

# FastAPI: основы | Я.Шпора

**FastAPI** — асинхронный веб-фреймворк.

Установка библиотеки:

```
pip install fastapi==0.78.0
```

Пример минимального приложения на FastAPI; код можно разместить в файле с любым именем, но обычно это файл *main.py*:

```
from fastapi import FastAPI

# Создание объекта приложения.
app = FastAPI()

# Декоратор, определяющий, что GET-запросы к основному URL приложения
# должны обрабатываться этой функцией.
@app.get('/')
def read_root():
    return {'Hello': 'FastAPI'}
```

Для обработки других типов запросов в декораторе можно указать нужный метод. Например, для обработки POST-запросов декоратор будет выглядеть так:

```
@app.post(<адрес_запроса>)
```

Для запуска приложения на FastAPI понадобится веб-сервер Uvicorn:

```
pip install "uvicorn[standard]==0.17.6"
```

Запуск Uvicorn:

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

Флаг `--reload` автоматически перезапускает сервер, если код приложения был изменен; это удобно при разработке.

При запуске сервера в консоль будет выведена служебная информация и адрес главной страницы приложения, обычно это `http://127.0.0.1:8000`.

## Структура типичного проекта

На схеме не отображены директории для git/venv/Docker и пустые файлы `__init__.py`

```
папка проекта
├── .env ..... Секреты. uvicorn --reload не следит за этим файлом
├── the_app.db ..... База
├── app/ .....
│   ├── core/ .....
│   │   ├── config.py ..... Настройки проекта
│   │   └── db.py ..... Инициализация базы, предок для моделей, сессия и пр.
```

```

|   ├── models/ ..... Модели
|   |   ├── __init__.py ..... Пред-импорт моделей. Трюк для показа всех моделей Алхимии.
|   |   └── the_models.py .....
|   ├── crud/ ..... Элементарные операции с базой
|   |   ├── base_crud.py ..... Общие действия для всех моделей
|   |   └── the_models_crud.py · Уточнённые действия для конкретной модели
|   ├── schemas/ ..... Схемы JSON-данных
|   |   └── the_schemas.py ....
|   ├── api/ ..... API проекта
|   |   └── the_endpoints.py ..
|   └── main.py ..... Глобальные объекты
└── alembic/ ..... Миграции для базы. Папка появится автоматически.
    ├── env.py ..... Настройки. Нужно руками подключить секреты и модели.
    └── alembic.ini ..... Настройки миграций. Файл появится автоматически.

```

## Интерактивная документация

FastAPI генерирует интерактивную документацию к проекту:

- в формате Swagger — она доступна по адресу <http://127.0.0.1:8000/docs>;
- в формате ReDoc — она доступна по адресу <http://127.0.0.1:8000/redoc>.

При необходимости, документацию можно отключить:

```
app = FastAPI(docs_url=None, redoc_url=None)
```

Можно изменить адрес, по которому доступна документация:

```
app = FastAPI(docs_url='/swagger')
```

Теперь Swagger будет доступен по адресу `http://127.0.0.1:8000/swagger`.

💡 Параметр `docs_url` или `redoc_url` должен начинаться со слеша.

## Обработка параметров пути (path-параметров)

Path-параметр — это относительный адрес, часть URL, указанная после имени сайта. Например:

```
http://127.0.0.1:8000/Andrey    # Andrey - это path-параметр.  
http://shop.not/catalog/tv      # catalog/tv - это path-параметр.
```

Для передачи параметра пути во view-функцию необходимо указать его в фигурных скобках декоратора и в параметрах view-функции:

```
...
```

```
@app.get('/{name}') # Указываем path-параметр name.  
# Во view-функцию передаём параметр с тем же именем:
```

```
def greetings(name):  
    return {'Hello': name}
```

Код лучше аннотировать, это даёт двойную выгоду:

1. В Swagger будет доступна информация о типе аннотируемого параметра.
2. В IDE будут работать подсказки.

Таким образом, приведённый выше код лучше оформить так:

```
...  
  
@app.get('/{name}')  
def greetings(name: str) -> dict[str, str]:  
    return {'Hello': name}
```

## Обработка параметров запроса (query-параметров)

В любом типе HTTP-запросов можно передать GET-параметры — их указывают в адресе запроса после символа `?`. GET-параметры иначе называют «параметры запроса» или «query-параметры».

Параметры запроса передаются в формате `ключ=значение` и отделяются друг от друга амперсандом `&`.

```
http://127.0.0.1:8000/Ivan?surname=Petrov&age=18
```

В ссылке два параметра запроса: `surname` и `age`.

