int = Field(default=18, ge=18, lt=111)
8
9 app = FastAPI()
10
11 @app.get("/")
12 def root():
13     return FileResponse("public/index.html")
14
15 @app.post("/hello")
16 def hello(person: Person):
17     return {"message": f'Привет, {person.name}, твой возраст - {person.age}'}
```

В данном случае значение параметра name должно иметь не меньше 3 и не больше 20 символов, а параметр "age" должен представлять число в диапазоне от 18 (включительно) до 111 (не включая). Если в запросе не переданы значения для атрибутов класса, то атрибуты name и age получают значения по умолчанию: строку "Undefined" и число 18 соответственно.

### Получение списков

Подобным образом можно получать список объектов модели:

```
1 from fastapi import FastAPI
2 from fastapi.responses import FileResponse
3 from pydantic import BaseModel
4
5 class Person(BaseModel):
6     name: str
7     age: int
8
9 app = FastAPI()
10
11 @app.get("/")
12 def root():
13     return FileResponse("public/index.html")
14
15 @app.post("/hello")
16 def hello(people: list[Person]):
17     return {"message": people}
```

В этом случае для теста мы могли бы отправить данные из кода javascript следующим образом:

```
1 const response = await fetch("/hello", {
2     method: "POST",
3     headers: { "Accept": "application/json", "Content-Type": "application/json" },
4     body: JSON.stringify([
5         { name: "Tom", age: 38 },
6         { name: "Bob", age: 41 },
7         { name: "Sam", age: 25 }
8     ])
9 })
```

```

9     });
10    const data = await response.json();
11    console.log(data);

```

### Получение вложенных списков

Модель может содержать список. Например, класс Person содержит список изучаемых языков:

```

1  from fastapi import FastAPI
2  from fastapi.responses import FileResponse
3  from pydantic import BaseModel
4
5  class Person(BaseModel):
6      name: str
7      languages: list = []
8
9  app = FastAPI()
10
11  @app.get("/")
12  def root():
13      return FileResponse("public/index.html")
14
15  @app.post("/hello")
16  def hello(person: Person):
17      return {"message": f"Name: {person.name}. Languages: {person.languages}"}

```

В данном случае для хранения языков в классе Person определен атрибут languages. В этом случае отправка данных из javascript выглядела бы следующим образом:

```

1  const response = await fetch("/hello", {
2      method: "POST",
3      headers: { "Accept": "application/json", "Content-Type": "application/json" },
4      body: JSON.stringify({
5          name: "Tom",
6          languages: ["Python", "JavaScript"]
7      })
8  });
9  const data = await response.json();
10 console.log(data); // {message: "Name: Tom. Languages: ['Python', 'JavaScript']}

```

Также у атрибута можно установить значение по умолчанию на случай, если в запросе не содержится соответствующих данных:

```

1  class Person(BaseModel):
2      name: str
3      languages: list = ["Java", "Python", "JavaScript"]

```

### Вложенные модели

Одна модель может содержать другую модель. Например, пользователь работает в какой-нибудь компании. И для хранения данных компании можно создать отдельную модель - Company:

```

1  from fastapi import FastAPI
2  from fastapi.responses import FileResponse
3  from pydantic import BaseModel
4
5  class Company(BaseModel):
6      name: str

```

```

8 class Person(BaseModel):
9     name: str
10    company: Company
11
12 app = FastAPI()
13
14 @app.get("/")
15 def root():
16     return FileResponse("public/index.html")
17
18 @app.post("/hello")
19 def hello(person: Person):
20     return {"message": f"{person.name} ({person.company.name})"}

```

Для простоты здесь класс компании имеет только один атрибут - название компании. Отправка запроса в коде javascript в этом случае могла бы выглядеть так:

