ChatGPT

Flask и FastAPI: Руководство для продвинутого Python-разработчика

Flask

Flask – это лёгкий веб-фреймворк на Python, позволяющий быстро создавать серверные приложения. Простейший пример приложения возвращает «Hello, World!»:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Привет, мир!</p>"
```

Здесь с помощью декоратора `@app.route("/")` мы связываем URL `/` с функцией-обработчиком. При запуске приложения запросы к корневому пути вернут строку «Привет, мир!» [1].

Маршруты: GET, POST и параметры

По умолчанию маршрут во Flask отвечает только на `GET`-запрос [2]. Чтобы обрабатывать другие методы, указывайте параметр `methods` в декораторе:

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # логика обработки POST-запроса
        return do_the_login()
    else:
        # логика обработки GET-запроса
        return show_login_form()
```

В этом примере один маршрут `/login` обрабатывает и GET, и POST. Для разделения логики можно использовать сочетание декораторов `@app.get` и `@app.post` на разные функции:

```
@app.get('/login')
def login_get():
    return show_login_form()

@app.post('/login')
```

```
def login_post():
    return do_the_login()
```

Flask автоматически добавит поддержку метода `HEAD` при наличии `GET`, а также сгенерирует ответ на `OPTIONS` [3].

Маршруты могут принимать параметры в пути. Например, `<username>` или `<int:post_id>` в URL:

```
@app.route('/user/<username>')
def profile(username):
    return f"Профиль пользователя {username}"

@app.route('/post/<int:post_id>')
def show_post(post_id):
    return f"Пост №{post_id}"
```

В этих функциях части пути в угловых скобках передаются как аргументы функции [4]. Фреймворк автоматически преобразует типы по аннотации (`int:post_id` приведёт параметр к `int`) [4].

Flask с базой данных (SQLite)

Flask не содержит встроенного ORM, но для простых задач можно использовать стандартный модуль `sqlite3`. Рекомендуемый подход – хранить соединение с БД в контексте запроса через `flask.g`. Например:

```
import sqlite3
from flask import Flask, g

app = Flask(__name__)
DATABASE = 'example.db'

def get_db():
    if 'db' not in g:
        # Устанавливаем соединение с SQLite; оно будет сохранено в g
        g.db = sqlite3.connect(DATABASE)
        g.db.row_factory = sqlite3.Row
    return g.db

@app.route('/create_db')
def create_db():
    db = get_db()
    db.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY,
name TEXT)")
    db.commit()
    return "База данных создана."
```

```
@app.route('/add_user/<username>')
def add_user(username):
    db = get_db()
    db.execute("INSERT INTO users (name) VALUES (?)", (username,))
    db.commit()
    return f"Пользователь {username} добавлен."

@app.route('/users')
def list_users():
    db = get_db()
    rows = db.execute("SELECT * FROM users").fetchall()
    # Возвращаем список пользователей
    users = [dict(row) for row in rows]
    return {"users": users}

@app.teardown_appcontext
def close_db(error):
    db = g.pop('db', None)
    if db is not None:
        db.close()
```

В этом примере функция `get_db()` устанавливает соединение при первом обращении и сохраняет его в объекте `g` на время запроса [5]. После окончания запроса (в `teardown_appcontext`) соединение автоматически закрывается [6]. Маршруты `/create_db`, `/add_user/<username>` и `/users` демонстрируют создание таблицы, добавление записи и получение списка из SQLite.

FastAPI

FastAPI – современный асинхронный веб-фреймворк на Python, ориентированный на высокую производительность и удобство разработки. Пример простого приложения FastAPI:

```
from fastapi import FastAPI

app = FastAPI()

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

Здесь используется асинхронная функция-обработчик (с `async def`), возвращающая JSON-ответ. FastAPI автоматически генерирует OpenAPI-схему и интерактивную документацию (Swagger UI и ReDoc).

Pydantic (модели запросов и ответов)

FastAPI широко использует **Pydantic** для моделирования данных и их валидации. Определите класс, унаследованный от `BaseModel`, с полями и типами данных. Например:

```python
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    description: str | None = None

@app.post("/items/")
async def create_item(item: Item):
    # FastAPI валидирует JSON и создает объект Item
    return {"name": item.name, "price_with_tax": item.price * 1.1}
```

В примере FastAPI автоматически преобразует JSON-тело запроса в экземпляр класса `Item`, выполняет проверку типов и обязательных полей [7]. Если данные не соответствуют модели, клиент получит ошибку валидации. Таким образом, `Pydantic` обеспечивает мощную и декларативную валидацию данных для запросов и ответов.

Depends (внедрение зависимостей)

FastAPI имеет встроенную систему *Dependency Injection*. Любую функцию можно использовать как зависимость, поместив её в аргументы обработчика с помощью `Depends`. Например, общие параметры запросов:

```python
from fastapi import Depends

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

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

Здесь функция `common_parameters` возвращает словарь с параметрами. В обработчике она указывается через `Depends`, и FastAPI вызывает её автоматически перед основным кодом. Результат передается в аргумент `common` [8]. Это позволяет избегать дублирования кода и организовывать повторно используемые компоненты.

APIRouter (группировка маршрутов)

Для более крупного приложения маршруты можно разделять на группы с помощью `APIRouter`. Например:

```python
from fastapi import APIRouter

router = APIRouter(prefix="/users", tags=["users"])

@router.get("/")
```

```
async def get_users():
    return [{"username": "Alice"}, {"username": "Bob"}]

app.include_router(router)
```

В этом примере маршруты, определённые в `router`, автоматически получают префикс `/users`. Затем регистрируем маршрутизатор в основном приложении через `app.include_router(router)` [9]. Это даёт гибкость структурировать приложение на модули (каждый `APIRouter` может быть в своём файле) и устанавливать общие зависимости или теги сразу для группы маршрутов.

