# Описание

1. Удобный фреймворк для создания бэкенда С АВТОМАТИЧЕСКОЙ генерацией сваггера по умолчанию.

# Quick start

### Базовый сервис

from typing import Union

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

return {"Hello": "World"}

@app.get("/items/{item\_id}")

def read\_item(item\_id: int, q: Union[str, None] = None):

return {"item\_id": item\_id, "q": q}

Запуск с помощью

fastapi dev app.py – эта команда автоматически настраивает параметры сервера

#### Но лучше

from typing import Union

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

return {"Hello": "World"}

@app.get("/items/{item\_id}")

def read\_item(item\_id: int, q: Union[str, None] = None):

return {"item\_id": item\_id, "q": q}

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host=0.0.0.0, port=8000)

# Общий механизм работы

1. Сервер: прослушивание портов делает Uvicorn
2. Эндпоинты: ФастАПИ принимает запросы от uvicorn обрабатывает их как эндпоинты

# Сервер

Если у тебя есть структура проекта:

myapp/

├── mod/

│ ├── \_\_init\_\_.py

│ └── main.py # <- Здесь `app = FastAPI()`

И в main.py определено ASGI-приложение:

from fastapi import FastAPI

app = FastAPI() # <- Это ASGI-приложение (app)

То можно запустить Uvicorn

Из консоли:

uvicorn mod.main:app

Из файла:

if \_\_name\_\_ = “\_\_main”:

uvicorn.run(“mod.main:app”, port = …)

Если более сложная структура то как file.file.mod.main:app

# Разработка эндпоинтов FastAPI

## Общая структура

1. Типичная структура выглядит следующим образом:

from fastapi import FastAPI

app = FastAPI()

@app.get("/)

def endpoint1():

return "Hello"

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host=0.0.0.0, port=8000)

## Эндпоинт

1. Эндпоинт всегда состоит из

* функции, которая выполняет функционал эндпоинта
* FastAPI декоратора, который оборачивает функцию в требуемый функционал для обработки запросов.

Также декоратор содержит путь path, о чем подробнее в Части эндпоинта

Реализация эндпоинта в коде выглядит так:

@app.get("/)

def endpoint1():

return "Hello"

### Части эндпоинта

#### Декоратор эндпоинта

1. Путь так так

@app.get("/)

1. Путь может иметь параметры о чем подробнее в Обработка запросов эндпоинтом

#### Функция эндпоинта

1. Функция эндпоинта представляет собой алгоритм действий необходимых совершить при получении запроса на этот эндпоинт.

Этот алгоритм может зависить от того какой запрос придёт – то есть для разных запросов разные алгоритмы действий

def endpoint1():

return "Hello"

1. Аргументами функции эндпоинта являются – информация из http запроса (параметры и данные запроса), пришедшего к этому эндпоинту.

Об этом подробнее в Обработка запросов эндпоинтом.

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Обработка запросов эндпоинтом

1. Общая схема
2. Uvicorn – FastAPI сервер обслуживает внешние запросы\*, приходящие на него.

\* Внешними запросами в основном явлются http/https, но также поддерживается websocket

1. Uvicorn принимает запрос и передаёт его FastAPI (ASGI).
2. Проверяя path http запроса и сравнивая с декларируемыми путями эндпоинтов FastAPI находит нужный (проверка происходит сверху вниз).

Таким образом внешний запрос приходит на конкретный эндпоинт.

### Извлечение данных из запроса, приходящего на эндпоинт

1. Извлечение данных происходит следующим образом:
2. Эндпоинт FastAPI получает HTTP-запрос
3. После получения фастапи анализирует сигнатуру функции обработчика (1) и по аннотациям типов определяет, какие данные из запроса нужно подставить в аргументы.   
     
   def endpoint( token: str = Header(), data: str = Body() ):

…

1. Таким образом, из http/https запроса, в соотвествии с их устройством, можно извлечь следующие данные:

* Path-параметры — части URL, определённые в маршруте (/users/{id} ).
* Query-параметры — данные после ? в URL (/search?q=abc).
* Header-параметры — HTTP-заголовки (Authorization, User-Agent).
* Body-данные — тело запроса (JSON, form-data, raw text, файлы).  
    