```

1 const response = await fetch("/hello", {
2     method: "POST",
3     headers: { "Accept": "application/json", "Content-Type": "application/json" },
4     body: JSON.stringify({
5         name: "Tom",
6         company: {name: "Google"}
7     })
8 });
9 const data = await response.json();
10 console.log(data);

```

## Task 12 -Создание простейшего API

Рассмотренного в прошлых темах материала достаточно для создания примитивного приложения. В этой теме попробуем реализовать простейшее приложение Web API в стиле REST. Архитектура REST предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером, где каждый тип запроса отвечает за определенное действие:

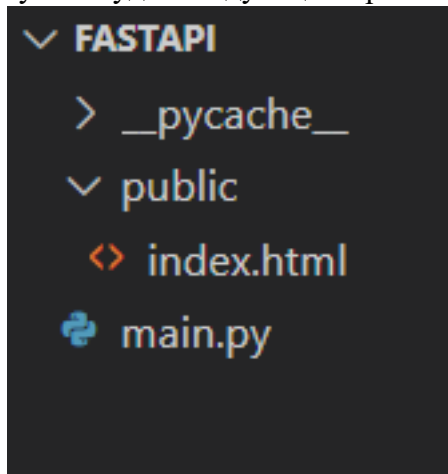
**GET** (получение данных)

**POST** (добавление данных)

**PUT** (изменение данных)

**DELETE** (удаление данных)

Для каждого из этих типов запросов класс **FastAPI** предоставляет соответствующие методы. Рассмотрим, как мы можем реализовать с помощью этих методов простейший API. Пусть у нас будет следующий проект:



### Создание сервера

В файле **main.py** определим следующий код:

```
1 import uuid
2 from fastapi import FastAPI, Body, status
3 from fastapi.responses import JSONResponse, FileResponse
4
5 class Person:
6     def __init__(self, name, age):
7         self.name = name
8         self.age = age
9         self.id = str(uuid.uuid4())
10
11 # условная база данных - набор объектов Person
12 people = [Person("Tom", 38), Person("Bob", 42), Person("Sam", 28)]
13
14 # для поиска пользователя в списке people
15 def find_person(id):
16     for person in people:
17         if person.id == id:
18             return person
19     return None
20
21 app = FastAPI()
22
```

```

23 @app.get("/")
24 async def main():
25     return FileResponse("public/index.html")
26
27 @app.get("/api/users")
28 def get_people():
29     return people
30
31 @app.get("/api/users/{id}")
32 def get_person(id):
33     # получаем пользователя по id
34     person = find_person(id)
35     print(person)
36     # если не найден, отправляем статусный код и сообщение об ошибке
37     if person==None:
38         return JSONResponse(
39             status_code=status.HTTP_404_NOT_FOUND,
40             content={ "message": "Пользователь не найден" }
41         )
42     #если пользователь найден, отправляем его
43     return person
44
45
46 @app.post("/api/users")
47 def create_person(data = Body()):
48     person = Person(data["name"], data["age"])
49     # добавляем объект в список people
50     people.append(person)
51     return person
52
53 @app.put("/api/users")
54 def edit_person(data = Body()):
55
56     # получаем пользователя по id
57     person = find_person(data["id"])
58     # если не найден, отправляем статусный код и сообщение об ошибке
59     if person == None:
60         return JSONResponse(
61             status_code=status.HTTP_404_NOT_FOUND,
62             content={ "message": "Пользователь не найден" }
63         )
64     # если пользователь найден, изменяем его данные и отправляем обратно клиенту
65     person.age = data["age"]
66     person.name = data["name"]
67     return person
68
69
70 @app.delete("/api/users/{id}")
71 def delete_person(id):
72     # получаем пользователя по id
73     person = find_person(id)
74

```

```

75     # если не найден, отправляем статусный код и сообщение об ошибке
76     if person == None:
77         return JsonResponse(
78             status_code=status.HTTP_404_NOT_FOUND,
79             content={ "message": "Пользователь не найден" }
80         )
81
82     # если пользователь найден, удаляем его
83     people.remove(person)
84     return person

```

Разберем в общих чертах этот код. Прежде всего для представления данных, с которыми мы будем работать, определяем класс **Person**.

