

СПРИНТ 6
Разработка API на Python
Часть 1

Назначение API	2
Технологии и инструменты разработки на Python	2
Особенности фреймворка FastAPI	3
Начало разработки. Структура проекта.....	3
Конфигурирование приложения	6
Подключение базы данных	7
Создание сервиса. Внедрение зависимостей	10
Получение, добавление, обновление и удаление объектов из БД.....	11
Создание эндпоинтов. Документирование. Исключения	15

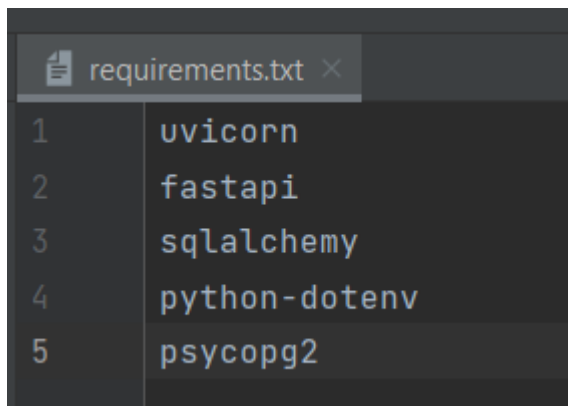
Назначение API

Прикладные программные интерфейсы (Applied Programming Interfaces – API) являются точками доступа клиентов к серверу в приложениях, использующих клиент-серверную архитектуру. Мы будем рассматривать как API как серверное web-приложение, предоставляющее клиентам набор конечных точек доступа (эндпоинтов – endpoints), к которым клиенты могут обращаться по протоколу HTTP (выполняя запросы GET, POST, UPDATE, DELETE и при необходимости другие).

В первой части спринта мы создадим собственный API на Python, а во второй части доработаем его. В ходе работы рассмотрим большинство нюансов, которые возникают в процессе Backend-разработки.

Технологии и инструменты разработки на Python

Для создания и запуска нам потребуется виртуальное окружение, в котором должны быть установлены следующие зависимости:



```
requirements.txt
1 uvicorn
2 fastapi
3 sqlalchemy
4 python-dotenv
5 psycopg2
```

Сразу разберемся, для чего нужна каждая из зависимостей:

- uvicorn – это ASGI-сервер, который необходим для запуска приложений FastAPI;
- fastapi – фреймворк для разработки API на Python, который мы будем использовать;
- sqlalchemy – ORM-инструмент, который мы будем использовать для взаимодействия с БД;
- python-dotenv – инструмент, упрощающий создание конфигураций приложения;

- `psycopg2` – драйвер `sqlalchemy` для СУБД PostgreSQL (которую мы будем использовать, поскольку с ней вы работали в прошлом спринте).

Особенности фреймворка FastAPI

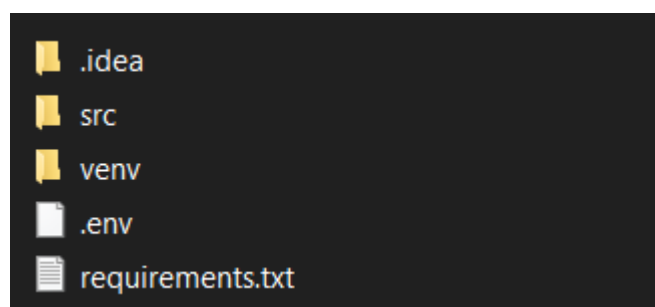
Выделим основные особенности используемого фреймворка, который делают его очень востребованным в сфере Backend-разработки на Python:

- основан на фреймворке `starlette` и библиотеке `pydantic`; из первого в первую очередь следует возможность написания полностью асинхронного кода (которую мы не будем использовать в этом курсе, поскольку пишем только синхронный); из второго следует валидация моделей данных в запросах из коробки и возможность валидации ответов;
- имеет собственную реализацию IoC-контейнера и возможность внедрения зависимостей (Dependency Injection);
- автоматическая генерация документации по стандарту OpenAPI;
- имеет отличную документацию и открытый исходный код.

