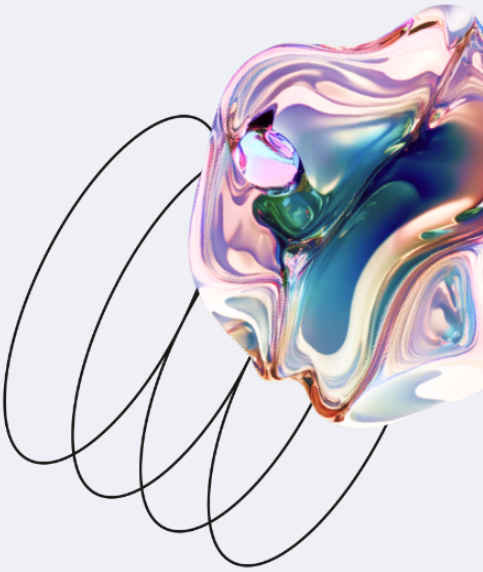
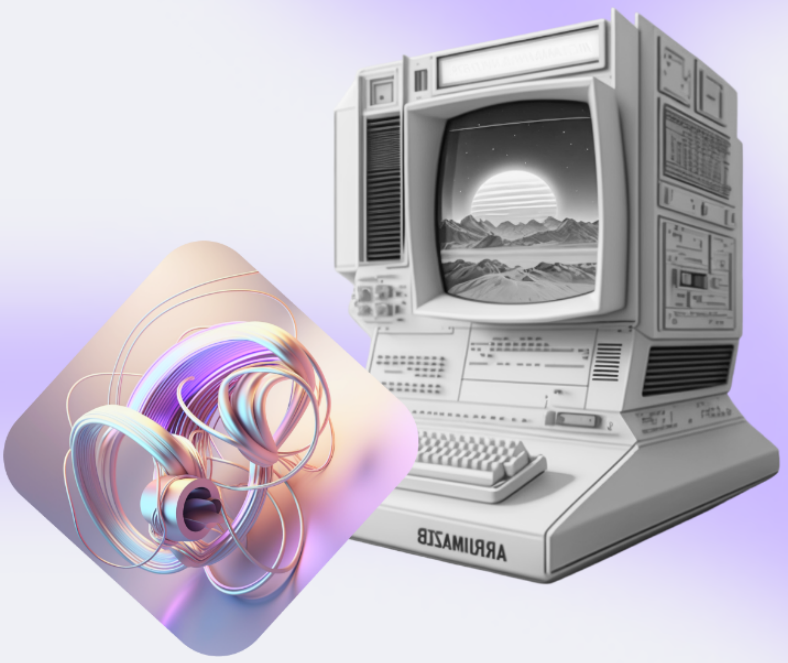
GeekBrains



Лекция 6.

Дополнительные возможности FastAPI



Оглавление

На этой лекции мы

1. Узнаем про модели данных в FastAPI
2. Разберёмся в работе с базой данных
3. Изучим дополнительный возможности проверки данных

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Узнали про FastAPI и его возможности
2. Разобрались в настройке среды разработки
3. Изучили создание базового приложения FastAPI
4. Узнали об обработке HTTP-запросов и ответов
5. Разобрались в создании конечных точек API
6. Изучили автоматическую документацию по API

Подробный текст лекции

Введение в модели данных в FastAPI

FastAPI предоставляет возможность определять модели данных для описания формата запросов и ответов API. Это позволяет автоматически генерировать документацию и проверять данные.

Определение моделей данных

Pydantic — это библиотека для валидации данных и сериализации объектов Python. Она используется в FastAPI для валидации данных, получаемых из запросов, и генерации документации API на основе моделей данных.
Модель данных — это класс Python, определяющий поля и их типы для описания данных. Для определения моделей данных в FastAPI используется класс BaseModel из модуля pydantic. Классы моделей содержат поля, которые описывают структуру данных.