```

1     class Person:
2         def __init__(self, name, age):
3             self.name = name
4             self.age = age
5             self.id = str(uuid.uuid4())

```

Этот класс содержит три атрибута. Два атрибута - name и age будут представлять имя и возраст пользователя и будут устанавливаться через конструктор. А третий атрибут - id будет служить для уникальной идентификации данного объекта и будет хранить значение guid. Для генерации guid применяется функция **uuid.uuid4()** из пакета uuid. В конструкторе Person сгенерированный guid преобразуется в строку и присваивается атрибуту id.

Для хранения данных в приложении определим список people, который будет выполнять роль условной базы данных и будет хранить объекты Person.

```

1     people = [Person("Tom", 38), Person("Bob", 42), Person("Sam", 28)]

```

Для поиска объекта Person в этом списке определена вспомогательная функция find\_person().

При обращении к корню веб-приложения, то есть по пути "/", оно будет отправлять в ответ файл index.html, то есть веб-страницу, посредством которой пользователь сможет взаимодействовать с сервером:

```

1     @app.get("/")
2     def main():
3         return FileResponse("public/index.html")

```

Далее определяются функции, которые собственно и представляют API. Вначале определяется функция, которая обрабатывает запрос типа GET по пути "api/users":

```

1     @app.get("/api/users")
2     def get_people():
3         return people

```

Запрос GET предполагает получение объектов, и в данном случае отправляем выше определенный список объектов Person.

Когда клиент обращается к приложению для получения одного объекта по id в запрос типа GET по адресу "api/users/{id}", то срабатывает другая функция:

```

1     @app.get("/api/users/{id}")
2     def get_person(id):
3         person = find_person(id)
4         if person==None:
5             return JsonResponse(
6                 status_code=status.HTTP_404_NOT_FOUND,

```



```
        content={ "message": "Пользователь не найден" }
    )
    return person
```

Здесь через параметр `id` получаем из пути запроса идентификатор объекта `Person` и по этому идентификатору ищем нужный объект в списке `people`. Если объект по `id` не был найден, то возвращаем с помощью класса `JSONResponse` статусный код 404 с некоторым сообщением в формате JSON. Если объект найден, то отправляем найденный объект клиенту.

При получении запроса типа DELETE по маршруту `"/api/users/{id}"` срабатывает другая функция:

```

15 @app.delete("/api/users/{id}")
16 def delete_person(id):
17     person = find_person(id)
18     if person == None:
19         return JsonResponse(
20             status_code=status.HTTP_404_NOT_FOUND,
21             content={ "message": "Пользователь не найден" }
22         )
23     people.remove(person)
24     return person

```

Здесь действует аналогичная логика - если объект по id не найден, отправляет статусный код 404. Если же объект найден, то удаляем его из списка и посылаем клиенту.

При получении запроса с методом POST по адресу "/api/users" срабатывает следующая функция:

```
@app.post("/api/users")
def create_person(data = Body()):
    person = Person(data["name"], data["age"])
    people.append(person)
    return person
```

Запрос типа POST предполагает передачу приложению отправляемых данных. Причем мы ожидаем, что клиент отправит данные, которые содержат значения name и age. Для краткости мы пока опускаем валидацию входных данных. И для получения данных из тела запроса с помощью класса Body получаем данные в параметр data и затем используем данные из этого параметра для создания объекта Person. Затем созданный объект добавляется в список people и отправляется назад клиенту.

Если приложению приходит PUT-запрос по адресу `"/api/users"`, то аналогичным образом получаем отправленные клиентом данные в виде объекта `Person` и пытаемся найти подобный объект в списке `people`. Если объект не найден, отправляем статусный код `404`. Если объект найден, то изменяем его данные и отправляем обратно клиенту:

```

3 @app.put("/api/users")
4 def edit_person(data = Body()):
5
6     person = find_person(data["id"])
7     if person == None:
8         return JSONResponse(
9             status_code=status.HTTP_404_NOT_FOUND,
10            content={ "message": "Пользователь не найден" }
11        )
12     person.age = data["age"]
13     person.name = data["name"]
14     return person

```

Таким образом, мы определили простейший API. Теперь добавим код клиента.

Определение клиента

Теперь в проекте определим папку **public**, в которую добавим новый файл **index.html**

Определим в файле **index.html** следующим код для взаимодействия с сервером FastAPI:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>Astana IT University </title>
6 </head>
7 <body>
8   <h2>Список пользователей</h2>
9   <div>
10     <input type="hidden" id="userId" />
11     <p>
12       Имя:<br/>
13       <input id="userName" />
14     </p>
15     <p>
16       Возраст:<br />
17       <input id="userAge" type="number" />
18     </p>
19     <button id="saveBtn">Сохранить</button>
20     <button id="resetBtn">Сбросить</button>
21   </div>
22   <table>
23     <thead><tr><th>Имя</th><th>Возраст</th><th></th></tr></thead>
24     <tbody>
25     </tbody>
26   </table>
27   <script>
28     // Получение всех пользователей
29     async function getUsers() {
30       // отправляет запрос и получаем ответ
31       const response = await fetch("/api/users", {
32         method: "GET",
33         headers: { "Accept": "application/json" }
34       });
35       // если запрос прошел нормально
36       if (response.ok === true) {
37         // получаем данные
38         const users = await response.json();
39         const rows = document.querySelector("tbody");
40         // добавляем полученные элементы в таблицу
```

```

48         users.forEach(user => rows.append(row(user)));
49     }
50 }
51 // Получение одного пользователя
52 async function getUser(id) {
53     const response = await fetch(`/api/users/${id}`, {
54         method: "GET",
55         headers: { "Accept": "application/json" }
56     });
57     if (response.ok === true) {
58         const user = await response.json();
59         document.getElementById("userId").value = user.id;
60         document.getElementById("userName").value = user.name;
61         document.getElementById("userAge").value = user.age;
62     }
63     else {
64         // если произошла ошибка, получаем сообщение об ошибке
65         const error = await response.json();
66         console.log(error.message); // и выводим его на консоль
67     }
68 }
69 // Добавление пользователя
70 async function createUser(userName, userAge) {
71
72     const response = await fetch("api/users", {
73         method: "POST",
74         headers: { "Accept": "application/json", "Content-Type": "application/json"
75     },
76     body: JSON.stringify({
77         name: userName,
78         age: parseInt(userAge, 10)
79     })
80     });
81     if (response.ok === true) {
82         const user = await response.json();
83         document.querySelector("tbody").append(row(user));
84     }
85     else {
86         const error = await response.json();
87         console.log(error.message);
88     }
89 }
90 // Изменение пользователя
91 async function editUser(userId, userName, userAge) {
92     const response = await fetch("api/users", {
93         method: "PUT",
94         headers: { "Accept": "application/json", "Content-Type": "application/json"
95     },
96     body: JSON.stringify({
97         id: userId,
98         name: userName,
99         age: parseInt(userAge, 10)

```

```

100         })
101     });
102     if (response.ok === true) {
103         const user = await response.json();
104         document.querySelector(`tr[data-
105 rowid='${user.id}']`).replaceWith(row(user));
106     }
107     else {
108         const error = await response.json();
109         console.log(error.message);
110     }
111 }
112 // Удаление пользователя
113 async function deleteUser(id) {
114     const response = await fetch(`/api/users/${id}`, {
115         method: "DELETE",
116         headers: { "Accept": "application/json" }
117     });
118     if (response.ok === true) {
119         const user = await response.json();
120         document.querySelector(`tr[data-rowid='${user.id}']`).remove();
121     }
122     else {
123         const error = await response.json();
124         console.log(error.message);
125     }
126 }
127
128 // сброс данных формы после отправки
129 function reset() {
130     document.getElementById("userId").value =
131     document.getElementById("userName").value =
132     document.getElementById("userAge").value = "";
133 }
134 // создание строки для таблицы
135 function row(user) {
136
137     const tr = document.createElement("tr");
138     tr.setAttribute("data-rowid", user.id);
139
140     const nameTd = document.createElement("td");
141     nameTd.append(user.name);
142     tr.append(nameTd);
143
144     const ageTd = document.createElement("td");
145     ageTd.append(user.age);
146     tr.append(ageTd);
147
148     const linksTd = document.createElement("td");
149
150     const editLink = document.createElement("button");
151     editLink.append("Изменить");