  А также:
* Cookies — значения cookie, переданные клиентом.
* Зависимости (Depends) — любые дополнительные объекты, которые можно «впрыснуть» в эндпоинт (например, подключение к БД, проверки прав, декодированный токен).
* Request/Response объекты — сам объект Request или Response, если указать их в сигнатуре.

Подробнее о том, как это реализовано в коде рассмотрено в Извлечение данных из http.

#### Извлечение данных из http

1. Запрос, содержащий все основные типы парметров выглядит так:

|  |  |
| --- | --- |
| POST /api/users?limit=10&page=1 HTTP/1.1 | Path  (Path, query parameters) |
| Host: example.com  Content-Type: application/json  Authorization: Bearer abc123def456  User-Agent: Mozilla/5.0 | Header  (Header parameters) |
| {  "name": "John Doe",  "email": "john@example.com",  "age": 30,  "isActive": true  } | Body |

##### Path

1. http://127.0.0.1:8000/items/3

@app.get( "/items/{item\_id}" )

async def read\_item(item\_id: int):

return {"item\_id": item\_id}

##### Query

1. При объявлении других параметров функции, которые не являются частью параметров пути, они автоматически интерпретируются как query параметры.
2. http://127.0.0.1:8000/items/?skip=0&limit=10:

@app.get("/items/")

async def read\_item(skip: int = 0, limit: int = 10):

return fake\_items\_db[skip : skip + limit]

1. http://127.0.0.1:8000/items/foo?short=1:

@app.get("/items/{item\_id}")

async def read\_item(item\_id: str, q: str | None = None, short: bool = False):

item = {"item\_id": item\_id}

if q:

item.update({"q": q})

if not short:

item.update(

{"description": "This is an amazing item that has a long description"}

)

return item

Можно ещё так:

# Старый синтаксис (все еще очень популярен)

@app.get("/items/{item\_id}")

async def read\_item(

item\_id: int = Path(..., description="ID item"), # Просто и понятно

skip: int = Query(0, description="Skip items") # Без Annotated

):

return {"item\_id": item\_id, "skip": skip}

##### Header

1. 1

from typing import Annotated

from fastapi import FastAPI, Header

@app.get("/items/")

async def read\_items(user\_agent: Annotated[str | None, Header()] = None):

return {"User-Agent": user\_agent}

##### Body

1. Стандартный способ через Body()

from fastapi import FastAPI, Body

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

@app.post("/items/")

async def create\_item(item: Item = Body(..., example={

"name": "Book",

"description": "A good book",

"price": 12.5,

"tax": 1.5

})):

return item

1. Через Annotated:

from typing import Annotated

from fastapi import FastAPI, Body

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

@app.post("/items/")

async def create\_item(

item: Annotated[

Item,

Body(..., example={

"name": "Book",

"description": "A good book",

"price": 12.5,

"tax": 1.5

})

]

):

return item

1. Только через Pydantic

Нельзя добавить примеры, описание и прочие вещи для документации

Подходит для очень простых случаев  
  
Любая Pydantic модель в аргументах определяется как формат данных, которые нужно извлечь из Body запроса

Если аргумент — это Pydantic-модель (BaseModel), то FastAPI автоматически предполагает, что данные должны прийти из тела запроса (body).

from fastapi import FastAPI

from pydantic import BaseModel

Item(BaseModel):

name: str

description: str | None = None

price: float tax: float | None = None

app = FastAPI()

@app.post("/items/")

async def create\_item(item: Item):

return item

##### Более сложные случаи

1. Так если требуется извлечь сразу несколько заголовков можно сделать отдельную pydantic model для этого  
     
   Подробнее об этом ([Header Parameter Models - FastAPI](https://fastapi.tiangolo.com/tutorial/header-param-models/#header-parameters-with-a-pydantic-model)) и многом другом в см в документации FastAPI [Learn - FastAPI](https://fastapi.tiangolo.com/learn/)

##### Комментарии

###### Имя параметра http запроса и имя аргумента функции

1. Совпадение по имени  
   FastAPI по умолчанию связывает имя аргумента функции с параметром запроса:

* Path → имя из маршрута (/users/{user\_id} → user\_id).
* Query → имя из строки запроса (?q=... → q).
* Header → имя аргумента, где “\_” автоматически заменяется на “-“ .

user\_agent→ User-Agent

* Cookie → имя cookie = имя аргумента.

