**Основы FastAPI**

**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>

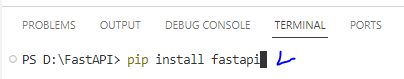
На данный момент поддерживается Python версии 3.6 и выше.

**Необходимые инструменты и установка**

Для 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**



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](https://www.uvicorn.org/) или [Hypercorn](https://hypercorn.readthedocs.io/en/latest/). В данном случае будем использовать Univer. Также установим его пакеты с помощью менеджера pip с помощью следующей команды:

pip install "uvicorn[standard]"

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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import HTMLResponse    app = FastAPI()    @app.get("/")  def read\_root():      html\_content = "<h2>Hello World!</h2>"      return HTMLResponse(content=html\_content) |

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

|  |
| --- |
| app = FastAPI() |

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

|  |
| --- |
| @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:

|  |
| --- |
| def read\_root():      html\_content = "<h2>Hello World!</h2>"      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 World!</h2>" со значением " Hello World!". То есть когда клиент обратится к веб-приложению по пути "/", ему будет отправлен html-код "<h2>Hello World!</h2>".

**Запуск приложения**

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

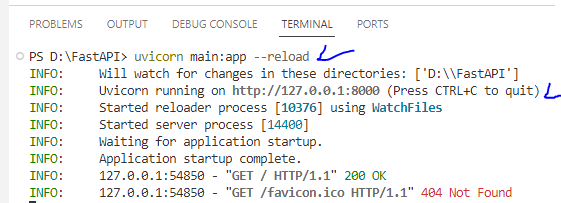
uvicorn main:app --reload

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

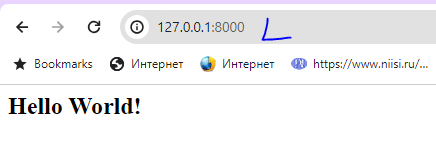
main указывает на название модуля, которое по умолчанию совпадает с названием файла - main

app указывает на объект приложения, созданный в строке app = FastAPI()

--reload позволяет отслеживать изменения в файлах исходного кода и автоматически перезапускать.



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:



**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](https://www.starlette.io/) представляет другой легковесный ASGI-фреймворк для создания асинхронных веб-сервисов на Python. Собственно fastAPI работает поверх Scarlette, используя и дополняя его функциональность. Это касается не только самого класса FastAPI, но и других классов фреймворка - многие из них используют функционал Scarlette.

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 World!"} |

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.:8000/).

Функция возвращает некоторые 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.: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 World!"}  @app.get("/about")  def about():      return {"message": "О сайте"} |

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

**Отправка ответа**

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

|  |
| --- |
| from fastapi import FastAPI    app = FastAPI()    @app.get("/")  def root():      return {"message": "Hello World!"} |

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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import JSONResponse  from fastapi.encoders import jsonable\_encoder    app = FastAPI()    @app.get("/")  def root():      data = {"message": "Hello World!"}      json\_data = jsonable\_encoder(data)      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 формает json. В реальности мы можем возвратить 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 позволяется также отправить ответ, который не покрывается встроенными классами, например, в каком-то нестандартном формате. Для отпределения 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:

|  |
| --- |
| from fastapi import FastAPI, Response    app = FastAPI()      @app.get("/")  def root():      data = "Hello World!"      return Response(content=data, media\_type="text/plain") |

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**

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import PlainTextResponse    app = FastAPI()    @app.get("/")  def root():      data = "Hello World!"      return PlainTextResponse(content=data) |

**HTMLResponse**

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

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import HTMLResponse    app = FastAPI()    @app.get("/")  def root():      data = "<h2>Hello World!</h2>"      return HTMLResponse(content=data) |

**Установка типа ответа через методы FastAPI**

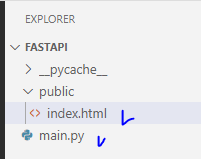
Методы FastAPI такие как get(), post() и т.д. позволяют задать тип ответа с помощью параметра **response\_class**:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import PlainTextResponse, JSONResponse, HTMLResponse    app = FastAPI()    @app.get("/text", response\_class = PlainTextResponse)  def root\_text():      return "Hello World!"    @app.get("/html", response\_class = HTMLResponse)  def root\_html():      return "<h2>Hello World!</h2>" |

**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:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>My HTML </title>      <meta charset="utf-8" />  </head>  <body>      <h2>Hello World!</h2>  </body>  </html> |

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

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import FileResponse    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    # альтернативный вариант  @app.get("/file", response\_class = FileResponse)  def root\_html():      return "public/index.html" |

В качестве обязательного параметра конструктор FileResponse принимает путь к файлу.

При отправке файла браузер сможет интерпретировать его как некоторый код html, и соответственно мы сможем лицезреть эту страницу в браузере:

Браузер по умолчанию пытается интерпретировать и отобразить все файлы, которые может, например, текстовые файлы, файлы изображений, какие-то другие мульмедиа-файлы. Если файл не может быть интепретирован браузером, то он загружается. Однако может возникнуть необходимость автоматически загрузить без отображения какие-нибудь интерпретируемые файлы, например, те же самые файлы html. В этом случае мы можем установить для параметра media\_type значение **application/octet-stream**. Кроме того, с помощью параметра **filename** для загружаемого файла может задать имя:

|  |
| --- |
| import mimetypes  from fastapi import FastAPI  from fastapi.responses import FileResponse    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html",                          filename="mainpage.html",                          media\_type="application/octet-stream") |

**Параметры пути**

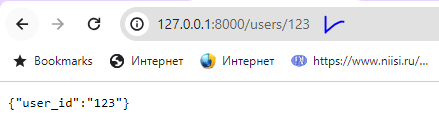
Путь запроса может содержать специальные значения, которые фреймворк FastAPI может связать с параметрыми функции обработчика запроса. Это так называемые параметры пути (path parameter). Благодаря параметрам пути можно передавать в приложения некоторые значения.

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

|  |
| --- |
| from fastapi import FastAPI    app = FastAPI()    @app.get("/users/{id}")  def users(id):      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.



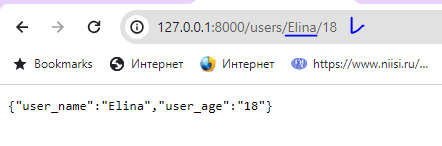
Так, на скриншоте видно, что браузер отправляет запрос по адресу http://127.0.0.1:8000/users/123, то есть путь запроса в данном случае представляет users/123. Фреймворк FastAPI видит, что этот путь соответствует шаблону "/users/{id}", поэтому данный запрос будет обрабатываться функцией users.

Параметр id в шаблоне пути составляет второй сегмент адреса. Соответственно фрейм ворк сможет сопоставть сегмент "123" с параметром id. Поэтому при обращении по адресу http://127.0.0.1:8000/users/123 параметр id в функции users получит значение 123.

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} |

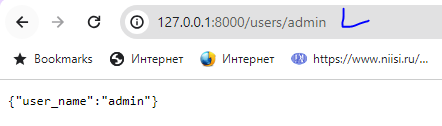
Такому шаблону соответствовал бы запрос по пути http://127.0.0.1:8000/users/Elina/18, и фреймворк FastAPI автоматически распарсил бы запрошенный путь и разделил бы его последний сегмент на параметры name и age.

**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}      @app.get("/users/admin")  def admin():      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:

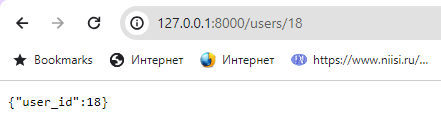
|  |
| --- |
| from fastapi import FastAPI    app = FastAPI()    @app.get("/users/admin")  def admin():      return {"message": "Hello admin"}      @app.get("/users/{name}")  def users(name):      return {"user\_name": name} |

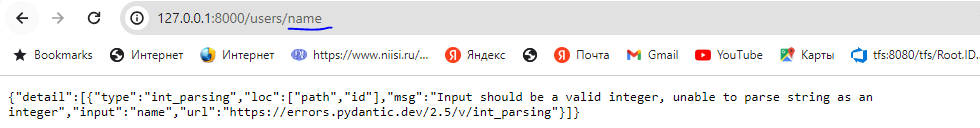
**Ограничения типа параметров**

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

|  |
| --- |
| from fastapi import FastAPI    app = FastAPI()    @app.get("/users/{id}")  def users(id: int):      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**

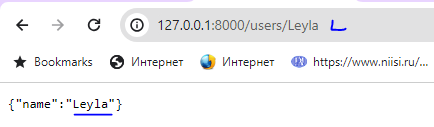
Дополнительно для работы с параметрами пути фреймворк FastAPI предоставляет класс **Path** из пакета fastapi. Класс Path позволяет валидировать значения параметров. В частности, через конструктор Path можно установить следующие параметры для валидации значений:

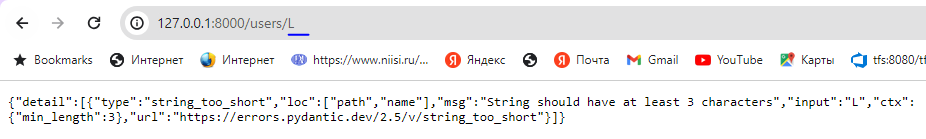
* **min\_length**: устанавливает минимальное количество символов в значении параметра
* **max\_length**: устанавливает максимальное количество символов в значении параметра
* **regex**: устанавливает регулярное выражение, которому должно соответствовать значение параметра
* **lt**: значение параметра 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 некотрые 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:

|  |
| --- |
| from fastapi import FastAPI, Path    app = FastAPI()    @app.get("/users/{name}/{age}")  def users(name:str  = Path(min\_length=3, max\_length=20),              age: int = Path(ge=18, lt=111)):      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:

|  |
| --- |
| from fastapi import FastAPI, Path    app = FastAPI()    @app.get("/users/{phone}")  def users(phone:str  = Path(regex="^\d{11}$")):      return {"phone": phone} |

Здесь параметр phone должен состоять из 11 цифр.

**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

|  |
| --- |
| http://127.0.0.1:8000/users/add?name=Annya&age=18 |

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= Annya&age=18, представляет строку запроса (query string). В данной статье нас будет интересовать прежде всего строка запроса.

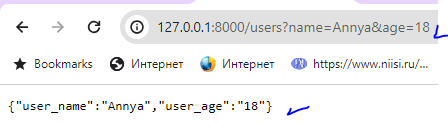
Строка запроса состоит из параметров. Каждый параметр определяется в форме

|  |
| --- |
| имя\_параметра=значение\_параметра |

Если строка запроса содержит несколько параметров, то они одтеляются друг от друга знаком амперсанда &. Так, в примере в адресом http://127.0.0.1:8000/users/add?name= Annya&age=18 строка запроса состоит из двух параметров: параметр name имеет значение " Annya", а параметр age имеет значение 18.

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 некотрые 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:

|  |
| --- |
| from fastapi import FastAPI, Query    app = FastAPI()    @app.get("/users")  def users(name:str  = Query(min\_length=3, max\_length=20),              age: int = Query(ge=18, lt=111)):      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:

|  |
| --- |
| from fastapi import FastAPI, Query    app = FastAPI()    @app.get("/users")  def users(phone:str  = Query(regex="^\d{11}$")):      return {"phone": phone} |

Здесь параметр phone должен состоять из 11 цифр.