Такие параметры напрямую передают в параметры view-функции:

```
...

@app.get('/{name}')
def greetings(name: str, surname: str, age: int) -> dict[str, str]:
    ...
```

## Необязательные параметры

Необязательные параметры view-функции можно аннотировать типом `Optional` из стандартной библиотеки `typing`:

```
...

@app.get('/{name}')
def greetings(
    name: str, surname: str, age: Optional[int] = None
) -> dict[str, str]:
    ...
```

Есть и другой способ: необязательным параметрам можно присваивать значение по умолчанию.

```
...

@app.get('/{name}')
```

```
def greetings(  
    name: str, surname: str, age: int = 10  
) -> dict[str, str]:  
    ...
```

## Булевы параметры

FastAPI гибко обрабатывает булевы параметры:

```
...  
  
@app.get('/{name}')  
def greetings(  
    name: str, surname: str, age: int = 10,  
    is_staff: bool = False  
) -> dict[str, str]:
```

В запросе, в качестве значений для `is_staff` можно передавать не только `True` и `False`, но и некоторые другие:

- в значении `True` можно применять строки `'1', 'on', 't', 'true', 'y', 'yes'`;
- в значении `False` — `'0', 'off', 'f', 'false', 'n', 'no'`.

Таким образом, функция успешно обработает запросы с такими, например, значениями GET-параметра `is_staff`:

```
http://127.0.0.1:8000/Ivan?surname=Petrov&is_staff=no  
http://127.0.0.1:8000/John?surname=Lennon&is_staff=1
```

```
http://127.0.0.1:8000/Paul?surname=McCartney&is_staff=yes
```

## Перечисления (Enum) в параметрах

Перечисление (Enum) позволяет создать набор из пар «имя-значение». В Python есть стандартная библиотека, позволяющая использовать перечисления:

```
from enum import Enum

class Fruit(Enum):
    # Синтаксис: имя = значение.
    APPLE = 110
    PEAR = 128
    PLUM = 256

print(Fruit.APPLE)          # Напечатает Fruit.APPLE.
print(Fruit.APPLE.value)    # Напечатает 110.
```

В FastAPI необходимо указывать тип данных для значений Enum. Для path- и query-параметров поддерживаются только два типа данных: строковые и целочисленные. Для этого используется множественное наследование:

```
from enum import Enum
```



```
class MyEnum(str, Enum):  
    ...
```

В Python 3.11 есть возможность использовать уже готовый класс:

```
from enum import StrEnum  
  
class MyEnum(StrEnum):  
    ...
```

Для целочисленных перечислений используется `IntEnum`:

```
from enum import IntEnum  
  
class MyEnum(IntEnum):  
    ...
```

Для применения перечислений в параметрах запросов достаточно указать Enum-класс в качестве аннотации типа для query- или path-параметра:

```
...  
from enum import Enum
```

```
class EducationLevel(str, Enum):  
    SECONDARY = 'Среднее'  
    SPECIAL = 'Специальное'  
    HIGHER = 'Высшее'  
  
@app.get('/{name}')  
def greetings(  
    education_level: EducationLevel,  
    name: str, surname: str, age: int = 10,  
    is_staff: bool = False,  
) -> dict[str, str]:  
    ...
```

В этом коде класс `EducationLevel` определяет допустимые значения для path-параметра `education_level`:

```
http://domain.not/Andrey&education_level=Среднее&surname=Petrov
```

-----

```
# Такой запрос будет принят и обработан, ведь значение education_level  
# соответствует одному из значений в классе EducationLevel.
```

Если же значение параметра `education_level` в запросе отличается от «разрешённых»...

```
domain.not/Andrey&education_level=Ясли&surname=Petrov
-----
# Такой запрос будет отклонён: значение параметра education_level
# не соответствует ни одному из значений в классе EducationLevel.
```

...запрос будет отклонён, вернётся ошибка.

## Документация: порядок параметров функции

Чтобы изменить порядок параметров функции в документации, используется специальный синтаксис параметров функции.

```
# * Синтаксис со звёздочкой доступен с версии Python 3.0.
# / Слеш доступен, начиная с версии Python 3.8.
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           |           |
    |   Позиционные или именованные |
    |                               |
    |   Л Позиционные аргументы   | Л Именованные аргументы
```

- Слеш `/` отделяет позиционные аргументы от аргументов, которые могут быть именованными или позиционными.
- Звездочка `*` указывает, что после нее будут только именованные параметры.