1. Использование alias  
   Однако имя параметра запроса может отличается от имени аргумента функции, можно указать alias:

from typing import Annotated

from fastapi import Query

async def search(keyword: Annotated[str, Query(alias="q")]):

return {"kw": keyword}

Запрос /search?q=test заполнит аргумент keyword.

###### Статическая типизация параметров и данных из http запроса

1. Как описано выше фастапи определяет то какую информацию нужно извлечь из запроса по сигнатуре функции обработчика эндпоинта
2. В связи с этим часто не просто прописывают все необходимые аргументы для параметров path и query + header или body

а используют для них специальные маркеры, которые помогают в создании swagger документации и более детально определяют откуда взять данные (например какой конкретно хэдер извлечь)

1. Современный и практичный способ ввести такие маркеры это – стандартный инструмент пайтона **Annotated**  из библиотеки typing.

Более подробно про annotated можно узнать в Python 3

Пример:

@app.get("/users/{user\_id}")

async def get\_user(

user\_id: Annotated[int, Path(title="ID пользователя", ge=1)]

):

return {"user\_id": user\_id}

В этом примере для user\_id добавляется аннотация типа и метаданные о том, из какой части http передать инфомацию в этот аргумент

1. В данном случае из http информация берется из path параметров и передаётся в аргумент
2. А также в swagger документации эта переменная будет называться ID пользователя
3. А ещё к тому добавляется валидация по значению: значение параметра должно быть больше чем 1 (ge – greater pr equal 1).

Запрос /users/5 → ✅ user\_id=5

Запрос /users/0 → ❌ ошибка валидации

### Возврщение ответа на запрос

#### Http протокол

1. В http протоколе сервер при получении запроса должен вернуть ответ (response), имеющий следующую структуру

|  |  |
| --- | --- |
| HTTP/1.1 200 OK | VERSION STATUS\_CODE REASON |
| Content-Type: application/json Content-Length: 85  Server: nginx/1.18.0  Date: Tue, 02 Sep 2025 12:00 GMT Set-Cookie: sessionId=xyz789 | Header  (Header parameters)  keys: value |
|  | Пустая строка отделяющая header от body |
| {  "name": "John Doe",  "email": "john@example.com",  "age": 30,  "isActive": true  } | Body  Любой формат данных (может быть пустым) |

1. Фастапи в эндпоинте конструирует ответ на запрос после return и из данных которые мы передали в return

@app.get("/)

def endpoint1():

return "Hello"

##### По умолчанию

from fastapi import FastAPI

from fastapi.responses import HTMLResponse, PlainTextResponse, Response

app = FastAPI()

# По умолчанию FastAPI вернёт JSON

@app.get("/json")

def get\_json():

return {"message": "hello"} # {"message":"hello"}

Будет сформирован такой http response:

HTTP/1.1 200 OK

content-type: application/json

content-length: 21

date: Tue, 02 Sep 2025 12:00:00 GMT

server: uvicorn

{"message":"hello"}

##### Через response\_class

1. Через response\_class

# Возврат JSON

@app.get("/text", response\_class=JSONResponse)

def get\_text():

return {“key”: “value”,

“key2”: “value2”}

# Возврат PlainText

@app.get("/text", response\_class=PlainTextResponse)

def get\_text():

return "just plain text"

# Возврат HTML

@app.get("/html", response\_class=HTMLResponse)

def get\_html():

return """

<html>

<head><title>Test</title></head>

<body><h1>Hello, HTML!</h1></body>

</html>

"""

##### Через классы ..Response в return

1. Через классы ..Response в return

# PlainText

@app.get("/manual-text")

def manual\_text():

content = "Manual plain text"

return PlainTextResponse(content=content, status\_code=200)

# HTMLResponse

@app.get("/manual-html")

def manual\_html():

content = "<h2>Manual HTML</h2>"

return HTMLResponse(content=content, status\_code=200)

##### Чистый Response

1. Нужно указывать media\_type, code и все остальное самостоятельно

@app.get("/raw-response")

def raw():

# Полный контроль: заголовки, тело, код

return Response(content="raw bytes", media\_type="application/octet-stream")