Пример:

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

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None

class User(BaseModel):
    username: str
    full_name: str = None

class Order(BaseModel):
    items: List[Item]
    user: User
```

В этом примере определены три модели данных: Item, User и Order. Каждая модель данных содержит поля с указанными для них типами. Поле is_offer в модели Item имеет значение по умолчанию None.

Работа с полями моделей данных

Поля модели данных могут иметь различные атрибуты, такие как заголовок, описание, значение по умолчанию и другие. Эти атрибуты можно использовать для создания документации и проверки данных.

Функция Field позволяет задавать различные параметры для поля, такие как тип данных, значение по умолчанию, ограничения на значения и т.д. Например, чтобы задать поле типа str с ограничением на длину в 10 символов, можно использовать следующий код:

```
from pydantic import BaseModel, Field

class User(BaseModel):
    name: str = Field(max_length=10)
```

💡 **Внимание!** Field импортируется непосредственно из pydantic, а не из fastapi как для всех остальных (Query, Path и т.д.).

В этом примере мы определили класс User, который содержит поле name, тип которого str. Мы также использовали функцию Field для задания ограничения на максимальную длину строки в 10 символов.

Еще один пример использования функции Field — это задание значения по умолчанию для поля. Например, чтобы задать поле типа int со значением по умолчанию 0, можно использовать следующий код:

```
class User(BaseModel):
    age: int = Field(default=0)
```

В этом примере мы определили класс User, который содержит поле age, тип которого int. Мы также использовали функцию Field для задания значения по умолчанию равного 0.

Рассмотрим ещё один пример:

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

app = FastAPI()


class Item(BaseModel):
    name: str = Field(title="Name", max_length=50)
    price: float = Field(title="Price", gt=0, le=100000)
     description: str = Field(default=None, title="Description",
max_length=1000)
    tax: float = Field(0, title="Tax", ge=0, le=10)


class User(BaseModel):
    username: str = Field(title="Username", max_length=50)
          full_name: str = Field(None, title="Full Name",
max_length=100)
```

В этом примере атрибуты title и description используются для создания документации. Атрибуты min_length и max_length используются для ограничения

длины строковых полей. Атрибуты ge и le используются для ограничения числовых полей.

Перечень принимаемых функцией Field параметров

Для валидации данных можно использовать следующие параметры при создании моделей:

- default: значение по умолчанию для поля
- alias: альтернативное имя для поля (используется при сериализации и десериализации)
- title: заголовок поля для генерации документации API
- description: описание поля для генерации документации API
- gt: ограничение на значение поля (больше указанного значения)
- ge: ограничение на значение поля (больше или равно указанному значению)
- lt: ограничение на значение поля (меньше указанного значения)
- le: ограничение на значение поля (меньше или равно указанному значению)
- multiple_of: ограничение на значение поля (должно быть кратно указанному значению)
- max_length: ограничение на максимальную длину значения поля
- min_length: ограничение на минимальную длину значения поля
- regex: регулярное выражение, которому должно соответствовать значение поля

💡 **Внимание!** Поля не ограничиваются этим списком. Полный перечень можно увидеть в актуальной версии документации к FastAPI

Валидация данных моделей

FastAPI автоматически проверяет данные запроса и ответа, используя модели данных. Если данные не соответствуют определенным типам и ограничениям, будет возбуждено исключение.

Пример:

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

```
app = FastAPI()


class Item(BaseModel):
    name: str = Field(title="Name", max_length=50)
    price: float = Field(title="Price", gt=0, le=100000)
     description: str = Field(default=None, title="Description",
max_length=1000)
    tax: float = Field(0, title="Tax", ge=0, le=10)


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

В этом примере функция create_item ожидает объект Item в качестве параметра запроса. Если данные запроса не соответствуют модели Item, будет возбуждено исключение.

Модели данных в FastAPI позволяют определять формат запросов и ответов API, создавать документацию и обеспечивать проверку данных.

Работа с базой данных

Создание подключения к базе данных

FastAPI предоставляет встроенную поддержку работы с базами данных. Мы обсудим, как создать подключение к базе данных и выполнить миграцию базы данных в FastAPI. FastAPI поддерживает различные базы данных, такие как SQLite, PostgreSQL, MySQL и MongoDB. Выбор базы данных зависит от требований и предпочтений проекта.
Рассмотрим, как создать подключение к базе данных и выполнить миграцию базы данных с использованием SQLAlchemy ORM и базы данных SQLite.
SQLite — это облегченная система управления реляционными базами данных на основе SQL.

Чтобы создать соединение с базой данных, нам нужно определить конфигурацию базы данных и использовать библиотеку ORM (Object-Relational Mapping), такую как Tortoise ORM, SQLAlchemy или Peewee.

> 🔥 **Важно!** Если вы не устанавливали SQLAlchemy и databases раньше, выполните команды:

```
pip install sqlalchemy
pip install databases[aiosqlite]
```

Пример:

```
import databases
import sqlalchemy
from fastapi import FastAPI

DATABASE_URL = "sqlite:///mydatabase.db"

database = databases.Database(DATABASE_URL)

metadata = sqlalchemy.MetaData()

...

engine = sqlalchemy.create_engine(DATABASE_URL)
metadata.create_all(engine)

app = FastAPI()


@app.on_event("startup")
async def startup():
    await database.connect()


@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()
```

В этом примере мы используем SQLAlchemy для создания подключения к базе данных SQLite. Переменная DATABASE_URL определяет строку подключения.

Событие запуска используется для создания схемы базы данных. Событие выключения используется для удаления ядра базы данных.

Подключение к PostgreSQL

Если мы хотим зменить SQLite на PostgreSQL, достоточно заменить данные в константе подключения к БД:

```
DATABASE_URL = "postgresql://user:password@localhost/dbname"
```

Указав тип базы данных, имя пользователя, пароль, хост и название базы данных мы установим с ней соединение.

Создание API операций CRUD

Обсудим, как создать API операций CRUD (создание, чтение, обновление и удаление) с использованием FastAPI и SQLAlchemy ORM.
Операции CRUD — это основные функции, которые используются в любом приложении, управляемом базой данных. Они используются для создания, чтения, обновления и удаления данных из базы данных. В FastAPI с SQLAlchemy ORM мы можем создавать эти операции, используя функции и методы Python.

- CREATE, Создать: добавление новых записей в базу данных.
- READ, Чтение: получение записей из базы данных.
- UPDATE, Обновление: изменение существующих записей в базе данных.
- DELETE, Удалить: удаление записей из базы данных.

Работа с БД в CRUD операциях с SQLAlchemy и databases

Для работы с базой данных в операциях CRUD с SQLAlchemy ORM нам необходимо сначала установить соединение с базой данных. Мы можем использовать любую базу данных по нашему выбору, такую как MySQL, PostgreSQL или SQLite. После того,

как мы установили соединение, мы можем выполнять операции CRUD в базе данных, используя SQLAlchemy ORM.
Например, предположим, что у нас есть база данных SQLite, в которой мы создадим таблицу под названием «пользователи». Мы можем подключиться к базе данных, используя следующий код:

```
import databases
import sqlalchemy
from fastapi import FastAPI
from pydantic import BaseModel

DATABASE_URL = "sqlite:///mydatabase.db"

database = databases.Database(DATABASE_URL)

metadata = sqlalchemy.MetaData()

users = sqlalchemy.Table(
    "users",
    metadata,
                    sqlalchemy.Column("id",     sqlalchemy.Integer,
primary_key=True),
    sqlalchemy.Column("name", sqlalchemy.String(32)),
    sqlalchemy.Column("email", sqlalchemy.String(128)),
)

engine = create_engine(
    DATABASE_URL, connect_args={"check_same_thread": False}
)
metadata.create_all(engine)
```

💡 **Внимание!** По умолчанию SQLite разрешает взаимодействовать с ним только одному потоку, предполагая, что каждый поток будет обрабатывать независимый запрос. Это сделано для предотвращения случайного использования одного и того же соединения для разных вещей (для разных запросов). Но в FastAPI при использовании обычных функций (def) несколько потоков могут взаимодействовать с базой данных для одного и того же запроса, поэтому нам нужно сообщить SQLite, что он должен разрешать это с помощью connect_args={"check_same_thread": False}.

Создание моделей для взаимодействия с таблицей в БД

Создадим две модели данных Pydantic:

```
class UserIn(BaseModel):
    name: str = Field(max_length=32)
    email: str = Field(max_length=128)


class User(BaseModel):
    id: int
    name: str = Field(max_length=32)
    email: str = Field(max_length=128)
```

Первая модель нужна для получения информации о пользователе от клиента. А вторая используется для возврата данных о пользователе из БД клиенту.

Добавление тестовых пользователей в БД

Прежде чем работать над созданием API и проходить всю цепочку CRUD для клиента сгенерируем несколько тестовых пользователей в базе данных.

```
@app.get("/fake_users/{count}")
async def create_note(count: int):
    for i in range(count):
                query = users.insert().values(name=f'user{i}',
email=f'mail{i}@mail.ru')
        await database.execute(query)
    return {'message': f'{count} fake users create'}
```

Принимаем целое число count и создаём в БД указанное число пользователей с именами и почтами. Теперь мы готовы не только разрабатывать CRUD, но и тестировать его.

🔥 **Важно!** Не забудьте перейти по адресу http://127.0.0.1:8000/fake_users/25 чтобы добавить пользователей.

💡 **Внимание!** В реальном проекте подобные функции должны быть отключены перед запускам в продакшен.

Формирование CRUD

Создадим необходимые маршруты для реализации REST API.

➢ Создание пользователя в БД, create

Чтобы создать нового пользователя в таблице «users», мы можем определить функцию следующим образом:

```
@app.post("/users/", response_model=User)
async def create_user(user: UserIn):
                query    =    users.insert().values(name=user.name,
email=user.email)
    last_record_id = await database.execute(query)
    return {**user.dict(), "id": last_record_id}
```

Мы определяем маршрут "/users/" для создания нового пользователя. В параметре функции мы ожидаем объект типа UserIn, который содержит имя и email пользователя. Затем мы создаем SQL-запрос на добавление новой записи в таблицу "users" с указанными данными. Выполняем запрос и возвращаем данные созданного пользователя, включая его ID.

➢ Чтение пользователей из БД, read

Чтобы прочитать всех пользователей из таблицы «users», мы можем определить функцию следующим образом:

```
@app.get("/users/", response_model=List[User])
async def read_users():
    query = users.select()
    return await database.fetch_all(query)
```

Мы определяем маршрут "/users/" для чтения всех пользователей. В функции мы создаем SQL-запрос на выборку всех записей из таблицы "users". Выполняем запрос и возвращаем полученные данные в виде списка объектов типа User.

➢ Чтение одного пользователя из БД, read

Чтобы прочитать одного пользователей из таблицы «users», мы можем определить функцию следующим образом:

```
@app.get("/users/{user_id}", response_model=User)
async def read_user(user_id: int):
    query = users.select().where(users.c.id == user_id)
    return await database.fetch_one(query)
```

Мы определяем маршрут "/users/{user_id}" для чтения одного пользователя по его ID. В параметре функции мы ожидаем передачу ID пользователя. Затем мы создаем SQL-запрос на выборку записи из таблицы "users" с указанным ID. Выполняем запрос и возвращаем полученные данные в виде объекта типа User.

➢ Обновление пользователя в БД, update

Чтобы обновить данные пользователя в таблице «users», мы можем определить функцию следующим образом:

```
@app.put("/users/{user_id}", response_model=User)
async def update_user(user_id: int, new_user: UserIn):
                query    =    users.update().where(users.c.id    ==
user_id).values(**new_user.dict())
    await database.execute(query)
    return {**new_user.dict(), "id": user_id}
```

Мы определяем маршрут "/users/{user_id}" для обновления данных пользователя по его ID. В параметре функции мы ожидаем передачу ID пользователя и объекта типа UserIn, который содержит новые данные пользователя. Затем мы создаем SQL-запрос на обновление записи в таблице "users" с указанным ID и новыми данными. Выполняем запрос и возвращаем обновленные данные пользователя.

➢ Удаление пользователя из БД, delete

Чтобы удалить пользователя из таблицы «users», мы можем определить функцию следующим образом:

```
@app.delete("/users/{user_id}")
async def delete_user(user_id: int):
    query = users.delete().where(users.c.id == user_id)
    await database.execute(query)
    return {'message': 'User deleted'}
```

Мы определяем маршрут "/users/{user_id}" для удаления пользователя по его ID. В параметре функции мы ожидаем передачу ID пользователя. Затем мы создаем SQL-запрос на удаление записи из таблицы "users" с указанным ID. Выполняем запрос и возвращаем сообщение об успешном удалении пользователя.

Тестирование операций CRUD

Чтобы протестировать наш API операций CRUD, мы можем использовать такие инструменты, как Postman или Swagger UI. Мы можем отправлять HTTP-запросы к нашему API и проверять правильность создания, чтения, обновления и удаления данных.
Мы можем использовать интерактивную документацию или curl для проверки этих конечных точек, отправляя HTTP-запросы с соответствующими параметрами. Например, чтобы создать нового пользователя, мы можем отправить запрос POST на конечную точку « /users » с данными пользователя в теле запроса.

```
curl -X 'POST' \
  'http://127.0.0.1:8000/users/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "name": "Alex",
  "email": "my@mail.ru"
}'
```

Затем мы можем убедиться, что пользователь был создан в базе данных, отправив запрос GET на конечную точку « /users » и проверив, что пользователь присутствует в ответе.
Создание API операций CRUD в FastAPI - это простой процесс. Мы можем использовать функции и методы Python для выполнения основных функций создания, чтения, обновления и удаления данных из базы данных. Мы можем протестировать наш API с помощью таких инструментов, как Postman или Swagger UI, чтобы убедиться, что он работает правильно.

Больше про валидацию данных

Ранее мы рассматривали возможность указать тип для переменной, чтобы FastAPI сделал проверку данных по типу. В начале лекции поговорили о модели данных и возможностях pydantic.Field для валидации полей модели. Рассмотрим работу с fastapi.Path и fastapi.Query

Проверка параметра пути через Path

fastapi.Path — это класс, который используется для работы с параметрами пути (path parameters) в URL и проверки данных. Он позволяет определять параметры пути, которые будут передаваться в URL, а также задавать для них ограничения на тип данных и значения.

Пример 1:

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int = Path(..., ge=1), q: str =
None):
    return {"item_id": item_id, "q": q}
```

В этом примере мы создаем маршрут "/items/{item_id}" с параметром пути "item_id". Параметр "item_id" имеет тип int и должен быть больше или равен 1. Мы используем многоточие (...) в качестве значения по умолчанию для параметра "item_id", что означает, что параметр обязателен для передачи в URL. Если параметр не передан или его значение меньше 1, то будет сгенерировано исключение.

Пример 2:

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int = Path(..., title="The ID of the
item"), q: str = None):
    return {"item_id": item_id, "q": q}
```

В этом примере мы создаем маршрут "/items/{item_id}" с параметром пути "item_id". Кроме ограничений на тип данных и значения, мы также задаем для параметра "item_id" заголовок "The ID of the item". Это заголовок будет использоваться при генерации документации API: http://127.0.0.1:8000/redoc.
Примеры демонстрируют использование fastapi.Path для работы с параметрами пути и проверки данных. При использовании Path мы можем определять параметры пути, задавать для них ограничения на тип данных и значения, а также указывать заголовки для документации API.

Проверка параметра запроса через Query

fastapi.Query — это класс, который используется для работы с параметрами запроса и проверки строк. Он позволяет определять параметры запроса, которые будут передаваться в URL, а также задавать для них ограничения на тип данных и значения.

Пример 1:

```
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(q: str = Query(None, min_length=3, max_length=50)):
        results = {"items": [{"item_id": "Spam"}, {"item_id": "Eggs"}]}
    if q:
        results.update({"q": q})
    return results
```

В этом примере мы создаем маршрут "/items/" с параметром запроса "q". Параметр "q" имеет тип str и может быть длиной от 3 до 50 символов. Если параметр "q" не передан в запросе, то ему будет присвоено значение None. Если же параметр "q" передан, то его значение будет добавлено к результатам запроса.

Пример 2:

```
from fastapi import FastAPI, Query

app = FastAPI()
```

```
@app.get("/items/")
async def read_items(q: str = Query(..., min_length=3)):
        results = {"items": [{"item_id": "Spam"}, {"item_id":
"Eggs"}]}
    if q:
        results.update({"q": q})
    return results
```

В этом примере мы создаем маршрут "/items/" с параметром запроса "q". Параметр "q" имеет тип str и должен быть длиной не менее 3 символов. В отличие от первого примера, здесь мы используем многоточие (...) в качестве значения по умолчанию для параметра "q". Это означает, что параметр "q" обязателен для передачи в запросе. Если параметр не передан, то будет сгенерировано исключение.
Примеры демонстрируют использование fastapi.Query для работы с параметрами запроса и проверки строк. При использовании Query мы можем определять параметры запроса, задавать для них ограничения на тип данных и значения, а также указывать значения по умолчанию.

Общая основа в виде Param

На самом деле Query, Path, Field и другие фильтры создают объекты подклассов общего класса Param, который сам является подклассом класса FieldInfo из модуля Pydantic. Все они возвращают объекты подкласса FieldInfo.

> 🔥 **Внимание!** Помните, что когда вы импортируете Query, Path, и другие из fastapi, это на самом деле функции, которые возвращают специальные классы.

Как результат Field работает так же Query, как Path, имеет все те же параметры и т. д. Важно лишь выбрать нужную в текущей реализации функцию

Вывод

На этой лекции мы:

1. Узнали про модели данных в FastAPI
2. Разобрались в работе с базой данных
3. Изучили дополнительный возможности проверки данных

Домашнее задание

Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.