Приглашаем преподавателей поучаствовать в интервью с нашей командой исследований. Если вы готовы участвовать, пожалуйста, заполните форму. Приглашение на интервью.

Урок FastAPI

FastAPI и Pydantic

1. Введение

Наконец, мы добрались до самой "модной" на сегодня связки библиотек для создания веб-приложений на Python — `FastAPI` и `Pydantic`.

`FastAPI` — это современный, высокопроизводительный веб-фреймворк для создания API на Python 3.6+ на основе стандартных аннотаций типов Python.

`Pydantic` — это библиотека для валидации данных и управления настройками, используемая `FastAPI` для обработки и валидации входящих данных.

Если сравнить с другими асинхронными фреймворками, то `FastAPI` имеет следующие преимущества:

- Производительность: Высокая производительность, сравнимая с `Tornado`
- Автогенерация документации
- Использование аннотаций типов: Полностью использует аннотации типов Python для валидации и автоматической генерации документации
- Простой синтаксис (интуитивнее, чем у других веб-фреймворков)

Начнем с установки, помимо самого `FastApi` нам понадобится еще `Uvicorn` — ASGI (Asynchronous Server Gateway Interface) сервер с помощью pip.

```
pip install fastapi uvicorn
```

2. Основы FastAPI

Создадим простое веб приложение, которое будет принимать запросы на `/` и возвращать ответ `Hello, World!`

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def read_root():
    return {"message": "Hello, World!"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

Синтаксис `FastAPI` очень прост и похож на `Flask`. Создаем экземпляр приложения класса `FastAPI`, отмечаем обработчики декораторами, а затем запускаем наше приложение с помощью сервера `uvicorn`.

Как и многие другие веб-фреймворки `FastAPI` поддерживает пути с параметрами.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello/{name}")
async def read_item(name: str):
    return {"message": f"Hello, {name}!"}
```

А кроме того, позволяет удобно доставать значения параметров.

```
from fastapi import FastAPI

app = FastAPI()
```

```
@app.get("/items/")
async def read_item(name: str, age: int):
    return {"name": name, "age": age}
```

Можно легко обрабатывать значение заголовков

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/headers/")
async def read_headers(user_agent: str = Header(None)):
    return {"User-Agent": user_agent}
```

А также проставлять заголовки в ответном сообщении.

```
from fastapi import FastAPI, Response

app = FastAPI()

@app.get("/response-headers/")
async def custom_response(response: Response):
    response.headers["X-Custom-Header"] = "Custom value"
    return {"message": "Check the response headers"}
```

Загрузка файлов поддерживается с помощью отдельных классов `File` и `UploadFile`.

```
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile = File(...)):
    return {"filename": file.filename}
```

А отправка с помощью класса `responses.FileResponse`

```
from fastapi import FastAPI
from fastapi.responses import FileResponse
```

```
app = FastAPI()

@app.get("/downloadfile/")
async def download_file():
    return FileResponse("example.txt")
```

3. Работа с Pydantic моделями

Как вы уже могли заметить, `FastAPI` любит принимать в свои методы параметры напрямую, а не через какие-то классы запроса (как в некоторых других веб-фреймворках). С одной стороны — это очень удобно и наглядно, а с другой при большом количестве необходимых параметров становится легко запутаться, а сама сигнатура метода становится громоздкой.

Тут нам на помощь приходит `Pydantic`, который используется для валидации и сериализации данных. Рассмотрим простой пример:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

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

@app.post("/items/")
async def create_item(item: Item):
    return item
```

Мы создали модель товара, которую унаследовали от `BaseModel`. Все поля модели являются обязательными, кроме `description` и `tax`, которые имеют значения по умолчанию. При передаче данных в метод `create_item` данные будут автоматически валидированы и преобразованы в объект `Item`.

`Pydantic` предоставляет множество возможностей для настройки полей моделей

```python
from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):
    name: str = Field(..., min_length=3, max_length=50, descriptio
    description: str = Field(None, max_length=300, description="Оп
    price: float = Field(..., gt=0, description="Цена товара")
    tax: float = Field(None, ge=0, description="Налог на товар")

@app.post("/items/")
async def create_item(item: Item):
    return item
```

Мы можем указать значение по умолчанию, ограничение на длину для строковых полей, ограничение на числовые поля и добавить описание для полей.

`Pydantic` поддерживает вложенные модели для описания сложных структур данных.

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

class User(BaseModel):
    username: str
    email: str

class Item(BaseModel):
    name: str
    price: float
    owner: User

@app.post("/items/")
async def create_item(item: Item):
    return item
```

Кроме того, можно использовать стандартные перечисления Python (Enum) при описании моделей.

```python
from fastapi import FastAPI
from pydantic import BaseModel
from enum import Enum

app = FastAPI()

class Status(str, Enum):
    available = "available"
    sold = "sold"

class Item(BaseModel):
    name: str
    price: float
    status: Status

@app.post("/items/")
async def create_item(item: Item):
    return item
```

Помимо встроенной валидации, `Pydantic` позволяет добавлять методы класса для кастомной валидации.

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, validator

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float

    @validator('price')
    def price_must_be_positive(cls, v):
        if v <= 0:
            raise ValueError('Цена должна быть положительной')
        return v

@app.post("/items/")
async def create_item(item: Item):
    return item
```

4. Заключение

FastAPI и Pydantic предоставляют мощные инструменты для создания высокопроизводительных, типизированных и хорошо документированных API.

Мы рассмотрели самые основы создания приложений с помощью FastAPI, уделили немного внимания Pydantic и его возможностям. Теперь у вас есть все необходимые знания для создания собственных API с использованием FastAPI и Pydantic.

А на следующем уроке мы добавим еще асинхронное общение с базами данных, чтобы сделать наши веб-приложения еще круче.