```

```

152     editLink.addEventListener("click", async() => await getUser(user.id));
153     linksTd.append(editLink);
154
155     const removeLink = document.createElement("button");
156     removeLink.append("Удалить");
157     removeLink.addEventListener("click", async () => await deleteUser(user.id));
158
159     linksTd.append(removeLink);
160     tr.appendChild(linksTd);
161
162     return tr;
163 }
164 // сброс значений формы
165 document.getElementById("resetBtn").addEventListener("click", () => reset());
166
167 // отправка формы
168 document.getElementById("saveBtn").addEventListener("click", async () => {
169
170     const id = document.getElementById("userId").value;
171     const name = document.getElementById("userName").value;
172     const age = document.getElementById("userAge").value;
173     if (id === "")
174         await createUser(name, age);
175     else
176         await editUser(id, name, age);
177     reset();
178 });
179
180 // загрузка пользователей
181 getUsers();
</script>
</body>
</html>

```

Основная логика здесь заключена в коде javascript. При загрузке страницы в браузере получаем все объекты из БД с помощью функции getUsers():

```

1  async function getUsers() {
2      // отправляет запрос и получаем ответ
3      const response = await fetch("/api/users", {
4          method: "GET",
5          headers: { "Accept": "application/json" }
6      });
7      // если запрос прошел нормально
8      if (response.ok === true) {
9          // получаем данные
10         const users = await response.json();
11         const rows = document.querySelector("tbody");
12         // добавляем полученные элементы в таблицу
13         users.forEach(user => rows.append(row(user)));
14     }
15 }

```

Для добавления строк в таблицу используется функция `getRow()`, которая возвращает строку. В этой строке будут определены ссылки для изменения и удаления пользователя.

Ссылка для изменения пользователя с помощью функции `getUser()` получает с сервера выделенного пользователя:

```
1  async function getUser(id) {
2      const response = await fetch(`/api/users/${id}`, {
3          method: "GET",
4          headers: { "Accept": "application/json" }
5      });
6      if (response.ok === true) {
7          const user = await response.json();
8          document.getElementById("userId").value = user.id;
9          document.getElementById("userName").value = user.name;
10         document.getElementById("userAge").value = user.age;
11     }
12     else {
13         // если произошла ошибка, получаем сообщение об ошибке
14         const error = await response.json();
15         console.log(error.message); // и выводим его на консоль
16     }
17 }
```

И выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит `id` пользователя, мы можем узнать, какое действие выполняется - добавление или редактирование. Если `id` не установлен (равен пустой строке), то выполняется функция `createUser`, которая отправляет данные в POST-запросе:

```
1  async function createUser(userName, userAge) {
2      const response = await fetch("api/users", {
3          method: "POST",
4          headers: { "Accept": "application/json", "Content-Type": "application/json" },
5          body: JSON.stringify({
6              name: userName,
7              age: parseInt(userAge, 10)
8          })
9      });
10     if (response.ok === true) {
11         const user = await response.json();
12         document.querySelector("tbody").append(row(user));
13     }
14     else {
15         const error = await response.json();
16         console.log(error.message);
17     }
18 }
19 }
```

Если же ранее пользователь был загружен на форму, и в скрытом поле сохранился его `id`, то выполняется функция `editUser`, которая отправляет PUT-запрос:

```
1  async function editUser(userId, userName, userAge) {
2      const response = await fetch("api/users", {
```

```

3     method: "PUT",
4     headers: { "Accept": "application/json", "Content-Type": "application/json" },
5     body: JSON.stringify({
6         id: userId,
7         name: userName,
8         age: parseInt(userAge, 10)
9     })
10    });
11    if (response.ok === true) {
12        const user = await response.json();
13        document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));
14    }
15    else {
16        const error = await response.json();
17        console.log(error.message);
18    }
19 }

```

И функция `deleteUser()` посылает на сервер запрос типа DELETE на удаление пользователя, и при успешном удалении на сервере удаляет объект по `id` из списка объектов `Person`.