Middleware (промежуточное ПО)

Middleware – это функции, которые обрабатывают **каждый** запрос до основного обработчика и каждый ответ перед отправкой клиенту. FastAPI (на базе Starlette) позволяет легко добавлять middleware через декоратор:

```
from fastapi import Request

@app.middleware("http")
async def add_process_time(request: Request, call_next):
    # Код до вызова обработчика (например, замер времени)
    response = await call_next(request)
    # Код после вызова обработчика (например, модификация ответа)
    response.headers["X-Process-Time"] = "..."
    return response
```

Как описано в документации, middleware получает запрос, может выполнить произвольный код и передать `request` дальше в приложение, а затем обработать возвращённый `response` [10]. Middleware выполняются в том же порядке, в каком были добавлены. Если в обработчиках есть `BackgroundTasks`, они запускаются **после** завершения всех middleware [11].

BackgroundTasks (фоновые задачи)

Иногда нужно выполнять долгие операции **после** отправки ответа клиенту. FastAPI поддерживает фоновые задачи: достаточно объявить параметр типа `BackgroundTasks` и добавить задачу методом `add_task`:

```
from fastapi import BackgroundTasks

def write_notification(email: str):
    with open("notification.log", "a") as f:
        f.write(f"Notification for {email}\n")

@app.post("/notify/{email}")
async def send_notification(email: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(write_notification, email)
    return {"message": "Уведомление будет отправлено в фоне"}
```

В примере при POST-запросе по `/notify/{email}` FastAPI возвращает ответ сразу, а функция `write_notification` будет выполнена в фоне [12]. Это удобно для рассылок писем, логирования, обработки больших файлов и пр., когда клиенту не нужно ждать завершения задачи.

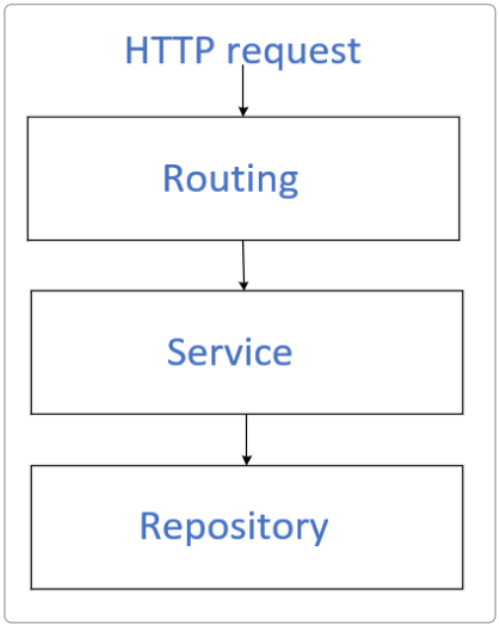

Рисунок: Схема обработки HTTP-запроса в типовом приложении FastAPI. Запрос от клиента поступает в слой маршрутизации (Routing), далее передаётся в бизнес-логику (Service) и, при необходимости, в слой доступа к данным (Repository). В реальном приложении могут быть добавлены Middleware до и после маршрутизатора, а внутри обработчика – запуск BackgroundTasks.

Swagger UI и ReDoc

FastAPI автоматически генерирует документацию API по стандарту OpenAPI. По умолчанию пользовательские интерфейсы для тестирования API доступны по путям `/docs` (Swagger UI) и `/redoc` (ReDoc) [13]. Например, вызов `http://localhost:8000/docs` откроет Swagger UI с интерактивным описанием всех эндпоинтов. Параметры `docs_url` и `redoc_url` при создании `FastAPI()` позволяют изменить эти пути или полностью отключить один из интерфейсов [13].

Обработчик с несколькими HTTP-методами

FastAPI позволяет одному обработчику отвечать на несколько методов HTTP. Для этого используется декоратор `@app.api_route` с параметром `methods`. Например:

```
from fastapi import FastAPI, Request

app = FastAPI()

@app.api_route("/items", methods=["GET", "POST"])
async def items_endpoint(request: Request):
    if request.method == "POST":
```

```
        return {"result": "Обработка POST"}
    return {"result": "Обработка GET"}
```

Здесь маршрут `/items` обрабатывает и GET, и POST запросы. Мы анализируем `request.method` внутри функции. Аналогично Flask, FastAPI автоматически поддерживает HEAD и OPTIONS при наличии GET.

---

1 2 3 4 Quickstart — Flask Documentation (3.1.x)
https://flask.palletsprojects.com/en/stable/quickstart/

5 6 Define and Access the Database — Flask Documentation (3.1.x)
https://flask.palletsprojects.com/en/stable/tutorial/database/

7 Request Body - FastAPI
https://fastapi.tiangolo.com/tutorial/body/

8 Dependencies - FastAPI
https://fastapi.tiangolo.com/tutorial/dependencies/

9 APIRouter class - FastAPI
https://fastapi.tiangolo.com/reference/apirouter/

10 11 Middleware - FastAPI
https://fastapi.tiangolo.com/tutorial/middleware/

12 Background Tasks - FastAPI
https://fastapi.tiangolo.com/tutorial/background-tasks/

13 Metadata and Docs URLs - FastAPI
https://fastapi.tiangolo.com/tutorial/metadata/