Query позволяет установить значение по умолчанию с помощью параметра default:

|  |
| --- |
| from fastapi import FastAPI, Query    app = FastAPI()    @app.get("/users")  def users(name: str = Query(default="Undefined", min\_length=2)):      return {"name": name} |

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

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

|  |
| --- |
| from fastapi import FastAPI, Query    app = FastAPI()    @app.get("/users")  def users(name:str | None = Query(default=None, min\_length=2)):      if name==None:          return {"name": "Undefined"}      else:          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:

|  |
| --- |
| http://127.0.0.1:8000/users?people=Elza&people=Dilyara&people=Lada |

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

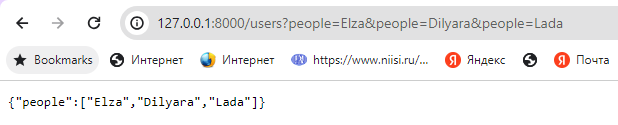
Определим следующее приложение:

|  |
| --- |
| from fastapi import FastAPI, Query    app = FastAPI()    @app.get("/users")  def users(people: list[str]  = Query()):      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=Elza&people=Dilyara&people=Lada>



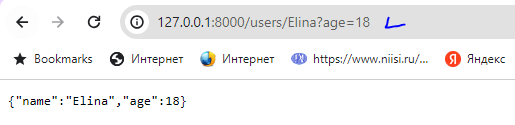
**Сочетание параметров пути и строки запроса**

При необходимости можно сочетать параметры пути и строки запроса:

|  |
| --- |
| from fastapi import FastAPI, Path, Query    app = FastAPI()    @app.get("/users/{name}")  def users(name:str  = Path(min\_length=3, max\_length=20),              age: int = Query(ge=18, lt=111)):      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/Elina?age=18>



**Отправка статусных кодов**

Одной из расспространненых 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 успешноее выполнение операции
* **3xx**: предназначены для переадресации
* **4xx**: предназначены для отправки информации об ошибок 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

|  |
| --- |
| from fastapi import FastAPI, status    app = FastAPI()    @app.get("/notfound", status\_code=status.HTTP\_404\_NOT\_FOUND)  def notfound():      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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import JSONResponse    app = FastAPI()    @app.get("/notfound")  def notfound():      return JSONResponse(content={"message": "Resource Not Found"}, status\_code=404) |

**Изменение статусного кода**

Можно комбинировать оба подхода:

|  |
| --- |
| from fastapi import FastAPI, Response, Path    app  = FastAPI()    @app.get("/users/{id}", status\_code=200)  def users(response: Response, id: int = Path()):      if id < 1:          response.status\_code = 400          return {"message": "Incorrect Data"}      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)

**Переадресация**

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

|  |
| --- |
| import mimetypes  from fastapi import FastAPI  from fastapi.responses import RedirectResponse, PlainTextResponse    app = FastAPI()    @app.get("/old")  def old():      return RedirectResponse("/new")    @app.get("/new")  def new():      return PlainTextResponse("Новая страница") |

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

|  |
| --- |
| @app.get("/old", response\_class= RedirectResponse)  def old():      return "/new" |

Также можно перенаправлять по абсолютному адресу:

|  |
| --- |
| @app.get("/old")  def old():      return RedirectResponse("<https://mysite/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:

|  |
| --- |
| @app.get("/old")  def old():      return RedirectResponse("/new", status\_code=302) |

или так

|  |
| --- |
| @app.get("/old", response\_class= RedirectResponse, status\_code=302)  def old():      return "/new" |

**Статические файлы**

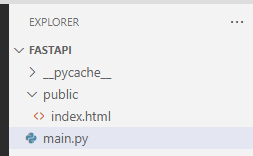
Для определения интерфейса для работы с сервером нередко используются html-страницы, то есть cтатические файлы с кодом html, которые могут использовать какие-то другие статические файлы - файлы стилей css, изображений, скриптов javascript и т.д. Для работы со статическими файлами FastAPI предоставляет 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 определеим в этом каталоге простейнький файл **index.html** со следующим кодом:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>My site</title>      <meta charset="utf-8" />  </head>  <body>      <h2>Hello World!</h2>  </body>  </html> |

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

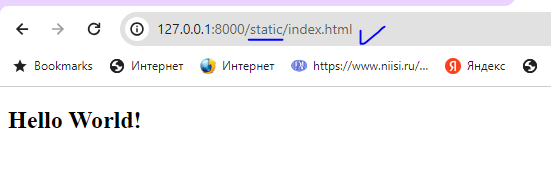
|  |
| --- |
| from fastapi import FastAPI  from fastapi.staticfiles import StaticFiles    app = FastAPI()    app.mount("/static", StaticFiles(directory="public")) |

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 обработчка 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:



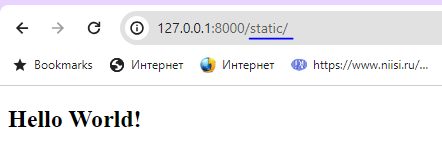
Подобным образом мы можем добавлять в каталог public и другие статические файлы.

**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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.staticfiles import StaticFiles    app = FastAPI()    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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.staticfiles import StaticFiles    app = FastAPI()    app.mount("/", StaticFiles(directory="public", html=True)) |

**Получение данных запроса**

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

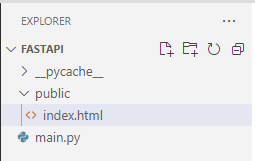
**Body**

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>My site</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 введеное 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.

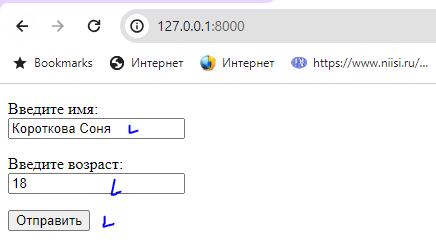
Для обработки полученных в POST-запросе данных по адресу "/hello" определена функция hello(). Эта функция имеет один параметр - data, который получает содержимое тела запроса:

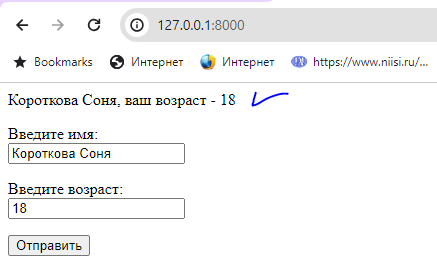
|  |
| --- |
| data = Body() |

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".





**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:

|  |
| --- |
| 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), age = Body(embed=True)):      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 некотрые параметры:

|  |
| --- |
| 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:str  = Body(embed=True, min\_length=3, max\_length=20),              age: int = Body(embed=True, ge=18, lt=111)):      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)

**Получение данных запроса в виде объекта класса**

В прошлой теме 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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import FileResponse  from pydantic import BaseModel    class Person(BaseModel):      name: str      age: int    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    @app.post("/hello")  def hello(person: Person):      return {"message": f"Привет, {person.name}, твой возраст - {person.age}"} |

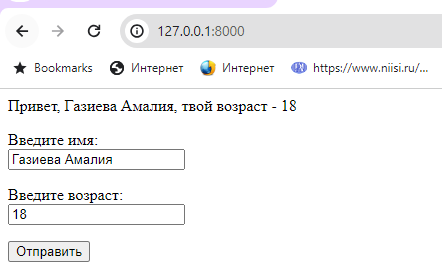
Здесь в функции hello получаем данные из тела запроса в объект класса Person. Данный класс унаследован от **BaseModel**. Класс Person определяет два атрибута, которые соответствуют данным из тела запроса, которые мы собираемся получить. В данном случае это атрибут name, который представляет строку, и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

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>My site</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 введеные в поля 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> |

Здесь на сервер по адресу "/hello" в запросе типа POST отправляются введеные 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**:

|  |
| --- |
| from fastapi import FastAPI, Body  from fastapi.responses import FileResponse  from pydantic import BaseModel    class Person(BaseModel):      name: str      age: int | None = None    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    @app.post("/hello")  def hello(person: Person):      if person.age == None:          return {"message": f"Привет, {person.name}"}      else:          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 валдации 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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import FileResponse  from pydantic import BaseModel, Field    class Person(BaseModel):      name: str = Field(default="Undefined", min\_length=3, max\_length=20)      age: int= Field(default=18, ge=18, lt=111)    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    @app.post("/hello")  def hello(person: Person):      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 получаются значения по умолчанию: строку "Undefined" и число 18 соответственно.

**Получение списков**

Подобным образом можно получать список объектов модели:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import FileResponse  from pydantic import BaseModel    class Person(BaseModel):      name: str      age: int    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    @app.post("/hello")  def hello(people:list[Person]):      return {"message": people} |

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

|  |
| --- |
| const response = await fetch("/hello", {      method: "POST",      headers: { "Accept": "application/json", "Content-Type": "application/json" },      body: JSON.stringify([          { name: "Гумерова Карина", age: 18 },          { name: "Загриева Иделия", age: 19 },          { name: "Закирова Лейла", age: 17 }      ])  });  const data = await response.json();  console.log(data); |

**Получение вложенных списков**

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

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import FileResponse  from pydantic import BaseModel    class Person(BaseModel):      name: str      languages: list = []    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    @app.post("/hello")  def hello(person: Person):      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:

|  |
| --- |
| const response = await fetch("/hello", {      method: "POST",      headers: { "Accept": "application/json", "Content-Type": "application/json" },      body: JSON.stringify({          name: "Ключникова Анна",          languages: ["Python", "JavaScript"]      })  });  const data = await response.json();  console.log(data);      // {message: "Name: Ключникова Анна. Languages: ['Python', 'JavaScript']"} |

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

|  |
| --- |
| class Person(BaseModel):      name: str      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:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import FileResponse  from pydantic import BaseModel    class Company(BaseModel):      name: str    class Person(BaseModel):      name: str      company: Company    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")    @app.post("/hello")  def hello(person: Person):      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:

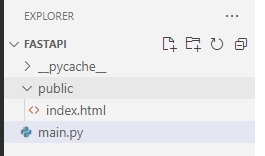
|  |
| --- |
| const response = await fetch("/hello", {      method: "POST",      headers: { "Accept": "application/json", "Content-Type": "application/json" },          body: JSON.stringify({          name: "Яшнова Анна",          company: {name: "Google"}      })  });  const data = await response.json();  console.log(data); |

**Cоздание 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:

|  |
| --- |
| import uuid  from fastapi import FastAPI, Body, status  from fastapi.responses import JSONResponse, FileResponse    class Person:      def \_\_init\_\_(self, name, age):          self.name = name          self.age = age          self.id = str(uuid.uuid4())    # условная база данных - набор объектов Person  people = [Person("Абдуллина Диляра", 18), Person("Довыдова Анна", 17), Person("Ахтямова Элина", 16)]    # для поиска пользователя в списке people  def find\_person(id):     for person in people:          if person.id == id:             return person     return None    app = FastAPI()    @app.get("/")  async def main():      return FileResponse("public/index.html")    @app.get("/api/users")  def get\_people():      return people    @app.get("/api/users/{id}")  def get\_person(id):      # получаем пользователя по id      person = find\_person(id)      print(person)      # если не найден, отправляем статусный код и сообщение об ошибке      if person==None:          return JSONResponse(                  status\_code=status.HTTP\_404\_NOT\_FOUND,                  content={ "message": "Пользователь не найден" }          )      #если пользователь найден, отправляем его      return person      @app.post("/api/users")  def create\_person(data  = Body()):      person = Person(data["name"], data["age"])      # добавляем объект в список people      people.append(person)      return person    @app.put("/api/users")  def edit\_person(data  = Body()):        # получаем пользователя по id      person = find\_person(data["id"])      # если не найден, отправляем статусный код и сообщение об ошибке      if person == None:          return JSONResponse(                  status\_code=status.HTTP\_404\_NOT\_FOUND,                  content={ "message": "Пользователь не найден" }          )      # если пользователь найден, изменяем его данные и отправляем обратно клиенту      person.age = data["age"]      person.name = data["name"]      return person      @app.delete("/api/users/{id}")  def delete\_person(id):      # получаем пользователя по id      person = find\_person(id)        # если не найден, отправляем статусный код и сообщение об ошибке      if person == None:          return JSONResponse(                  status\_code=status.HTTP\_404\_NOT\_FOUND,                  content={ "message": "Пользователь не найден" }          )        # если пользователь найден, удаляем его      people.remove(person)      return person |

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

|  |
| --- |
| class Person:      def \_\_init\_\_(self, name, age):          self.name = name          self.age = age          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.

|  |
| --- |
| people = [Person("Абдуллина Диляра", 18), Person("Довыдова Анна", 17), Person("Ахтямова Элина", 16)] |