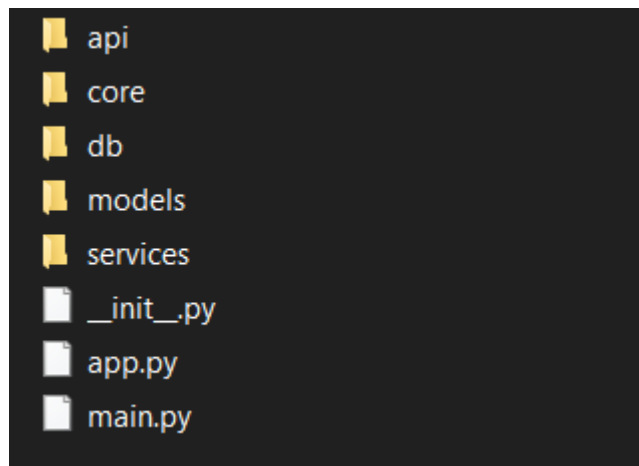
Ну и следуя своему названию, он работает быстро!

Начало разработки. Структура проекта.

Для начала организуем проект. На верхнем уровне расположим папку `src` для хранения файлов исходного кода, а также файл со списком зависимостей (`requirements`) и файл с информацией о переменных окружения (`.env`).



Внутри папки `src` организуем структуру пакетов (директорий) в соответствии с тем, как это описывалось во 2 спринте.



Итак, создадим и запустим первый API. В файле `app` будем создавать объект приложения и работать с ним.

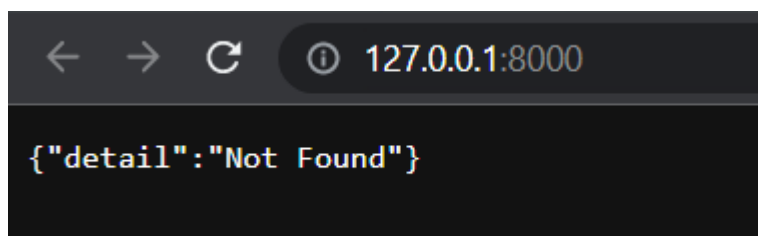
```
app.py x
1  from fastapi import FastAPI
2
3
4  app = FastAPI(
5      title='Мое первое приложение FastAPI',
6      description='Это мое первое приложение Fast API!',
7      version='0.0.1',
8  )
9
```

В файле `main` сделаем точку входа в программу и будем запускать объект `app` из `app` с помощью сервера `uvicorn`.

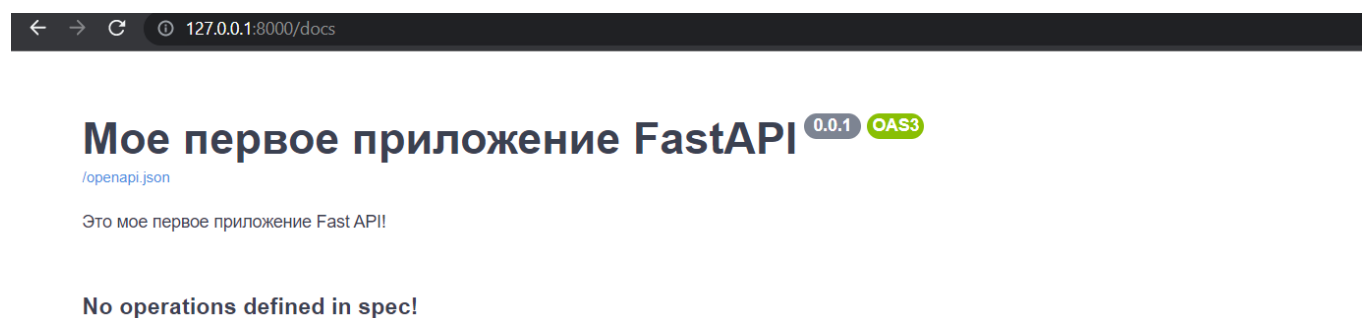
```
main.py x
1  import uvicorn
2
3  if __name__ == '__main__':
4      uvicorn.run('app:app', reload=True)
```

При использовании параметра `reload=True`, `uvicorn` будет отслеживать изменения в файлах и автоматически перезапускать приложение в случае изменений (Hot Reload). Запустим файл `main` и через браузер откроем страницу.