Без этого синтаксиса будет невозможно расположить аргументы, например, в таком порядке:

```
@app.get('/{name}/{surname}')
def greetings(
    name: str,
    surname: str,
    age: int = 10,
    education_level: EducationLevel,
    is_staff: bool = False,
) -> dict[str, str]:
    return {'hello': 'hello'}
```

При запуске приложения возникнет ошибка из-за неправильного положения аргумента `education_level: EducationLevel`:

```
SyntaxError: non-default argument follows default argument
```

Если перед аргументами поставить звездочку `*`, то аргумент `education_level: EducationLevel` можно разместить на любой позиции: все аргументы будут восприниматься как именованные.

```
@app.get('/{name}/{surname}')
def greetings(
    *,
    name: str,
    surname: str,
    age: int = 10,
    education_level: EducationLevel,
```

```
is_staff: bool = False,  
) -> dict[str, str]:  
    return {'hello': 'hello'}
```

## Расширенные возможности документации

FastAPI и Swagger дают возможность сделать документацию более информативной. Для этого используются специальные параметры декораторов view-функций. Значения этих параметров будут отображены в документации.

Для назначения тега маршруту используется параметр `tags`. Можно указать несколько тегов в списке.

```
...  
  
@app.get('/me', tags=['special methods'])  
def hello_author():  
    ...
```

Название маршрута указывается через параметр `summary`:

```
...  
  
@app.get('/me', tags=['special methods'], summary='Приветствие автора')  
def hello_author():  
    ...
```

Описание маршрута указывается через параметр `description`:

```
...

@app.get(
    '/{name}',
    tags=['common methods'],
    summary='Общее приветствие',
    description='Приветствие человека по имени и фамилии; '
                'опционально указывается возраст, образование и статус сотрудника'
)
def greetings(
    ...
```

Описание ответа указывается через параметр `response_description`:

```
...

@app.get(
    '/{name}',
    tags=['common methods'],
    summary='Общее приветствие',
    response_description='Полная строка приветствия'
)
```

```
def greetings(  
    ...
```

**GET** `/ {name}/{surname}` Общее приветствие

**Parameters** Try it out

Name	Description
<b>name</b> * required string (path)	<input type="text" value="name"/>
<b>surname</b> * required string (path)	<input type="text" value="surname"/>
<b>age</b> integer (query)	Default value : 10 <input type="text" value="10"/>
<b>education_level</b> * required string (query)	Available values : Среднее образование, Среднее специальное образование, Высшее образование <input type="text" value="Среднее образование"/>
<b>is_staff</b> boolean (query)	Default value : false <input type="text" value="false"/>

**Responses**

Code	Description	Links
200	Полная строка приветствия	No links

Media type



Валидация значений path-параметров проводится при помощи класса `Path`, валидация query-параметров — при помощи класса `Query`. Эти классы импортируются из модуля `fastapi`.

```
from fastapi import FastAPI, Path, Query

...

@app.get('/{name}')
def greetings(
    # У параметров запроса name и surname значений по умолчанию нет,
    # поэтому в первый параметр ставим многоточие, Ellipsis.
    name: str = Path(..., min_length=2, max_length=20),
    surname: str = Query(..., min_length=2, max_length=50),
    ...
) -> dict[str, str]:
    ...
```

В примере выше указаны валидаторы длины строки: `min_length` указывает минимальный размер строки, `max_length` — максимальный размер.

Подробное описание валидации параметров есть в документации:

- <https://fastapi.tiangolo.com/tutorial/query-params-str-validations/>
- <https://fastapi.tiangolo.com/tutorial/path-params-numeric-validations/>

Для расширения документации параметрам функции можно дать название и описание, передав их в классы `Path` и `Query`.

```
@app.get(
    ...
)
def greetings(
    ...,
    name: str = Path(
        ..., min_length=2, max_length=20,
        title='Полное имя', description='Можно вводить в любом регистре'
    ),
    ...
):
    ...
```

## Использование псевдонимов (alias)

Ключ параметра запроса в URL не обязательно должен соответствовать параметру view-функции. Установить имя ключа в URL можно через параметр `alias`:

```
@app.get(
    ...
)
def greetings(
```

```
...,
is_staff: bool = Query(False, alias='is-staff'),
...
):
    ...
```

Теперь имя параметра `is_staff` в запросе надо указывать через дефис:

```
http://127.0.0.1:8000/Ivan?surname=Petrov&is-staff=true.
```

## Исключение параметров из документации

Параметры функции можно исключать из генерируемой документации. Делается это при помощи параметра `include_in_schema` со значением `False`:

```
@app.get(
    ...
)
def greetings(
    ...,
    is_staff: bool = Query(
        False, alias='is-staff', include_in_schema=False
    ),
    ...
```

```
):  
    ...
```

## Передача списка через query-параметры

В FastAPI есть возможность передавать список в одном параметре:

```
@app.get(  
    ...  
)  
def greetings(  
    ...,  
    surname: list[str] = Query(..., <валидаторы>),  
    ...  
):  
    ...
```

При такой аннотации параметр `surname` может быть указан в GET-запросе несколько раз, а значением `surname` будет список.

```
/Eduardo?surname=Santos&surname=Tavares&surname=Melo&surname=Silva&age=98
```