Теперь запустим проект, и по умолчанию приложение отправит браузеру веб-страницу **index.html**, которая загрузит список объектов:

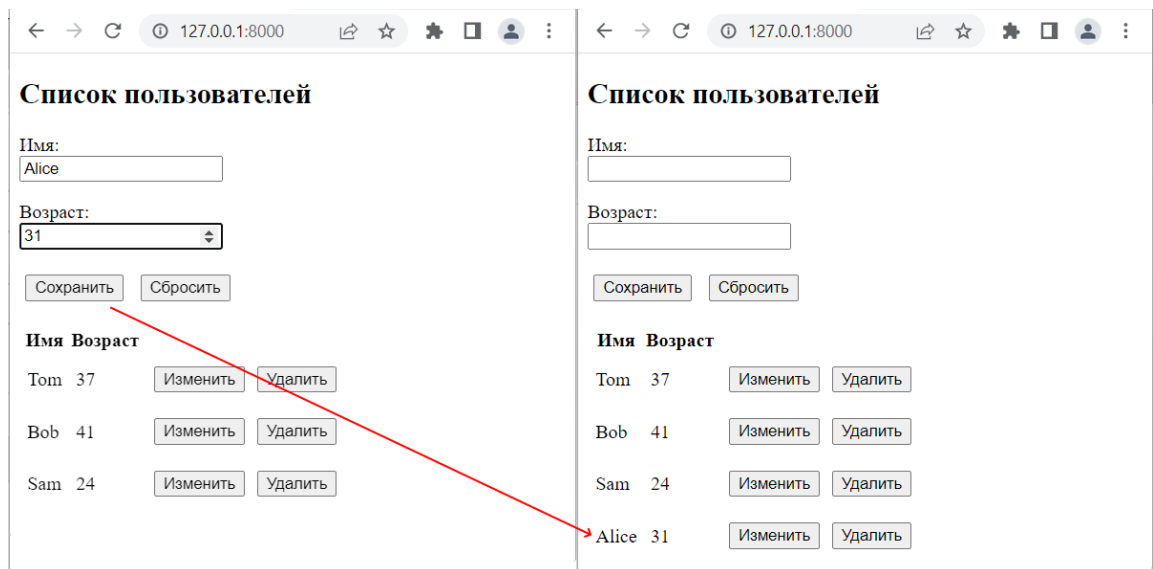
**Список пользователей**

Имя:

Возраст:

Имя	Возраст		
Tom	37	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Bob	41	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Sam	24	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>

После этого мы сможем выполнять все базовые операции с пользователями - получение, добавление, изменение, удаление. Например, добавим нового пользователя:

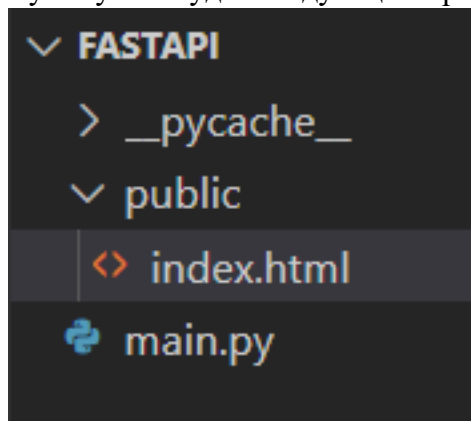


### Task 13 - Отправка форм

Работа с формами несколько отличается от получения на сервере данных в теле запроса. Прежде всего необходимо установить модуль **python-multipart** с помощью команды

Непосредственно в самом FastAPI для получения данных отправленных форм применяется класс **fastapi.Form**

Пусть у нас будет следующий проект:



В папке **public** определим файл **index.html** со следующим кодом:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Astana IT University</title>
5   <meta charset="utf-8" />
6 </head>
7 <body>
8   <h2>Форма ввода</h2>
9   <form action="postdata" method="post">
10    <p>

```