Обратите внимание: если вы запускаете через PyCharm, то скорее всего на данном этапе у вас не будет ошибки. Но если вы запускаете через терминал, то скорее всего вы получите ошибку с текстом, что не найден модуль `src`. Классическим способом для решения подобных проблем является добавление пути к проекту в переменную среды `PYTHONPATH`. Проверьте переменные среды, если там нет `PYTHONPATH`, то создайте такую переменную и в качестве значения установите путь до папки с проектом (папкой с проектом считаем ту, в которой лежит файл `requirements.txt`). В этом случае все вложенные пакеты (в частности `src`) будут нормально импортироваться. Кстати, PyCharm автоматически ставит чекбокс, что путь к проекту будет добавлен в `PYTHONPATH`, поэтому там этой проблемы можно не заметить. Когда все проблемы решились – наблюдаем в браузере следующее:



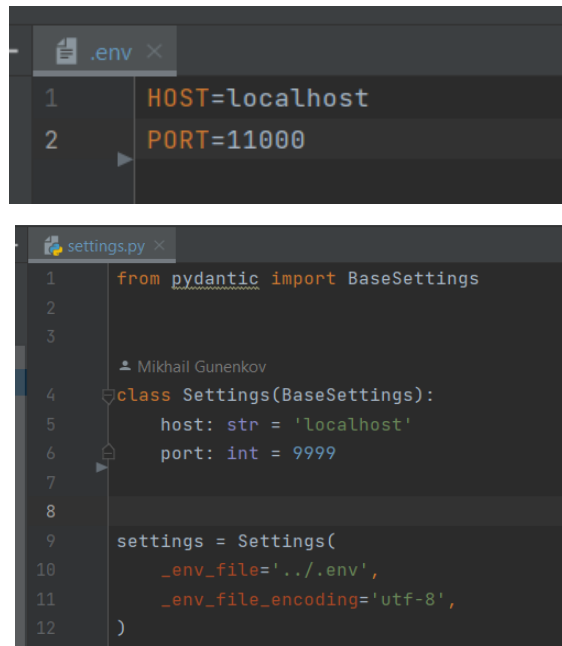
Все хорошо! По умолчанию API располагается по адресу `127.0.0.1` (localhost) на порту `8000`. Теперь посмотрим, как работает автоматическая генерация документации (особо документировать пока нечего, но тем не менее). Перейдем на роут `/docs`.



Здесь будет располагаться Swagger UI для нашего API. Это интерактивная документация для разработчиков клиентов, которые будут пользоваться нашим приложением. Вы можете увидеть, что заголовок, описание и версию мы сами сконфигурировали в файле `app`.

Конфигурирование приложения

Решим первую задачу возможность запуска приложения на с кастомным хостом и кастомным портом. Сделаем это с использованием BaseSettings из pydantic.



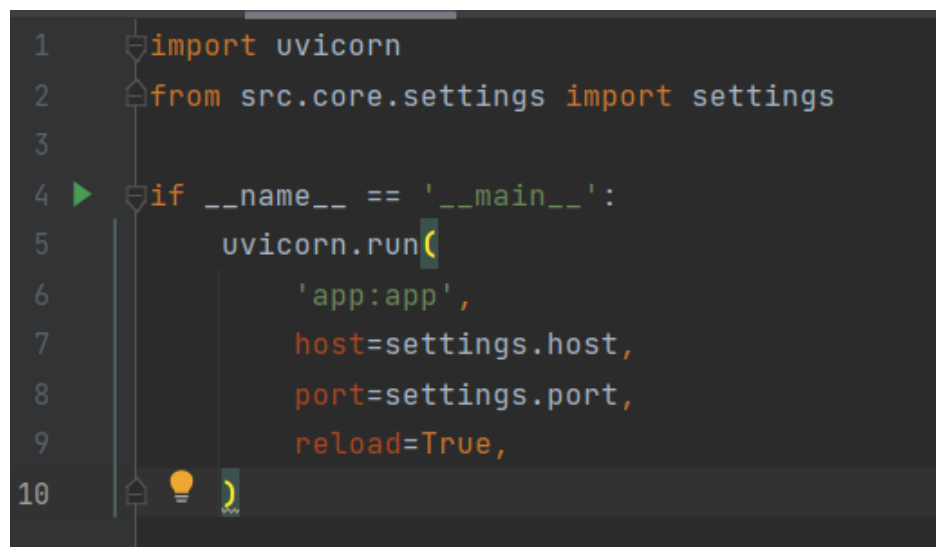
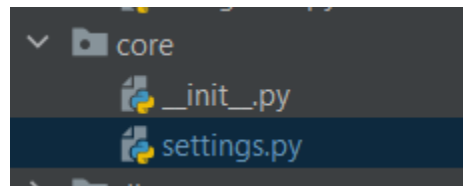
The screenshot shows two files in a code editor. The top file is `.env` with the following content:

```
1 HOST=localhost
2 PORT=11000
```