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:

|  |
| --- |
| @app.get("/")  def main():      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":

|  |
| --- |
| @app.get("/api/users")  def get\_people():      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:

|  |
| --- |
| @app.get("/api/users/{id}")  def get\_person(id):      person = find\_person(id)      if person==None:          return JSONResponse(                  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:

|  |
| --- |
| @app.delete("/api/users/{id}")  def delete\_person(id):      person = find\_person(id)      if person == None:          return JSONResponse(                  status\_code=status.HTTP\_404\_NOT\_FOUND,                  content={ "message": "Пользователь не найден" }          )      people.remove(person)      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:

|  |
| --- |
| @app.put("/api/users")  def edit\_person(data  = Body()):        person = find\_person(data["id"])      if person == None:          return JSONResponse(                  status\_code=status.HTTP\_404\_NOT\_FOUND,                  content={ "message": "Пользователь не найден" }          )      person.age = data["age"]      person.name = data["name"]      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:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>My site</title>  <style>  td {padding:5px;}  button{margin: 5px;}  </style>  </head>  <body>      <h2>Список пользователей</h2>      <div>          <input type="hidden" id="userId" />          <p>              Имя:<br/>              <input id="userName" />          </p>          <p>              Возраст:<br />              <input id="userAge" type="number" />          </p>           <p>              <button id="saveBtn">Сохранить</button>              <button id="resetBtn">Сбросить</button>          </p>      </div>      <table>          <thead><tr><th>Имя</th><th>Возраст</th><th></th></tr></thead>          <tbody>          </tbody>      </table>        <script>      // Получение всех пользователей          async function getUsers() {              // отправляет запрос и получаем ответ              const response = await fetch("/api/users", {                  method: "GET",                  headers: { "Accept": "application/json" }              });              // если запрос прошел нормально              if (response.ok === true) {                  // получаем данные                  const users = await response.json();                  const rows = document.querySelector("tbody");                  // добавляем полученные элементы в таблицу                  users.forEach(user => rows.append(row(user)));              }          }          // Получение одного пользователя          async function getUser(id) {              const response = await fetch(`/api/users/${id}`, {                  method: "GET",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  document.getElementById("userId").value = user.id;                  document.getElementById("userName").value = user.name;                  document.getElementById("userAge").value = user.age;              }              else {                  // если произошла ошибка, получаем сообщение об ошибке                  const error = await response.json();                  console.log(error.message); // и выводим его на консоль              }          }          // Добавление пользователя          async function createUser(userName, userAge) {                const response = await fetch("api/users", {                  method: "POST",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector("tbody").append(row(user));              }              else {                  const error = await response.json();                  console.log(error.message);              }          }          // Изменение пользователя          async function editUser(userId, userName, userAge) {              const response = await fetch("api/users", {                  method: "PUT",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      id: userId,                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));              }              else {                  const error = await response.json();                  console.log(error.message);              }          }          // Удаление пользователя          async function deleteUser(id) {              const response = await fetch(`/api/users/${id}`, {                  method: "DELETE",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector(`tr[data-rowid='${user.id}']`).remove();              }              else {                  const error = await response.json();                  console.log(error.message);              }          }            // сброс данных формы после отправки          function reset() {              document.getElementById("userId").value =              document.getElementById("userName").value =              document.getElementById("userAge").value = "";          }          // создание строки для таблицы          function row(user) {                const tr = document.createElement("tr");              tr.setAttribute("data-rowid", user.id);                const nameTd = document.createElement("td");              nameTd.append(user.name);              tr.append(nameTd);                const ageTd = document.createElement("td");              ageTd.append(user.age);              tr.append(ageTd);                const linksTd = document.createElement("td");                const editLink = document.createElement("button");              editLink.append("Изменить");              editLink.addEventListener("click", async() => await getUser(user.id));              linksTd.append(editLink);                const removeLink = document.createElement("button");              removeLink.append("Удалить");              removeLink.addEventListener("click", async () => await deleteUser(user.id));                linksTd.append(removeLink);              tr.appendChild(linksTd);                return tr;          }          // сброс значений формы          document.getElementById("resetBtn").addEventListener("click", () =>  reset());            // отправка формы          document.getElementById("saveBtn").addEventListener("click", async () => {                const id = document.getElementById("userId").value;              const name = document.getElementById("userName").value;              const age = document.getElementById("userAge").value;              if (id === "")                  await createUser(name, age);              else                  await editUser(id, name, age);              reset();          });            // загрузка пользователей          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():

|  |
| --- |
| async function getUsers() {      // отправляет запрос и получаем ответ      const response = await fetch("/api/users", {          method: "GET",          headers: { "Accept": "application/json" }      });      // если запрос прошел нормально      if (response.ok === true) {          // получаем данные          const users = await response.json();          const rows = document.querySelector("tbody");          // добавляем полученные элементы в таблицу          users.forEach(user => rows.append(row(user)));      }  } |

Для добавления строк в таблицу используется функция row(), 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:

|  |
| --- |
| async function getUser(id) {      const response = await fetch(`/api/users/${id}`, {          method: "GET",          headers: { "Accept": "application/json" }      });      if (response.ok === true) {          const user = await response.json();          document.getElementById("userId").value = user.id;          document.getElementById("userName").value = user.name;          document.getElementById("userAge").value = user.age;      }      else {          // если произошла ошибка, получаем сообщение об ошибке          const error = await response.json();          console.log(error.message); // и выводим его на консоль      }  } |

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:

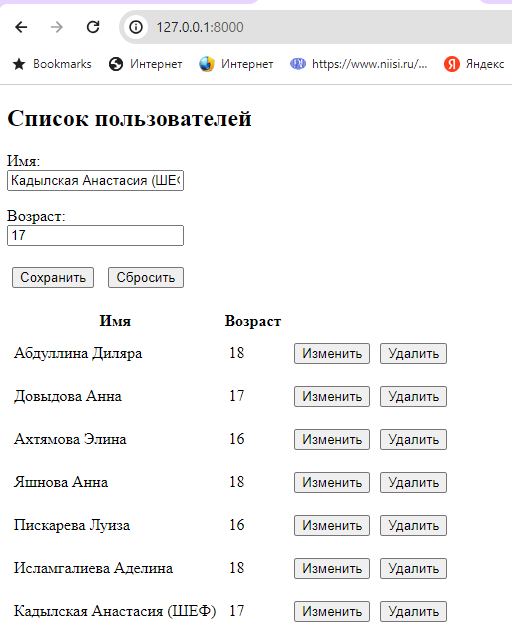
|  |
| --- |
| async function createUser(userName, userAge) {        const response = await fetch("api/users", {          method: "POST",          headers: { "Accept": "application/json", "Content-Type": "application/json" },          body: JSON.stringify({              name: userName,              age: parseInt(userAge, 10)          })      });      if (response.ok === true) {          const user = await response.json();          document.querySelector("tbody").append(row(user));      }      else {          const error = await response.json();          console.log(error.message);      }  } |

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

|  |
| --- |
| async function editUser(userId, userName, userAge) {      const response = await fetch("api/users", {          method: "PUT",          headers: { "Accept": "application/json", "Content-Type": "application/json" },          body: JSON.stringify({              id: userId,              name: userName,              age: parseInt(userAge, 10)          })      });      if (response.ok === true) {          const user = await response.json();          document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));      }      else {          const error = await response.json();          console.log(error.message);      }  } |

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:



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

**Отправка и получение заголовков**

**Отправка заголовков**

Для отправки заголовка в конструктор класса Response или его наследников параметру headers передается словарь, где ключи представляют названия заголовков:

|  |
| --- |
| from fastapi import FastAPI, Response    app = FastAPI()    @app.get("/")  def root():      data = "Hello World!"      return Response(content=data, media\_type="text/plain", headers={"Secret-Code" : "123459"}) |

Для примера в данном случае клиенту отправляется кастомный заголовок "Secret-Code" со значением "123459":

Также можно задать заголовки с помощью атрибута **headers**, который есть у класса Response и его наследников. Данный атрибут фактически представляет словарь, где ключи - названия заголовков:

|  |
| --- |
| from fastapi import FastAPI, Response    app = FastAPI()      @app.get("/")  def root(response: Response):      response.headers["Secret-Code"] = "123459"      return {"message": "Hello World!"} |

При данном подходе объект Response явным образом передается в функцию-обработчик через параметр.

**Получение заголовков**

Для получения заголовков запроса применяется класс **fastapi.Header**. Например, получим заголовок User-Agent:

|  |
| --- |
| from fastapi import FastAPI, Header    app = FastAPI()      @app.get("/")  def root(user\_agent: str = Header()):      return {"User-Agent": user\_agent} |

При этом параметр функции, в который получаем значение заголовка, должен называться как и заголовок за тем исключением, что вместо дефиса применяется прочерк: параметр "user\_agent" соответствует заголовку "User-Agent".

Но в запросе может отсутствовать нужный заголовок. В этом случае мы можем установить для него значение по умолчанию, например, **None**

|  |
| --- |
| from fastapi import FastAPI, Header    app = FastAPI()      @app.get("/")  def root(secret\_code: str | None = Header(default=None)):      return {"Secret-Code": secret\_code} |

**Куки**

**Установка куки**

Для установки куки на сервере у объекта **Response** и его классов-наследников применяется метод **set\_cookie()**. Этот метод принимает ряд параметров:

* **key**: ключ или имя куки. Обязательный параметр
* **value**: значение куки, значение по умолчанию - пустая строка
* **max\_age**: максимальное время жизни куки в секундах. Это может быть либо число, либо значение None (ограничивает время жизни куки текущей сессией браузера, является значением по умолчанию).
* **expires**: когда истекает действие куки. Это может быть либо число, либо значение None (ограничивает время жизни куки текущей сессией браузера, является значением по умолчанию).
* **path**: путь, для которого устанавливаются куки. Значение по умолчанию - корень веб-приложения "/"
* **domain**: домен, к которому применяются куки. значение по умолчанию None
* **secure**: устанавливает используемый протокол. Так, если имеет значение True, то куки будут посылаться на сервер только в запросе по протоколу https. Значение по умолчанию False
* **httponly**: устанавлиет доступность для скриптов javascript на клиенте. Так, значение True предотвращает доступ к куки из кода javascript на клиенте. Значение по умолчанию False
* **samesite**: устанавливает разрешения на отправку куки в кроссдоменных запросах. Так, значение lax (значение по умолчанию) указывают браузеру не посылать куки в кроссдоменных запросах.

Установим куку, которую назовем "last\_visit":

|  |
| --- |
| from fastapi import FastAPI, Response  from datetime import datetime    app = FastAPI()    @app.get("/")  def root(response: Response):      now = datetime.now()    # получаем текущую дату и время      response.set\_cookie(key="last\_visit", value=now)      return  {"message": "куки установлены"} |

В данном случае кука "last\_visit" представляет дату последнего визита и хранит текущую дату. Затем в браузере через инструменты разработчика мы можем увидеть параметры этой куки.

В примере выше объект Response передавался в качестве параметра. Но также можно установить куку и с помощью явно созданного объекта Response:

|  |
| --- |
| from fastapi import FastAPI  from fastapi.responses import JSONResponse  from datetime import datetime    app = FastAPI()      @app.get("/")  def root():      now = datetime.now()    # получаем текущую дату и время      response = JSONResponse(content={"message": "куки установлены"})      response.set\_cookie(key="last\_visit", value=now)      return  response |

**Получение куки**

Для получения куки на сервере применяется класс **fastapi.Cookie**. Например, получим выше установленную куку "last\_visit":

|  |
| --- |
| from fastapi import FastAPI, Cookie    app = FastAPI()    @app.get("/")  def root(last\_visit = Cookie()):      return  {"last visit": last\_visit} |

Для получения куки определяется параметр с именем куки, которому присваивается объект Cookie.

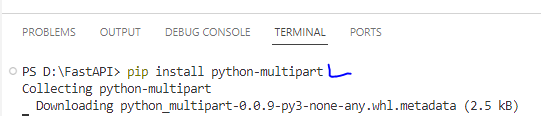
В примере выше куки была обязательна. Если ее нет в запросе, приложение выдало бы ошибку. Чтобы избежать этого, мы можем определить значение по умолчанию, в том числе None:

|  |
| --- |
| from fastapi import FastAPI, Cookie    app = FastAPI()    @app.get("/")  def root(last\_visit: str | None = Cookie(default=None)):      if last\_visit == None:          return {"message": "Это ваш первый визит на сайт"}      else:          return  {"message": f"Ваш последний визит: {last\_visit}"} |

**Отправка форм**

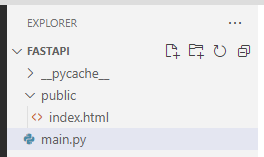
Работа с формами несколько отличается от получения на сервере данных в теле запроса. Прежде всего необходимо установить модуль **python-multipart** с помощью команды

**pip install python-multipart**



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:

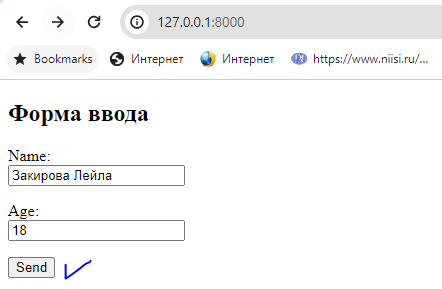
|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>My site</title>      <meta charset="utf-8" />  </head>  <body>      <h2>Форма ввода</h2>      <form action="postdata" method="post">          <p>              Name:<br>              <input name="username" />          </p>          <p>              Age:<br>              <input name="userage" type="number" />          </p>          <input type="submit" value="Send" />      </form>  </body>  </html> |

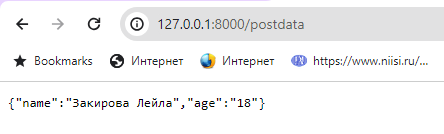
Здесь определена простейшая форма для ввода имени и возраста. И по нажатию на кнопку данные отправляются в запросе POST по адресу "postdata".

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

|  |
| --- |
| from fastapi import FastAPI, Form  from fastapi.responses import FileResponse    app = FastAPI()    @app.get("/")  def root():      return FileResponse("public/index.html")      @app.post("/postdata")  def postdata(username = Form(), userage=Form()):      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 некотрые параметры:

|  |
| --- |
| from fastapi import FastAPI, Form  from fastapi.responses import FileResponse    app = FastAPI()      @app.get("/")  def root():      return FileResponse("public/index.html")      @app.post("/postdata")  def postdata(username: str = Form(min\_length=2, max\_length=20),              userage: int =Form(ge=18, lt=111)):      return {"name": username, "age": userage} |

В данном случае параметр username должен иметь не меньше 2 и не больше 20 символов, а параметр userage 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 отсуствуют соответствующие значения:

|  |
| --- |
| from fastapi import FastAPI, Form  from fastapi.responses import FileResponse    app = FastAPI()      @app.get("/")  def root():      return FileResponse("public/index.html")      @app.post("/postdata")  def postdata(username: str = Form(default ="Undefined", min\_length=2, max\_length=20),              userage: int =Form(default=18, ge=18, lt=111)):      return {"name": username, "age": userage} |

**Отправка списков**

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

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>My site</title>      <meta charset="utf-8" />  </head>  <body>      <h2>Форма ввода</h2>      <form action="postdata" method="post">          <p>              Name:<br>              <input name="username" />          </p>          <p>              Languages:<br>              <input name="languages" /><br><br>              <input name="languages" /><br><br>              <input name="languages" /><br><br>          </p>          <input type="submit" value="Send" />      </form>  </body>  </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**:

|  |
| --- |
| from fastapi import FastAPI, Form  from fastapi.responses import FileResponse    app = FastAPI()      @app.get("/")  def root():      return FileResponse("public/index.html")      @app.post("/postdata")  def postdata(username: str = Form(),              languages: list =Form()):      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 полев ввода языка:

**Работа с базой данных через SQLAlchemy**

**Подключение к базе данных и создание таблиц**

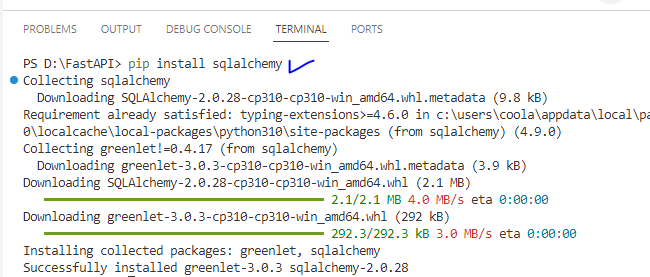
FastAPI поддерживает работу с самыми разными системами баз данных: PostgreSQL, MySQL, SQLite, Oracle, Microsoft SQL Server и т.д. Причем мы не ограничены только реалиционными базами данных и равным образом использовать и нереляционные, так называемые NoSQL-системы баз данных.

При работе с базами данных самым простым решением является использование специальных инструментов - **ORM** (Object Relational Mapper), которые позволяют абстрагироваться от строения базы данных в конкретной СУБД и позволяет автоматически связать сущности в коде Python с таблицами в базе данных. В FastAPI наиболее распространным ORM-инструментом является **SQLAlchemy ORM**

Рассмотрим работу с базой данных через **SQLAlchemy ORM** на примере SQLite, поскольку SQLite не требует установки сервера или каких-то дополнительных инструментов. Кроме того, перейти от одной СУБД к другой не составить особого труда и почти не потребует изменения кода взаимодействия с БД, за исключением пары строк кода.

Прежде всего нам надо установить сам SQLAlchemy с помощью команды

pip install sqlalchemy



**Определение строки подключения к БД**

Прежде всего для подключения к базе данных необходима строка подключения, которая включает адрес базы данных и другие различные параметры, необходимые для подключения. Для разных СУБД строка подключения может отличаться. Пример строки подключения к SQLite:

|  |
| --- |
| # строка подключения  SQLALCHEMY\_DATABASE\_URL = "[sqlite:///./sql\_app.db](sqlite://./sql_app.db)" |

Для sqlite строка подключения начинается с sqlite://, затем идет относительный или абсолютный путь к файлу:

|  |
| --- |
| sqlite:///относительный\_путь/file.db  sqlite:////абсолютный\_путь/file.db |

В данном случае это "./sql\_app.db", который указывает, что файл базы данных будет называться "sql\_app.db" и будет располагаться в текущей папке.

В зависимости от конкретной субд строка подключения будет различаться. Так, если бы мы подключись бы к базе данных PostgreSQL, то строка подключения имела бы следующий формат:

|  |
| --- |
| SQLALCHEMY\_DATABASE\_URL = "<postgresql://user:password@postgresserver/db>" |

где user - имя пользователя на сервере PostgreSQL, password - пароль, postgresserver - адрес сервера, а db - имя базы данных на сервере.

При подключении к другим системам баз данных по сути это единственная строка кода, которая потребует изменения.

**Определение движка**

После определения строки подключения создается движок SQLAlchemy с помощью функции **create\_engine()**:

|  |
| --- |
| from sqlalchemy import create\_engine    SQLALCHEMY\_DATABASE\_URL = "[sqlite:///./sql\_app.db](sqlite://./sql_app.db)"    # создание движка  engine = create\_engine(      SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}  ) |

Первый параметр функции указывает на строку подключения - то есть по сути базу данных, к которой идет подключение.

Второй параметр - дополнительные параметры подключения в виде словаря connect\_args. В частности, элемент "check\_same\_thread": False предназначен только для SQLite и указывает, что для взаимодействия с базой данных SQLite в рамках одного запроса может использоваться больше одного потока. Зачем это надо? По умолчанию SQLite разрешает взаимодействовать с БД только одному потоку, чтобы один поток обрабатывал отдельный запрос, что позволяет предотвратить использование одного потока для разных запросов. Но в FastAPI в рамках одного и того же запроса с базой данных могут взаимодействовать более одного потока. Собственно поэтому необходима подобная настройка для получения необходимых разрешений. При этом каждый запрос будет получать свой собственный сеанс подключения к базе данных через механизм внедрения зависимостей, поэтому в механизме по умолчанию нет необходимости.

При работе с другими СУБД достаточно указать только адрес подключения:

|  |
| --- |
| engine = create\_engine(SQLALCHEMY\_DATABASE\_URL) |

**Определение моделей**

Модели представляют классы, которые соответствуют определению таблиц в базе данных и объекты которых хранятся в этих таблицах. И одним из преимуществ SQLAlchemy является то, что мы можем работать таблицами через эти модели-классы языка Python, не прибегая к созданию запросов на языке SQL.

Для создания моделей необходима базовая модель, от которой потом наследуются остальные модели. Для создания базовой модели можно использовать различные способы, но самый короткий - создания класса модели с помощью функции **declarative\_base()**:

|  |
| --- |
| from sqlalchemy.ext.declarative import declarative\_base    Base = declarative\_base() |

То есть в данном случае определяется класс Base - базовая модель. Затем уже можно определить конкретные модели, данные которых будут храниться в БД. Например:

|  |
| --- |
| from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy import  Column, Integer, String    Base = declarative\_base()    class Person(Base):      \_\_tablename\_\_ = "people"        id = Column(Integer, primary\_key=True, index=True)      name = Column(String)      age = Column(Integer,) |

Здесь определена модель Person, которая представляет пользователя. Она наследуется от Base и определяет ряд атрибутов.

Атрибут **\_\_tablename\_\_** хранит имя таблицы, с которой будет сопоставляться текущая модель. То есть данные класса Person будут храниться в таблице "people".

Для хранения данных в классе модели определяются атрибуты, которые сопоставляются со столбцами таблицы. Такие атрибуты в качестве значения получаются объект **Column**. В данном случае класс Person определяет три таких атрибута: id, name и age, для которых в таблице people будут создаваться одноименные столбцы.

Класс Column с помощью параметров конструктора определяет настройки столбца в таблицы. Здесь в конструктор вначале передается тип столбца. Так, у id и age тип столбца - Integer, то есть целое число. А столбец name представляет тип String, то есть строку.

Кроме того, в конструктор можно передать ряд дополнительных параметров. Так, для столбца id передаются параметры primary\_key и index. Значение primary\_key=True указывает, что данный столбец будет представлять первичный ключ. А параметр index=True говорит, что для данного столбца будет создаваться индекс.

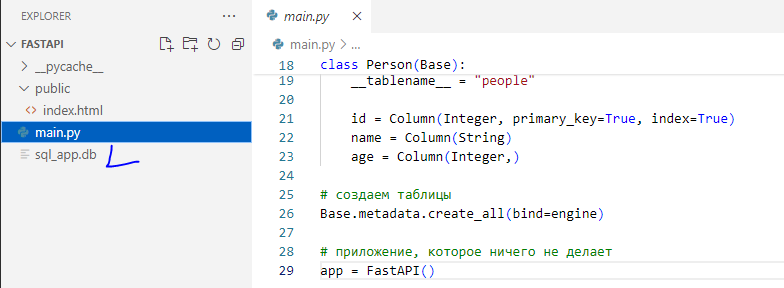
**Создание базы данных и таблиц**

Для создания базы данных и таблиц по метаданным моделей применяется метод Base.metadata.create\_all(). Его ключевой параметр - bind принимает класс, который используется для подключения к базе данных. В качестве такого класса применяется созданный ранее движок SQLAlchemy. Если база данных и все необходимые таблицы уже имеются, то метод не создает заново таблицы.

Посмотрим на простейшем примере. Все объединим и определим в файле приложения следующий код:

|  |
| --- |
| from sqlalchemy import create\_engine  from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy import  Column, Integer, String    from fastapi import FastAPI    # строка подключения  SQLALCHEMY\_DATABASE\_URL = "[sqlite:///./sql\_app.db](sqlite://./sql_app.db)"    # создаем движок SqlAlchemy  engine = create\_engine(      SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}  )  #создаем базовый класс для моделей  Base = declarative\_base()    # создаем модель, объекты которой будут храниться в бд  class Person(Base):      \_\_tablename\_\_ = "people"        id = Column(Integer, primary\_key=True, index=True)      name = Column(String)      age = Column(Integer,)    # создаем таблицы  Base.metadata.create\_all(bind=engine)    # приложение, которое ничего не делает  app = FastAPI() |

После запуска приложения в папке проекта будет создана таблица sql\_app.db:



Мы можем открыть эту базу данных с помощью любой программы для просмотра баз данных sqlite (например, с помощью [DB Browser for SQLite](https://sqlitebrowser.org/)) и увидеть содержимое бд:

**Взаимодействие с базой данных**

**Создание сессии базы данных**

Для взаимодействия с базой данных необходимо создать **сессию базы данных**, которая представляет объект **sqlalchemy.orm.Session**. Через этот объект идет вся работа с БД. Но для этого вначале надо создать класс-построитель Session с помощью функции-фабрики **sessionmaker()**

|  |
| --- |
| from sqlalchemy.orm import sessionmaker    SessionLocal = sessionmaker(autoflush=False, bind=engine) |

эта функция принимает ряд параметров, в частности, здесь применяется два параметра

**autoflush**: при значении True (значение по умолчанию) будет автоматически вызываться метод Session.flush(), который записывает все изменения в базу данных

**bind**: привязывает сессию бд к определенному движку, который применяется для установки подключения

Результатом функции является класс SessionLocal. После этого можно создать объект этого класса и через него взаимодействовать в бд:

|  |
| --- |
| from sqlalchemy.orm import sessionmaker    SessionLocal = sessionmaker(autoflush=False, bind=engine)  db = SessionLocal() |

Здесь db как раз представляет объект Session.

**CRUD**

Рассмотрим простейшие операции CRUD (Create-Read-Update-Delete).

**Добавление данных**

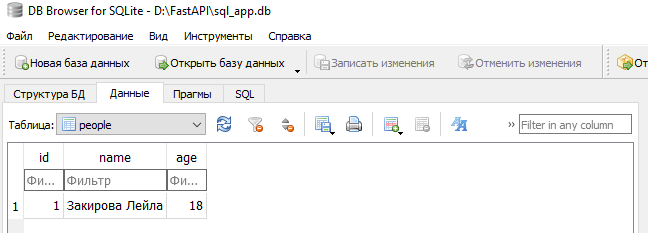
Для добавления в базу данных необходимо сначала создать объект модели, который передается в метод **add()** объекта Session. После добавления для подтверждения изменений у объекта Session вызывается метод **commit()**

|  |
| --- |
| # создаем объект Person для добавления в бд  leyla = Person(name="Закирова Лейла", age=18)  db.add(leyla)     # добавляем в бд  db.commit()     # сохраняем изменения |

Например, определим файле приложения следующий код:

|  |
| --- |
| from sqlalchemy import create\_engine  from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy.orm import sessionmaker, Session  from sqlalchemy import  Column, Integer, String    from fastapi import FastAPI    SQLALCHEMY\_DATABASE\_URL = "sqlite:///./sql\_app.db"  engine = create\_engine(SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False})    # создаем модель, объекты которой будут храниться в бд  Base = declarative\_base()  class Person(Base):      \_\_tablename\_\_ = "people"        id = Column(Integer, primary\_key=True, index=True)      name = Column(String)      age = Column(Integer,)    # создаем таблицы  Base.metadata.create\_all(bind=engine)    # создаем сессию подключения к бд  SessionLocal = sessionmaker(autoflush=False, bind=engine)  db = SessionLocal()    # создаем объект Person для добавления в бд  leyla = Person(name="Закирова Лейла", age=18)  db.add(leyla)     # добавляем в бд  db.commit()     # сохраняем изменения    print(leyla.id)   # можно получить установленный id    # приложение, которое ничего не делает  app = FastAPI() |

В данном случае создается объект Person, который добавляется в бд. Если после этого мы откроем бд, то сможем увидеть добавленный объект:



Стоит отметить, что в данным случае в классе Person атрибут id выступает в качестве первичного ключа и генерируется в самой бд при добавлении строки в таблицу. Но после добавления объекта мы можем получить значение данного атрибута:

|  |
| --- |
| print(leyla.id) |

Следует отметить, что после добавления или обновления объекта, если мы хотим использовать этот объект, обращаться к его атрибутами, то желательно, а иногда может быть необходимо, использовать метод **refresh()**, который обновляет состояние объекта:

|  |
| --- |
| # создаем объект Person для добавления в бд  leyla = Person(name="Закирова Лейла", age=18)  db.add(leyla)     # добавляем в бд  db.commit()     # сохраняем изменения  db.refresh(leyla)  # обновляем состояние объекта  print(leyla.id)   # можно получить установленный id |

Подобным образом можно добавить несколько объектов:

|  |
| --- |
| elina = Person(name="Ахтямова Элина", age=18)  elza = Person(name="Орлова Эльза", age=19)  db.add(elina)  db.add(elza)  db.commit() |

Причем метод commit() вызывается только один раз.

Однако есть надо добавить несколько объектов, то проще применить метод **add\_all()**, который добавляет список объектов:

|  |
| --- |
| # Создание массива объектов Person  people = [  Person(name="Абдуллина Диляра Ильдаровна", age=18),  Person(name="Афендин Хасан Зуфарович", age=18),  Person(name="Бикулов Артём Александрович", age=18),  Person(name="Довыдова Анна Владимировна", age=18),  Person(name="Зеленюк Максим Витальевич", age=18),  Person(name="Кольцов Денис Дмитриевич", age=18),  Person(name="Нургалиев Анвар Камилевич", age=18)  ]  db.add\_all(people)  db.commit() |

**Получение данных**

Для получения объектов из базы данных вначале у объекта Session необходимо вызывать метод **query()** - в него передается тип модели, данные которой необходимо получить:

|  |
| --- |
| db.query(Person) |

Но данный метод просто создает объект Query - некоторый запрос, который будет выполнен в будущем при непосредственном получении данных. Далее применяя к объекту Query различные методы, мы можем получить непосредственный результат. Например, если надо получить все объекты, применяется метод **all()**:

|  |
| --- |
| people = db.query(Person).all() |

Метод all возращает список объектов модели. Полный код:

|  |
| --- |
| from sqlalchemy import create\_engine  from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy.orm import sessionmaker  from sqlalchemy import  Column, Integer, String    from fastapi import FastAPI    SQLALCHEMY\_DATABASE\_URL = "[sqlite:///./sql\_app.db](sqlite://./sql_app.db)"  engine = create\_engine(SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False})      Base = declarative\_base()  class Person(Base):      \_\_tablename\_\_ = "people"        id = Column(Integer, primary\_key=True, index=True)      name = Column(String)      age = Column(Integer,)    SessionLocal = sessionmaker(autoflush=False, bind=engine)  db = SessionLocal()    # получение всех объектов  people = db.query(Person).all()  for p in people:      print(f"{p.id}.{p.name} ({p.age})")    # приложение, которое ничего не делает  app = FastAPI() |

Для получения одного объекта по id применяется метод **get()** класса Session. В качестве параметров метод получает тип модели и идентификатор объекта, который надо получить. Например, получим один объект Person, у которого id = 1:

|  |
| --- |
| # получение одного объекта по id  first\_person = db.get(Person, 1)  print(f"{first\_person.name} - {first\_person.age}") |

Для фильтрации у объекта Query применяется метод **filter()**, который принимает условие фильтрации. Например:

|  |
| --- |
| people = db.query(Person).filter(Person.age > 30).all()  for p in people:      print(f"{p.id}.{p.name} ({p.age})") |

Здесь получаем все объекты Person, у которых значение атрибута age более 30. Метод filter() также возвращает объект Query, поэтому для получения собственно списка объектов, которые соответствуют фильтру, в конце по цепочке применяется метод all()

Для получения только одного объекта применяется метод **first()** класса Query:

|  |
| --- |
| first = db.query(Person).filter(Person.id==1).first()  print(f"{first.name} ({first.age})") |

В данном случае получаем объект Person, у которого id = 1.

Стоит отметить, что методы get() и first() возвращают **None**, если объект не найден. Поэтому при получении одиночного объекта желательно проверять его на значение None.

**Обновление**

Для обновления объекта достаточно изменить значения его атрибутов и затем вызвать у объекта Session метод **commit()** для применения изменений:

|  |
| --- |
| # получаем один объект, у которого имя - leyla  leyla = db.query(Person).filter(Person.id==1).first()  print(f"{ leyla.id}.{ leyla.name} ({leyla age})")    # изменениям значения  leyla.name = "Закирова Лейлюша"  leyla.age = 17    db.commit() # сохраняем изменения    # проверяем, что изменения применены в бд - получаем один объект, у которого имя - "Закирова Лейлюша"  leylash = db.query(Person).filter(Person.id == 1).first()  print(f"{ leylash.id}.{ leylash.name} ({leylash.age})") |

**Фильтры**

1. Фильтрация по имени:

people\_with\_name\_leyla = db.query(Person).filter(Person.name == "Leyla").all()

1. Фильтрация по возрасту больше определенного значения:

people\_over\_18 = db.query(Person).filter(Person.age > 18).all()

1. Фильтрация по возрасту меньше определенного значения:

people\_under\_30 = db.query(Person).filter(Person.age < 30).all()

1. Фильтрация по нескольким условиям с использованием логического оператора AND:

specific\_person = db.query(Person).filter(Person.name == "Leyla", Person.age == 25).first()

1. Фильтрация по нескольким значениям имени с использованием оператора IN:

specific\_names = ["Alice", " Leyla", "Timur"]

people\_with\_specific\_names = db.query(Person).filter(Person.name.in\_(specific\_names)).all()

1. Фильтрация с использованием логического оператора OR:

people\_with\_name\_leyla\_or\_age\_25 = db.query(Person).filter(or\_(Person.name == "Leyla", Person.age == 25)).all()

1. Фильтрация по части имени с использованием like:

people\_with\_name\_like\_l = db.query(Person).filter(Person.name.like("%l%")).all()

1. Фильтрация по части имени без учета регистра:

people\_with\_name\_ilike\_a = db.query(Person).filter(Person.name.ilike("%a%")).all()

1. Фильтрация по ID:

person\_with\_id\_3 = db.query(Person).filter(Person.id == 3).first()

1. Фильтрация по отсутствию значения в поле:

people\_without\_age = db.query(Person).filter(Person.age.is\_(None)).all()

1. Фильтрация с использованием between для интервала возрастов:

people\_between\_20\_and\_30 = db.query(Person).filter(Person.age.between(20, 30)).all()

1. Фильтрация по значению, не равному определенному значению:

people\_not\_named\_john = db.query(Person).filter(Person.name != "John").all()

1. Фильтрация с использованием distinct для уникальных значений:

unique\_names = db.query(Person.name).distinct().all()

1. Фильтрация с использованием startswith для начала имени:

people\_with\_name\_starting\_with\_j = db.query(Person).filter(Person.name.startswith("J")).all()

1. Фильтрация с использованием endswith для окончания имени:

people\_with\_name\_ending\_with\_n = db.query(Person).filter(Person.name.endswith("n")).all()

1. Фильтрация с использованием contains для наличия подстроки в имени:

people\_with\_name\_containing\_ohn = db.query(Person).filter(Person.name.contains("ohn")).all()

1. Фильтрация с использованием in\_ для нескольких значений возраста:

specific\_ages = [25, 30, 35]

people\_with\_specific\_ages = db.query(Person).filter(Person.age.in\_(specific\_ages)).all()

1. Фильтрация по пустым значениям имени:

people\_with\_empty\_name = db.query(Person).filter(Person.name == "").all()

1. Фильтрация по непустым значениям имени:

people\_with\_non\_empty\_name = db.query(Person).filter(Person.name != "").all()

1. Фильтрация с использованием isNull для проверки наличия NULL значений в поле:

people\_with\_null\_age = db.query(Person).filter(Person.age.is\_(None)).all()

Это всего лишь некоторые примеры различных видов фильтров, которые могут быть полезны при работе с вашим классом Person. Вы можете комбинировать и изменять эти фильтры в зависимости от ваших конкретных потребностей.

**Удаление**

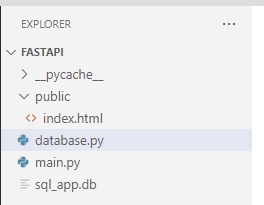
Для удаления у объекта Session применяется метод **delete()**, в который передается удаляемый объект:

|  |
| --- |
| dilyara = db.query(Person).filter(Person.id==2).first()  db.delete(dilyara)  # удаляем объект  db.commit()     # сохраняем изменения |

**Базовые операции с базой данных в веб-приложении**

  В прошлой теме были рассмотрены базовые операции с базой данных. Рассмотрим, как это будет выглядеть в рамках веб-приложения на простейшем примере.

В итоге получится следующий проект:



**Определение подключения к базе данных и моделей**

Для хранения всей инфраструктуры для работы с базой данных в папке проекта определим файл **database.py** со следующим кодом:

|  |
| --- |
| from sqlalchemy import create\_engine  from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy.orm import sessionmaker  from sqlalchemy import  Column, Integer, String    from fastapi import FastAPI    SQLALCHEMY\_DATABASE\_URL = "[sqlite:///./sql\_app.db](sqlite://./sql_app.db)"  engine = create\_engine(SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False})      Base = declarative\_base()  class Person(Base):      \_\_tablename\_\_ = "people"        id = Column(Integer, primary\_key=True, index=True)      name = Column(String)      age = Column(Integer,)    SessionLocal = sessionmaker(autoflush=False, bind=engine) |

Итак, в данном случае мы будем подключаться к базе данных SQLite, которая будет создаваться в папке проекта.

Хранимые в бд данные будут представлены классом Person с тремя атрибутами: id, name и age.

**Создание сервера**

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

|  |
| --- |
| from database import \*  from sqlalchemy.orm import Session  from fastapi import Depends, FastAPI, Body  from fastapi.responses import JSONResponse, FileResponse    # создаем таблицы  Base.metadata.create\_all(bind=engine)    app = FastAPI()    # определяем зависимость  def get\_db():      db = SessionLocal()      try:          yield db      finally:          db.close()    @app.get("/")  def main():      return FileResponse("public/index.html")    @app.get("/api/users")  def get\_people(db: Session = Depends(get\_db)):      return db.query(Person).all()    @app.get("/api/users/{id}")  def get\_person(id, db: Session = Depends(get\_db)):      # получаем пользователя по id      person = db.query(Person).filter(Person.id == id).first()      # если не найден, отправляем статусный код и сообщение об ошибке      if person==None:          return JSONResponse(status\_code=404, content={ "message": "Пользователь не найден"})      #если пользователь найден, отправляем его      return person      @app.post("/api/users")  def create\_person(data  = Body(), db: Session = Depends(get\_db)):      person = Person(name=data["name"], age=data["age"])      db.add(person)      db.commit()      db.refresh(person)      return person    @app.put("/api/users")  def edit\_person(data  = Body(), db: Session = Depends(get\_db)):        # получаем пользователя по id      person = db.query(Person).filter(Person.id == data["id"]).first()      # если не найден, отправляем статусный код и сообщение об ошибке      if person == None:          return JSONResponse(status\_code=404, content={ "message": "Пользователь не найден"})      # если пользователь найден, изменяем его данные и отправляем обратно клиенту      person.age = data["age"]      person.name = data["name"]      db.commit() # сохраняем изменения      db.refresh(person)      return person      @app.delete("/api/users/{id}")  def delete\_person(id, db: Session = Depends(get\_db)):      # получаем пользователя по id      person = db.query(Person).filter(Person.id == id).first()        # если не найден, отправляем статусный код и сообщение об ошибке      if person == None:          return JSONResponse( status\_code=404, content={ "message": "Пользователь не найден"})        # если пользователь найден, удаляем его      db.delete(person)  # удаляем объект      db.commit()     # сохраняем изменения      return person |

Разберем в общих чертах этот код. Прежде всего вначале создаем все необходимые таблицы в базе данных (если они отстуствуют)

|  |
| --- |
| Base.metadata.create\_all(bind=engine) |

Предпочтительным способом передачи сессии базы данных в функции обработки запроса представляет внедрение сессии бд в качестве зависимости. Поэтому далее создаем функцию get\_db(), через которую объект сессии базы данных будет передаваться в функцию обработки:

|  |
| --- |
| def get\_db():      db = SessionLocal()      try:          yield db      finally:          db.close() |

Здесь сначала создаем объект сессии базы данных. Затем в конструкции try..finally с помощью оператора **yield** возвращаем созданный объект. Таким образом, данный объект будет внедрен в функцию обработки запроса. Выражение **yield** будет выполняться при получении каждого нового запроса.

Помещение выражение yield в блок try позволяет получить и обработать любую ошибку, возникшую в процессе взаимодействия с базой данных.

После завершения операций с базой данных выполняется блок **finally**, в котором закрывается подключение к базе данных с помощью метода **close()**

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:

|  |
| --- |
| @app.get("/")  def main():      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":

|  |
| --- |
| @app.get("/api/users")  def get\_people(db: Session = Depends(get\_db)):      return db.query(Person).all() |

C помощью класса **Depends()** в функцию передается результат функции get\_db, то есть сессия базы данных, который передается параметру db. И через этот параметр мы сможем взаимодействовать с базой данных. Подобным образом сессия базы данных внедряется во все остальные функции-обработчики запросов.

Запрос GET предполагает получение объектов, и в данном случае с помощью метода db.query(Person).all() получаем список объектов Person, который отправляется клиенту.

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

|  |
| --- |
| @app.get("/api/users/{id}")  def get\_person(id, db: Session = Depends(get\_db)):      person = db.query(Person).filter(Person.id == id).first()      if person==None:          return JSONResponse(status\_code=404, 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 базе данных. 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:

|  |
| --- |
| @app.delete("/api/users/{id}")  def delete\_person(id, db: Session = Depends(get\_db)):      person = db.query(Person).filter(Person.id == id).first()        if person == None:          return JSONResponse( status\_code=404, content={ "message": "Пользователь не найден"})        db.delete(person)      db.commit()      return person |

Здесь действует аналогичная логика - если объект по id не найден, отправляет статусный код 404. Если же объект найден, то удаляем его из базы данных 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(), db: Session = Depends(get\_db)):      person = Person(name=data["name"], age=data["age"])      db.add(person)      db.commit()      db.refresh(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 опустим 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 добавляем в базу данных, с помощью метода db.refresh() обновляем его состояние и отправляем 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 базе данных. Если объект не найден, отправляем статусный код 404. Если объект найден, то изменяем его данные, обновляем его состояние и отправляем обратно клиенту:

|  |
| --- |
| @app.put("/api/users")  def edit\_person(data  = Body(), db: Session = Depends(get\_db)):        person = db.query(Person).filter(Person.id == data["id"]).first()      if person == None:          return JSONResponse(status\_code=404, content={ "message": "Пользователь не найден"})      person.age = data["age"]      person.name = data["name"]      db.commit() # сохраняем изменения      db.refresh(person)      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:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>My site</title>  <style>  td {padding:5px;}  button{margin: 5px;}  </style>  </head>  <body>      <h2>Список пользователей</h2>      <div>          <input type="hidden" id="userId" />          <p>              Имя:<br/>              <input id="userName" />          </p>          <p>              Возраст:<br />              <input id="userAge" type="number" />          </p>           <p>              <button id="saveBtn">Сохранить</button>              <button id="resetBtn">Сбросить</button>          </p>      </div>      <table>          <thead><tr><th>Имя</th><th>Возраст</th><th></th></tr></thead>          <tbody>          </tbody>      </table>        <script>      // Получение всех пользователей          async function getUsers() {              // отправляет запрос и получаем ответ              const response = await fetch("/api/users", {                  method: "GET",                  headers: { "Accept": "application/json" }              });              // если запрос прошел нормально              if (response.ok === true) {                  // получаем данные                  const users = await response.json();                  const rows = document.querySelector("tbody");                  // добавляем полученные элементы в таблицу                  users.forEach(user => rows.append(row(user)));              }          }          // Получение одного пользователя          async function getUser(id) {              const response = await fetch(`/api/users/${id}`, {                  method: "GET",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  document.getElementById("userId").value = user.id;                  document.getElementById("userName").value = user.name;                  document.getElementById("userAge").value = user.age;              }              else {                  // если произошла ошибка, получаем сообщение об ошибке                  const error = await response.json();                  console.log(error.message); // и выводим его на консоль              }          }          // Добавление пользователя          async function createUser(userName, userAge) {                const response = await fetch("api/users", {                  method: "POST",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector("tbody").append(row(user));              }              else {                  const error = await response.json();                  console.log(error.message);              }          }          // Изменение пользователя          async function editUser(userId, userName, userAge) {              const response = await fetch("api/users", {                  method: "PUT",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      id: userId,                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));              }              else {                  const error = await response.json();                  console.log(error.message);              }          }          // Удаление пользователя          async function deleteUser(id) {              const response = await fetch(`/api/users/${id}`, {                  method: "DELETE",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector(`tr[data-rowid='${user.id}']`).remove();              }              else {                  const error = await response.json();                  console.log(error.message);              }          }            // сброс данных формы после отправки          function reset() {              document.getElementById("userId").value =              document.getElementById("userName").value =              document.getElementById("userAge").value = "";          }          // создание строки для таблицы          function row(user) {                const tr = document.createElement("tr");              tr.setAttribute("data-rowid", user.id);                const nameTd = document.createElement("td");              nameTd.append(user.name);              tr.append(nameTd);                const ageTd = document.createElement("td");              ageTd.append(user.age);              tr.append(ageTd);                const linksTd = document.createElement("td");                const editLink = document.createElement("button");              editLink.append("Изменить");              editLink.addEventListener("click", async() => await getUser(user.id));              linksTd.append(editLink);                const removeLink = document.createElement("button");              removeLink.append("Удалить");              removeLink.addEventListener("click", async () => await deleteUser(user.id));                linksTd.append(removeLink);              tr.appendChild(linksTd);                return tr;          }          // сброс значений формы          document.getElementById("resetBtn").addEventListener("click", () =>  reset());            // отправка формы          document.getElementById("saveBtn").addEventListener("click", async () => {                const id = document.getElementById("userId").value;              const name = document.getElementById("userName").value;              const age = document.getElementById("userAge").value;              if (id === "")                  await createUser(name, age);              else                  await editUser(id, name, age);              reset();          });            // загрузка пользователей          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():

|  |
| --- |
| async function getUsers() {      // отправляет запрос и получаем ответ      const response = await fetch("/api/users", {          method: "GET",          headers: { "Accept": "application/json" }      });      // если запрос прошел нормально      if (response.ok === true) {          // получаем данные          const users = await response.json();          const rows = document.querySelector("tbody");          // добавляем полученные элементы в таблицу          users.forEach(user => rows.append(row(user)));      }  } |

Для добавления строк в таблицу используется функция row(), 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:

|  |
| --- |
| async function getUser(id) {      const response = await fetch(`/api/users/${id}`, {          method: "GET",          headers: { "Accept": "application/json" }      });      if (response.ok === true) {          const user = await response.json();          document.getElementById("userId").value = user.id;          document.getElementById("userName").value = user.name;          document.getElementById("userAge").value = user.age;      }      else {          // если произошла ошибка, получаем сообщение об ошибке          const error = await response.json();          console.log(error.message); // и выводим его на консоль      }  } |

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:

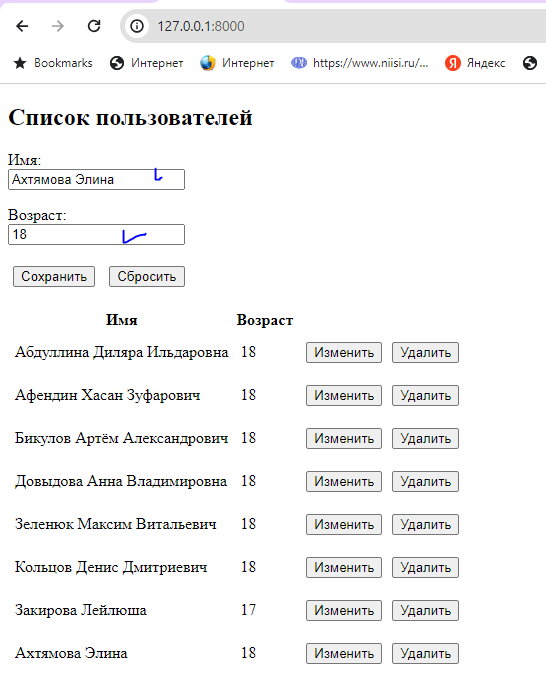
|  |
| --- |
| async function createUser(userName, userAge) {        const response = await fetch("api/users", {          method: "POST",          headers: { "Accept": "application/json", "Content-Type": "application/json" },          body: JSON.stringify({              name: userName,              age: parseInt(userAge, 10)          })      });      if (response.ok === true) {          const user = await response.json();          document.querySelector("tbody").append(row(user));      }      else {          const error = await response.json();          console.log(error.message);      }  } |

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

|  |
| --- |
| async function editUser(userId, userName, userAge) {      const response = await fetch("api/users", {          method: "PUT",          headers: { "Accept": "application/json", "Content-Type": "application/json" },          body: JSON.stringify({              id: userId,              name: userName,              age: parseInt(userAge, 10)          })      });      if (response.ok === true) {          const user = await response.json();          document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));      }      else {          const error = await response.json();          console.log(error.message);      }  } |

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:



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

**Swagger**

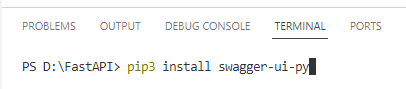
Swagger - это инструмент для документирования и тестирования веб-сервисов. Он позволяет создавать интерактивную документацию API, которая описывает доступные эндпоинты, параметры запросов, форматы ответов и т.д.

Для использования Swagger в FastAPI (Python) вы можете воспользоваться встроенной поддержкой Swagger. FastAPI автоматически генерирует спецификацию OpenAPI (ранее известную как Swagger) на основе аннотаций Python в вашем коде.

Для использования Swagger UI в вашем проекте сначала установите пакет swagger-ui-py с помощью следующей команды:

**pip3 install swagger-ui-py**

После установки этого пакета вы сможете интегрировать Swagger UI в ваше приложение для отображения документации вашего API.



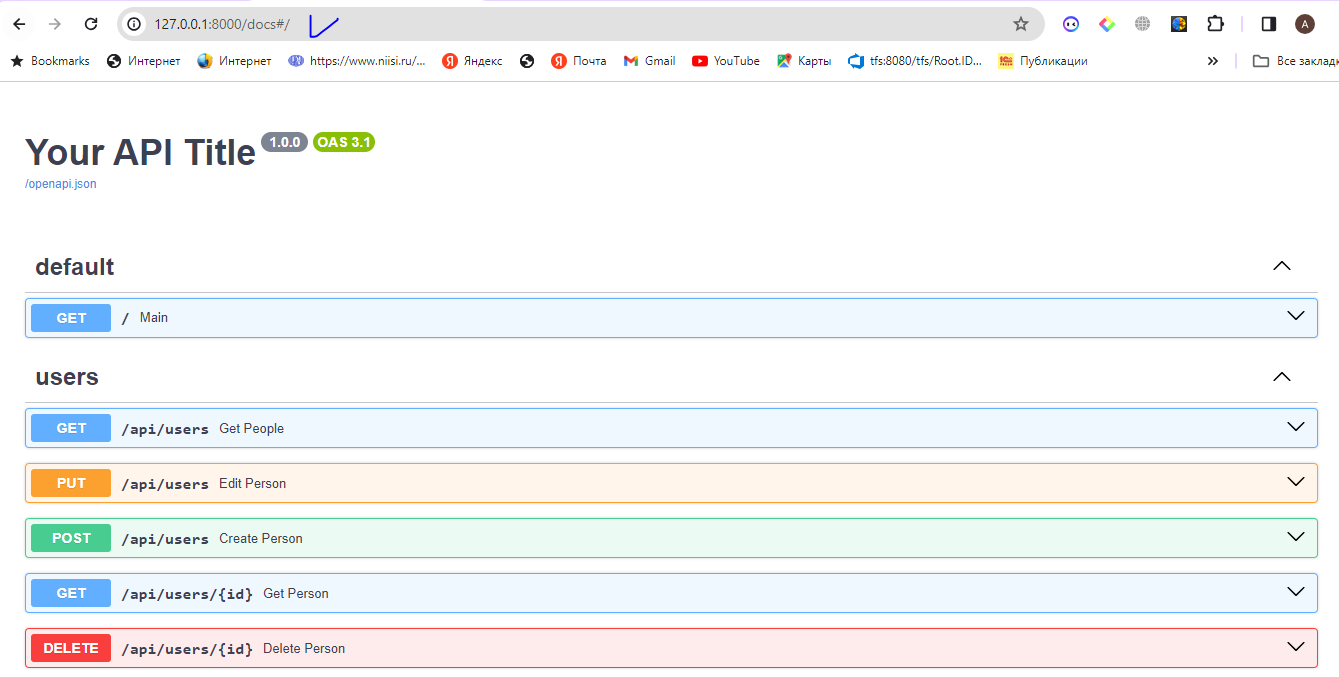
Для добавления Swagger документации к нашему FastAPI приложению, вам нужно внести следующие изменения:

* Добавить теги к каждому эндпоинту с помощью аргумента tags для группировки эндпоинтов в Swagger.
* Добавить документацию к каждому эндпоинту с помощью многострочных строк (тройные кавычки """) и описания функций.
* Добавить зависимости и аргументы функций для тел запросов с помощью модуля Body.

Пример кода с этими изменениями для Swagger документации:

|  |
| --- |
| from fastapi import FastAPI, Depends, Body  from fastapi.responses import JSONResponse, FileResponse  from sqlalchemy.orm import Session  from fastapi.openapi.utils import get\_openapi  from database import \*  from sqlalchemy.orm import Session  from fastapi import Depends, FastAPI, Body  from fastapi.responses import JSONResponse, FileResponse  from fastapi.openapi.docs import get\_swagger\_ui\_html  from database import Base  # создаем таблицы  Base.metadata.create\_all(bind=engine)  app = FastAPI()  # определяем зависимость  def get\_db():      db = SessionLocal()      try:          yield db      finally:          db.close()  @app.get("/")  def main():      return FileResponse("public/index.html")  @app.get("/api/users", tags=["users"])  def get\_people(db: Session = Depends(get\_db)):      """      Получение списка пользователей.      """      return db.query(Person).all()  @app.get("/api/users/{id}", tags=["users"])  def get\_person(id, db: Session = Depends(get\_db)):      """      Получение информации о пользователе по ID.      """      person = db.query(Person).filter(Person.id == id).first()      if person is None:          return JSONResponse(status\_code=404, content={"message": "Пользователь не найден"})      return person  @app.post("/api/users", tags=["users"])  def create\_person(data: dict = Body(...), db: Session = Depends(get\_db)):      """      Создание нового пользователя.      """      person = Person(name=data["name"], age=data["age"])      db.add(person)      db.commit()      db.refresh(person)      return person  @app.put("/api/users", tags=["users"])  def edit\_person(data: dict = Body(...), db: Session = Depends(get\_db)):      """      Редактирование информации о пользователе.      """      person = db.query(Person).filter(Person.id == data["id"]).first()      if person is None:          return JSONResponse(status\_code=404, content={"message": "Пользователь не найден"})        person.age = data["age"]      person.name = data["name"]      db.commit()      db.refresh(person)      return person  @app.delete("/api/users/{id}", tags=["users"])  def delete\_person(id, db: Session = Depends(get\_db)):      """      Удаление пользователя по ID.      """      person = db.query(Person).filter(Person.id == id).first()        if person is None:          return JSONResponse(status\_code=404, content={"message": "Пользователь не найден"})        db.delete(person)      db.commit()      return person  def custom\_openapi():      if app.openapi\_schema:          return app.openapi\_schema      openapi\_schema = get\_openapi(          title="Your API Title",          version="1.0.0",          routes=app.routes,      )      app.openapi\_schema = openapi\_schema      return app.openapi\_schema  app.openapi = custom\_openapi  @app.get("/docs", include\_in\_schema=False)  async def custom\_swagger\_ui\_html():      return get\_swagger\_ui\_html(openapi\_url="/openapi.json", title="docs")  @app.get("/openapi.json", include\_in\_schema=False)  async def get\_open\_api\_endpoint():      return app.openapi()  if \_\_name\_\_ == "\_\_main\_\_":      import uvicorn      uvicorn.run(app, host="0.0.0.0", port=8000) |

Запускаем проект



**Использовани Postman**

Цели модуля

• Используйте Postman для взаимодействия с микросервисами.

**Использовани Postman**

Postman - это популярный инструмент для тестирования и разработки API. Он предоставляет удобный интерфейс для создания, отправки и отладки HTTP-запросов, а также для проверки работы веб-сервисов и API.

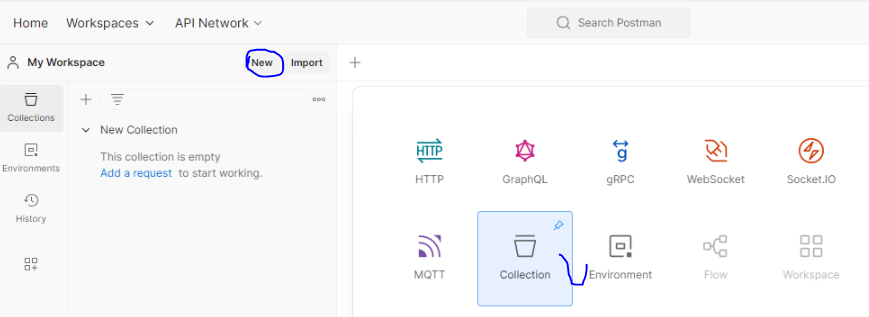
**Зачем использовать Postman?:**

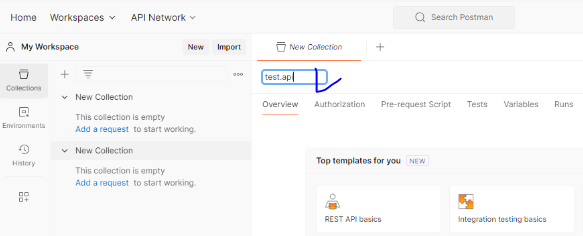
* Тестирование API: Postman позволяет отправлять различные типы запросов (GET, POST, PUT, DELETE и т. д.) на API-эндпоинты и проверять ответы на корректность.
* Отладка: Инструмент обеспечивает возможность отслеживания и анализа ответов API, а также проверку структуры возвращаемых данных.
* Автоматизация тестирования: Postman позволяет создавать и запускать коллекции запросов для автоматизации тестирования API.
* Работа с коллаборацией: Инструмент предоставляет возможность совместной работы над тестами и коллекциями запросов в команде.
* Мониторинг: Postman предоставляет инструменты для мониторинга API и проверки их доступности.

В целом, Postman упрощает процесс тестирования, отладки и разработки API, делая его более эффективным и удобным для разработчиков и тестировщиков.

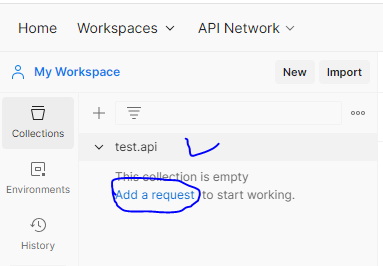
Сначала давайте скачаем Postman с официального сайта и установим его: <https://www.postman.com/downloads/>

После установки откроем Postman и создадим новое рабочее пространство. Затем перейдем на вкладку "Collection" и нажмем кнопку "New". Появится новое окно, где мы выберем "Collection" назовем «test.api».

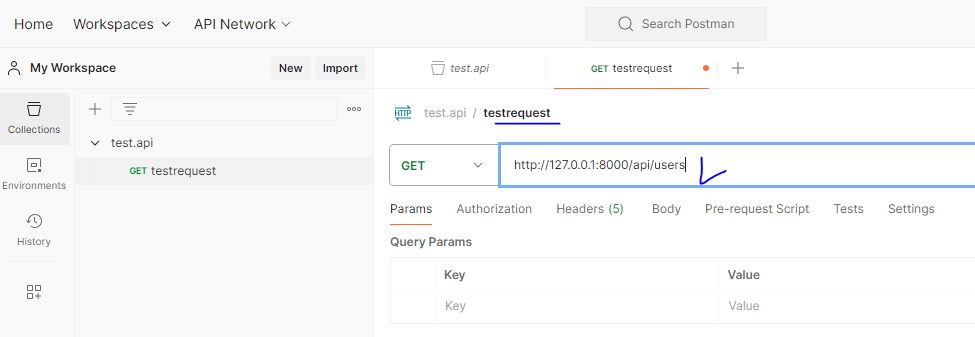




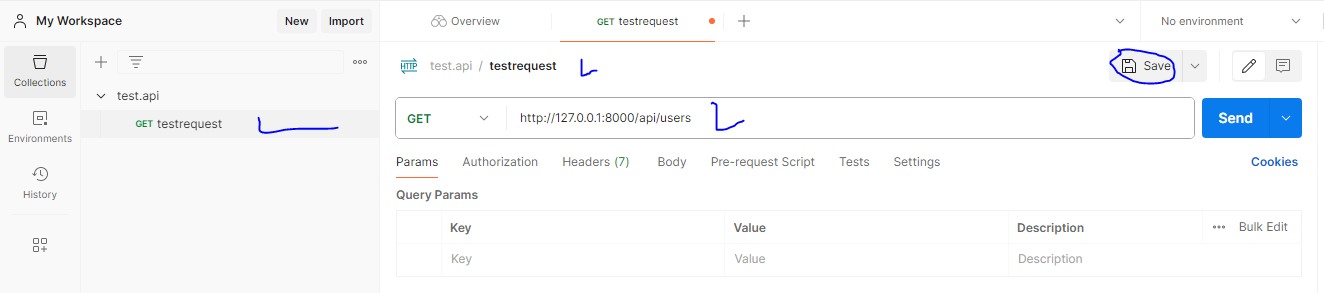
В только что созданной коллекции нажмите кнопку "Добавить запрос" и назовите его "testrequest".



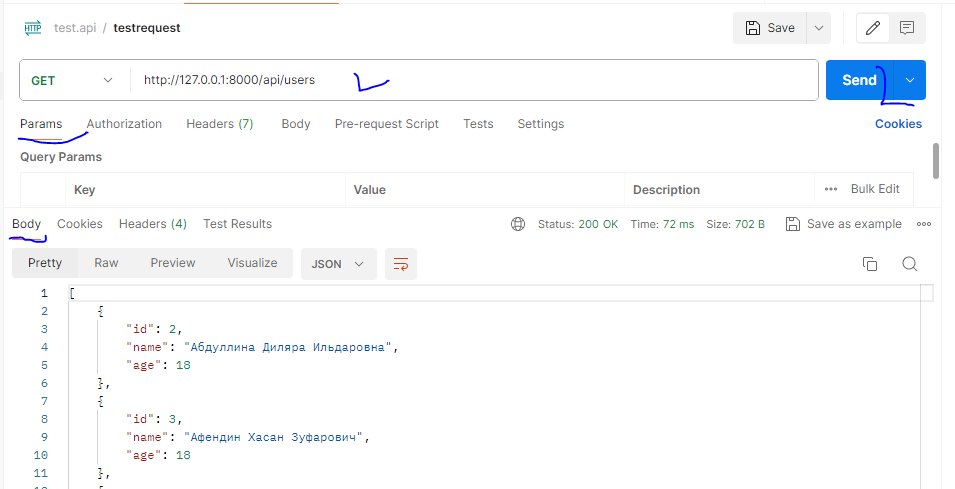
В поле "Введите URL или вставьте текст" введите URL адрес Get запроса <http://127.0.0.1:8000/api/users>

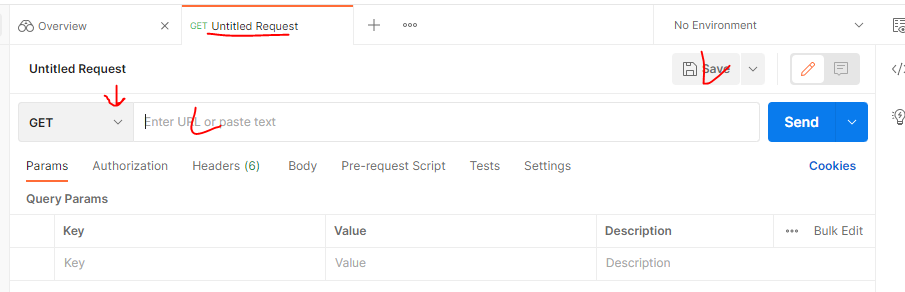


Таким образом, получим следующее. Нажмем кнопку "Save" для сохранения запроса.

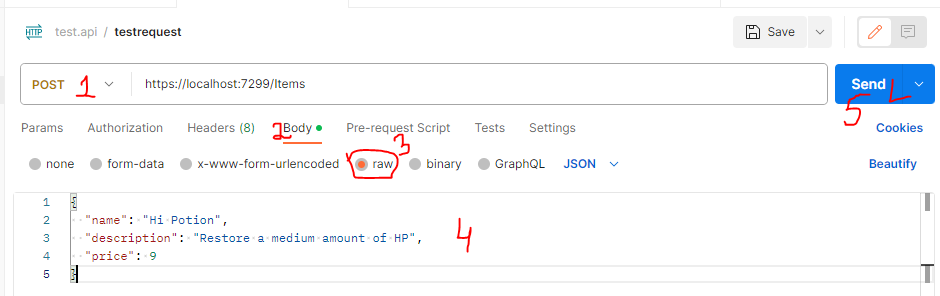


После этого, если мы нажмем кнопку "Send", то получим:



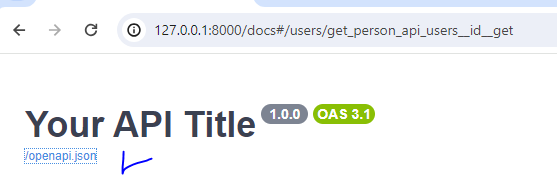


Чтобы добавить записи через Postman поступим так:



**Добавление базовый URL**

Для добавления базового URL запустим проект и перейдем по ссылке https://localhost:7299/swagger/v1/swagger.json.

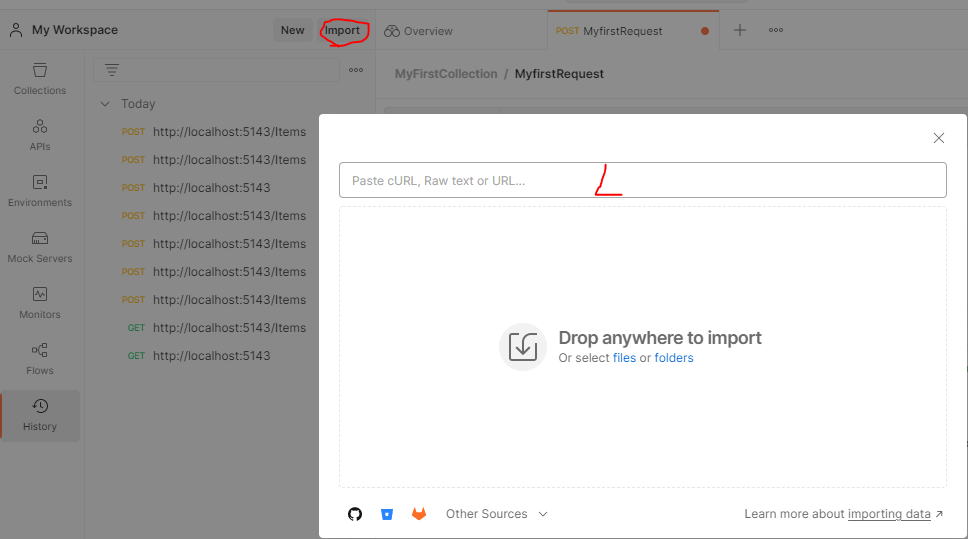


В результате откроется файл openapi.json , который содержит следующее:

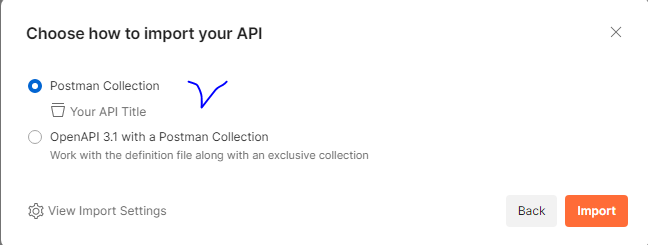


Здесь, скопируем адреса «http://127.0.0.1:8000/openapi.json»

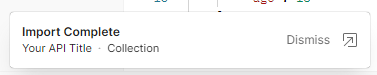
Переходим в PostMan нажимая Import и в  поле URL поставим данного адреса



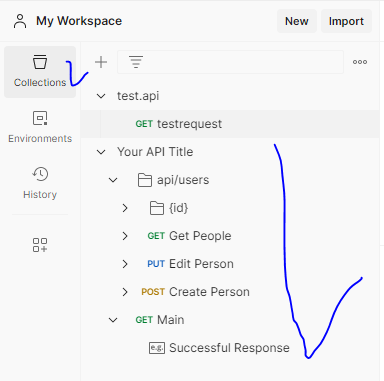
В следующее окно выберем Postman Collection



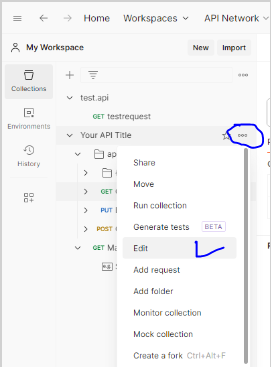
Если импорт успешно пройдеть, получим следующий результат:



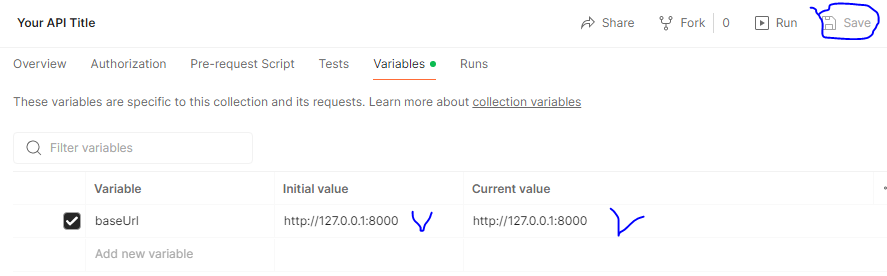
После успешных импортов можно видеть седеющего результата



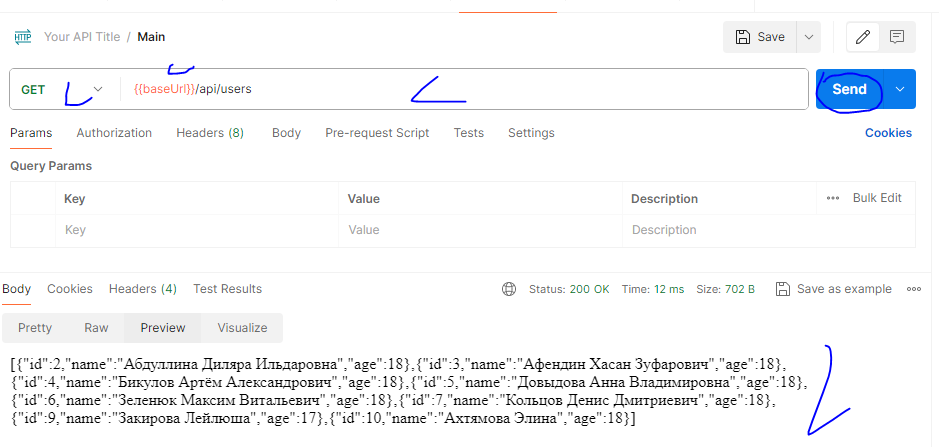
Теперь нажмем значок "..." в конце данной импортированной коллекции и выберем "Edit".



Затем перейдем на вкладку "Variables" и добавим URL-адреса.



Теперь, если мы запустим команду, все должно работать хорошо.



Таким образом, мы можем тестировать другие команды.

**Литература**

1. <https://fastapi.tiangolo.com/ru/>
2. <https://worksolutions.ru/useful/fastapi-vs-django/>
3. <https://pythonru.com/biblioteki/znakomstvo-s-fastapi>
4. <https://code.visualstudio.com/docs/python/tutorial-fastapi>
5. <https://metanit.com/>