```

11     Name:<br>
12     <input name="username" />
13 </p>
14 <p>
15     Age:<br>
16     <input name="userage" type="number" />
17 </p>
18     <input type="submit" value="Send" />
19 </form>
20 </body>
21 </html>

```

Здесь определена простейшая форма для ввода имени и возраста. И по нажатию на кнопку данные отправляются в запросе POST по адресу "postdata".

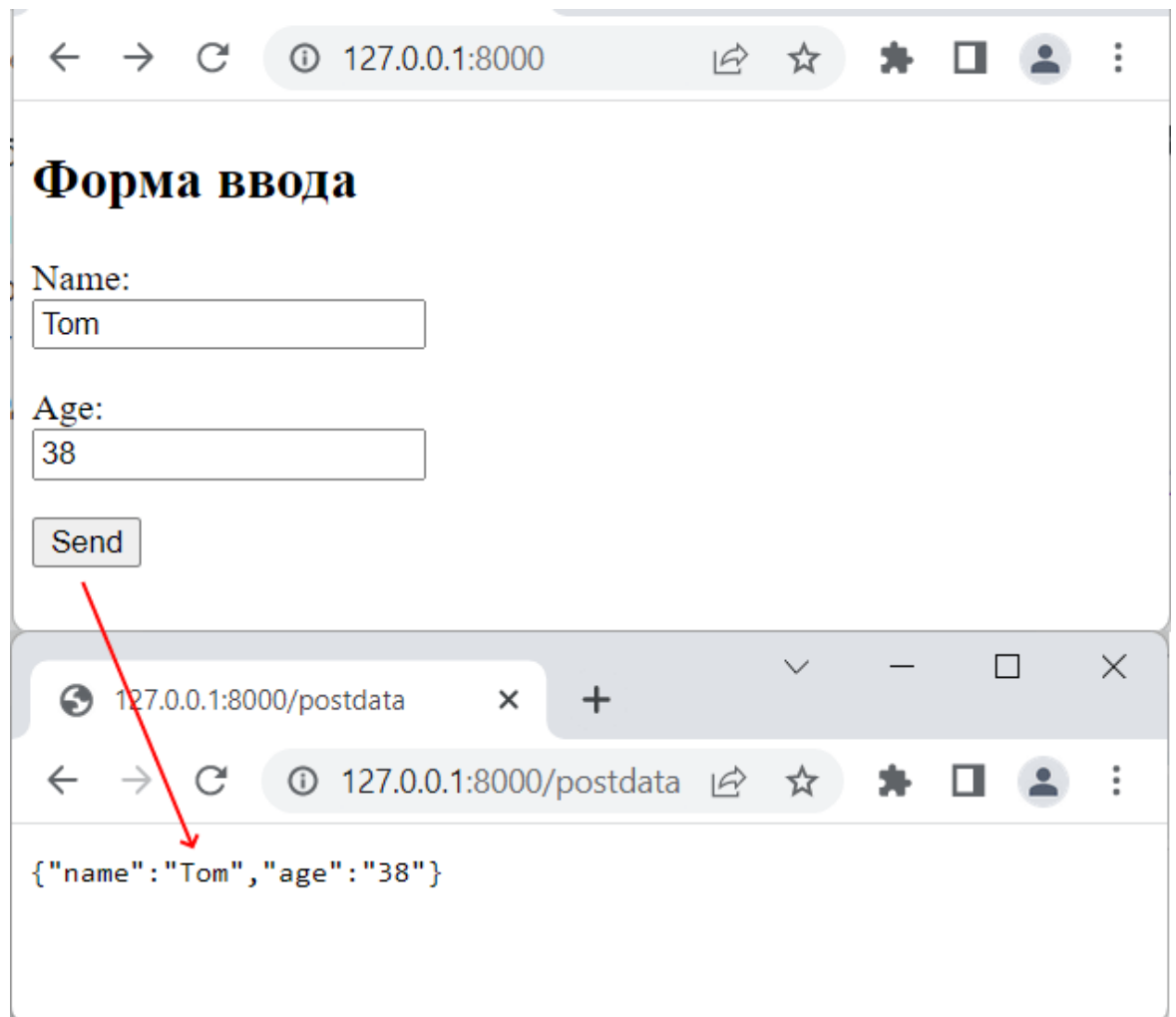
Для получения данных форм в файле **main.py** определим следующий код:

```

1 from fastapi import FastAPI, Form
2 from fastapi.responses import FileResponse
3
4 app = FastAPI()
5
6 @app.get("/")
7 def root():
8     return FileResponse("public/index.html")
9
10
11 @app.post("/postdata")
12 def postdata(username = Form(), userage=Form()):
13     return {"name": username, "age": userage}

```

В функции postdata, которая обрабатывает запрос по одноименному пути, через параметры получаем отправленные данные. Причем параметры называются также, как и атрибуты name у полей формы. А самим параметрам присваивается объект Form.



### Валидация

Класс **Form** из пакета `fastapi` предоставляет возможности валидации значений форм с помощью некоторых параметров конструктора:

**min\_length**: устанавливает минимальное количество символов в значении параметра

**max\_length**: устанавливает максимальное количество символов в значении параметра

**regex**: устанавливает регулярное выражение, которому должно соответствовать значение параметра

**lt**: значение параметра должно быть меньше определенного значения

**le**: значение параметра должно быть меньше или равно определенному значению

**gt**: значение параметра должно быть больше определенного значения

**ge**: значение параметра должно быть больше или равно определенному значению

Применим некоторые параметры:

```
1 from fastapi import FastAPI, Form
2
3 from fastapi.responses import FileResponse
4
5
6 app = FastAPI()
7
8
9 @app.get("/")
10 def root():
11     return FileResponse("public/index.html")
```

```

12 @app.post("/postdata")
13 def postdata(username: str = Form(min_length=2, max_length=20),
14             usage: int = Form(ge=18, lt=111)):
15     return {"name": username, "age": usage}

```

В данном случае параметр username должен иметь не меньше 2 и не больше 20 символов, а параметр usage должен представлять число в диапазоне от 18 (включительно) до 111 (не включая)

### Значение по умолчанию

С помощью параметра default конструктора Form можно установить значение по умолчанию на случай, если во входящих данных отсутствуют соответствующие значения:

```

1 from fastapi import FastAPI, Form
2 from fastapi.responses import FileResponse
3
4 app = FastAPI()
5
6 @app.get("/")
7 def root():
8     return FileResponse("public/index.html")
9
10
11 @app.post("/postdata")
12 def postdata(username: str = Form(default="Undefined", min_length=2,
13 max_length=20),
14             usage: int = Form(default=18, ge=18, lt=111)):
15     return {"name": username, "age": usage}

```

### Отправка списков

С помощью форм могут отправляться наборы данных. Например, изменим файл **index.html** следующим образом:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Astana IT university </title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <h2>Форма ввода</h2>
9     <form action="postdata" method="post">
10         <p>
11             Name:<br>
12             <input name="username" />
13         </p>
14         <p>
15             Languages:<br>
16             <input name="languages" /><br><br>
17             <input name="languages" /><br><br>
18             <input name="languages" /><br><br>
19         </p>
20         <input type="submit" value="Send" />

```

```
21     </form>
22 </body>
23 </html>
```

Здесь на форме определен набор элементов ввода, которые имеют одно и то же имя - "languages". При отправке формы из значений этих элементов будет формироваться набор. Для получения этого набора в коде сервера определим соответствующий параметр как параметр типа **list**:

```
1 from fastapi import FastAPI, Form
2 from fastapi.responses import FileResponse
3
4 app = FastAPI()
5
6
7 @app.get("/")
8 def root():
9     return FileResponse("public/index.html")
10
11
12 @app.post("/postdata")
13 def postdata(username: str = Form(),
14              languages: list = Form()):
15     return {"name": username, "languages": languages}
```

Единственное неудобство, с которым в данном случае можно столкнуться, это оправка пустых строк, как в скриншоте ниже в случае с третьим полем ввода языка:

