FastAPI

Фреймворк для ленивых

FastAPI — самый молодой среди рассматриваемых асинхронных фреймворков. Название говорит само за себя: он создан для быстрой разработки API на Python. В его основе лежат веб-фреймворк Starlette, использующий `asyncio`, и библиотека для валидации данных Pydantic.

Fast API досталась от Starlette классическая функциональность веб-фреймворка: - работа с Websocket, - фоновые задачи, - клиент для тестирования, - централизованная обработка исключений, - встроенная поддержка GraphQL, - поддержка пользовательских сессий и Cookie, - и прочие необходимые способности.

Pydantic отвечает за валидацию API при помощи встроенной в Python аннотации типов. Кроме встроенных типов, Pydantic обзавёлся своими, например, `Color` для цветов в css, а ещё `Json`, `AnyUrl` или `UUID5`.

В результате этого союза получился быстрый (как по скорости работы, так и по написанию кода) фреймворк, со встроенной автоматической валидацией и сериализацией данных на основе описанных моделей. Ещё он умеет авторизировать пользователя через JWT, api-key или OAuth2 и самостоятельно генерировать документацию!

Благодаря строгой типизации во фреймворке есть встроенная генерация OpenAPI-файла. После запуска приложения вы получите готовую документацию для вашего API и интерфейс для её просмотра. К сожалению, в обратную сторону это не работает: по OpenAPI-файлу нельзя получить готовый API, поэтому придётся писать его самостоятельно :(

Ещё одна отличительная способность FastAPI — поддержка внедрения зависимостей. Это делает код более гибким: вы можете переиспользовать нужные наборы параметров в разных частях приложения. А ещё ваши зависимости могут тоже иметь зависимости :)

В качестве примера рассмотрим реализацию API, которое складывает два числа.

Скопировать код

```
from fastapi import FastAPI
from pydantic import BaseModel
from pydantic.fields import Optional, Field

# Объявляем приложение и задаём ему название, которое будет
отображаться в документации
app = FastAPI(title="Простые математические операции")


# Объявляем модель, которая будет валидировать данные, поступающие
от пользователя
# При несовпадении данных со схемой пользователю вернётся ошибка
валидации
class Add(BaseModel):
    first_number: int = Field(title='Первое слагаемое')
    second_number: Optional[int] = Field(title='Второе слагаемое')


# Объявляем модель для формирования результата
# При несовпадении данных со схемой вы получите подробный
traceback :)
class Result(BaseModel):
    result: int = Field(title='Результат')


# Добавляем URL и привязываем к нему модели запроса и ответа
@app.post("/add",  response_model=Result)
async def create_item(item: Add):
    # Выполняем вычисления
    return {
        'result': item.first_number + item.second_number or 1
    }
```

Особенности

- Если вы используете `exception_handler` для оповещения пользователя об ошибках, магия автогенерации документации не сработает: Swagger не будет знать о новых вариантах ответа. Вам придётся дополнить OpenAPI вручную.
- Благодаря системе зависимостей вы можете создать «плагин» для приложения, используя те же структуру и синтаксис, что и

для операций с путями. Но стоит учитывать, что зависимости будут инициализироваться при каждом обращении. Вы не будете пользоваться одним объектом для нескольких URL.

- FasatAPI не ограничивает вас в дизайне архитектуры приложения. Например, вы можете указывать структуру тела запроса в аргументах обработчика или вынести их в отдельную модель.
- Событий, на которые можно повесить сигналы, довольно мало: FastAPI при старте и завершении работы приложения.

Для вашей задачи это самый удачный выбор: фреймворк заточен для написания API, прост в освоении и обладает широким набором возможностей.

В следующих уроках вы познакомитесь с ним ближе!