The bottom file is `settings.py` with the following content:

```
1 from pydantic import BaseSettings
2
3
4 class Settings(BaseSettings):
5     host: str = 'localhost'
6     port: int = 9999
7
8
9 settings = Settings(
10     _env_file='../.env',
11     _env_file_encoding='utf-8',
12 )
```

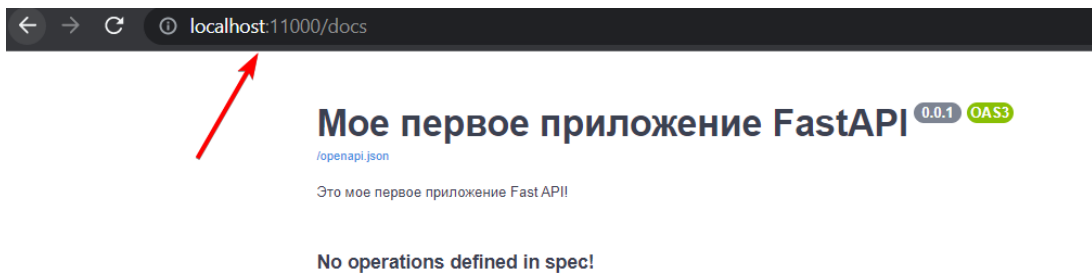
Файл settings расположим в пакете core



The screenshot shows a code editor with the following Python code:

```
1 import uvicorn
2 from src.core.settings import settings
3
4 if __name__ == '__main__':
5     uvicorn.run(
6         'app:app',
7         host=settings.host,
8         port=settings.port,
9         reload=True,
10     )
```

Запустим API еще раз и увидим.

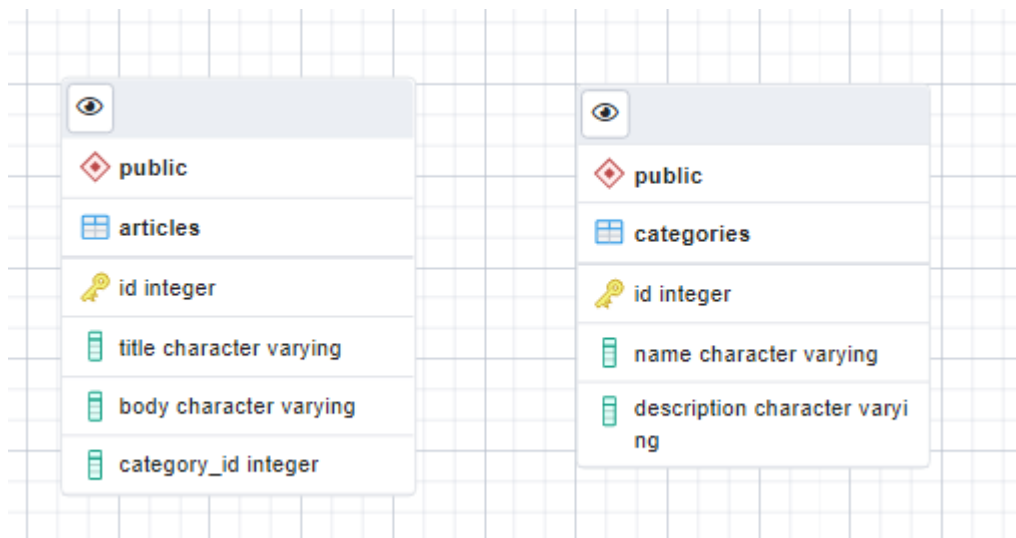


Подключение базы данных

Любое, более-менее серьезное, API работает с базой данных. Чаще всего для взаимодействия с базами используются ORM-инструменты. Мы будем использовать sqlalchemy для взаимодействия с реляционной базой данных PostgreSQL. Для того, чтобы связать наше приложение с базой нам потребуется:

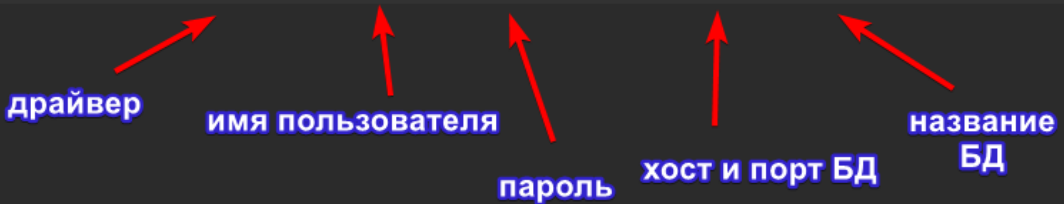
- создать базу данных;
- написать метод, который будет открывать сессию подключения к БД;
- написать классы, которые будут представлять собой модели таблиц базы данных;
- внедрить сессию в сервисы, посредством которых мы будем описывать логику работы с данными.

Пойдем по порядку. Пусть у нас есть небольшая база данных с двумя таблицами. Это статьи и категории статей. У статей есть поле `category_id` (внешний ключ).



В файле с информацией об окружении нам потребуется строка подключения к базе данных.

```
.env x
1 HOST=localhost
2 PORT=11000
3 CONNECTION_STRING=postgresql://postgres:postgres@localhost:5433/ascourse_1
```



Не забудем также добавить новое поле в настройки pydantic:

```
class Settings(BaseSettings):
    host: str = 'localhost'
    port: int = 9999
    connection_string: str
```

В пакете db создадим файл db. Используя sqlalchemy создадим движок (Engine) и класс, с помощью которого можно будет получать объект сессии:

```
1 from sqlalchemy import create_engine
2 from sqlalchemy.orm import sessionmaker
3 from src.core.settings import settings
4
5 engine = create_engine(
6     settings.connection_string
7 )
8
9 Session = sessionmaker(
10     engine,
11     autocommit=False,
12     autoflush=False,
13 )
```

Когда параметры autocommit и autoflush имеют значение False, мы сами выбираем момент, когда транзакция должна быть выполнена. То есть мы можем добавить объект, но не сохранять изменения в БД. А сохранить их тогда, когда добавим еще один объект. Про транзакции либо уже рассказали, либо расскажем на спринте по БД.

В этом же файле создадим функцию, которая будет предоставлять нам объект сессии:

```
def get_session():
    session = Session()
    try:
        yield session
    finally:
        session.close()
```

За счет использования `yield` функция не заканчивается после предоставления объекта `session`, а ожидает окончания работы с ним, после чего выполняется код в `finally`. Такую функцию мы можем использовать для внедрения зависимости, если эта зависимость должна создаваться только в момент необходимости в ней, а после окончания работы с ней она должна закрыться. При каждом запросе к БД у нас будет открываться новая сессия, после выполнения запроса она будет закрываться. Это стандартная практика при работе с БД.

Теперь нам нужно описать классы, который будут отражать таблицы в БД. Создадим для начала в папке `models` файл `base` и в нем создадим базовый класс для всех таких таблиц (более подробно о смысле этого действия будет рассказано во 2 части спринта, в разговоре о создании миграций):

```
base.py x
1  from sqlalchemy.orm import declarative_base
2
3  Base = declarative_base()
4
```

Теперь в той же папке `models` создадим файлы `category` и `article`. Опишем соответствующие классы:

```

category.py x
1  from sqlalchemy import Column, Integer, String
2  from src.models.base import Base
3
4
5  Mikhail Gunenkov
6  class Category(Base):
7      __tablename__ = 'categories'
8      id = Column(Integer, primary_key=True)
9      name = Column(String)
10     description = Column(String, nullable=True)

```

```

article.py x
2  from sqlalchemy.orm import relationship
3  from src.models.base import Base
4
5
6  Mikhail Gunenkov
7  class Article(Base):
8      __tablename__ = 'articles'
9      id = Column(Integer, primary_key=True)
10     title = Column(String)
11     body = Column(String)
12     category_id = Column(Integer, ForeignKey('categories.id'), index=True)
13     category = relationship('Category', backref='categories')

```

Как видно, sqlalchemy позволяет нам полностью смоделировать наши таблицы в виде классов. Этот прием позволит выполнять процедуры с данными без использования SQL (основная идея ORM в этом и заключается).

Создание сервиса. Внедрение зависимостей

Для работы с БД мы введем в приложение специальные объекты – сервисы. Они будут содержать методы, позволяющие получать /добавлять / обновлять / удалять объекты. Сервисы мы будем использовать при выполнении клиентских запросов к API.

В пакете services создадим файл categories, в котором реализуем сервис для работы с категориями.

```

Mikhail Gunenkov
class CategoriesService:
    Mikhail Gunenkov
    def __init__(self, session: Session = Depends(get_session)):
        self.session = session

```

И здесь мы сразу же встречаемся с внедрением зависимостей! Как было сказано ранее, сервис будет работать с БД. Сам сервис нам постоянно так же в памяти не нужен, мы будем создавать его только когда клиент сделает соответствующий запрос.

При этом в процессе создания (инстанцирования) сервиса мы должны получить объект сессии (инстанцировать сессию). В FastAPI существует функция `Depends`, которая позволяет резолвить (resolve) зависимости (напомню, что фреймворк включает в себя IoC-контейнер).

В данном случае, при создании объекта сервиса выполнится функция `get_session`, и мы в конструкторе получим объект сессии, который будем далее использовать для выполнения действий с БД.

Получение, добавление, обновление и удаление объектов из БД

Разберемся теперь, что мы можем делать внутри сервиса с полученной ранее сессией. В первую очередь, мы можем получить несколько записей с сортировкой и фильтрацией при желании.

```

def all(self) -> List[Category]:
    categories = (
        self.session
        .query(Category)
        .order_by(
            Category.id.desc()
        )
        .all()
    )
    return categories

```

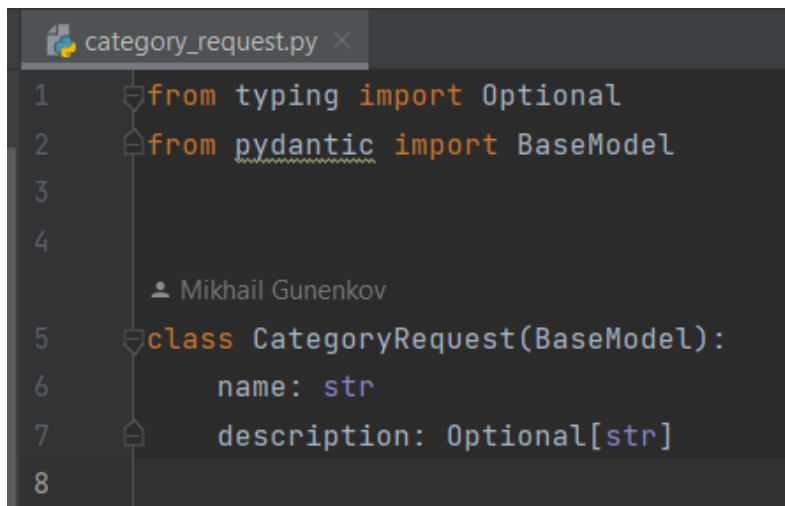
Можем получить только одну запись, по конкретному значению поля:

```
def get(self, category_id: int) -> Category:
    category = (
        self.session
        .query(Category)
        .filter(
            Category.id == category_id,
        )
        .first()
    )
    return category
```

За счет того, что мы описали таблицы в виде классов-моделей, мы можем использовать соответствующие поля в условиях фильтрации и в правилах сортировки. Но особенно сильно этот прием упрощает нам добавление, обновление и удаление данных. Напишем метод для добавления новой категории.

Но перед тем, как его писать, должен возникнуть справедливый вопрос. У нас есть модель данных, соответствующая строке таблицы. Но когда мы хотим создать новую запись в таблице, нам нужно принять какие-то данные от клиента, которые мы поле сможем преобразовать в объект `Category`. Более того, мы позже это увидим, когда мы получили записи из таблицы, мы должны вернуть их клиенту. И, в общем случае, клиенту не нужны все поля, которые есть в таблице (особенно актуально будет, когда хэши паролей пользователей будем хранить). Поэтому в Backend-разработке принято создавать дополнительные модели данных, которые используются при передаче данных между клиентом и сервером. Их называют объекты передачи данных – Data Transfer Objects (DTO). Мы будем также называть их схемами (schemas).

Создадим в пакете `models` пакет `schemas`. В нем создадим еще пакет `category`. Уже в нем создадим два файла – `category_request` и `category_response`. В `category_request` опишем структуру данных о категории с помощью `pydantic`, которую мы получаем от клиента (например, когда хотим добавить или обновить категорию). Обратите внимание в первую очередь на то, что здесь нет идентификатора.

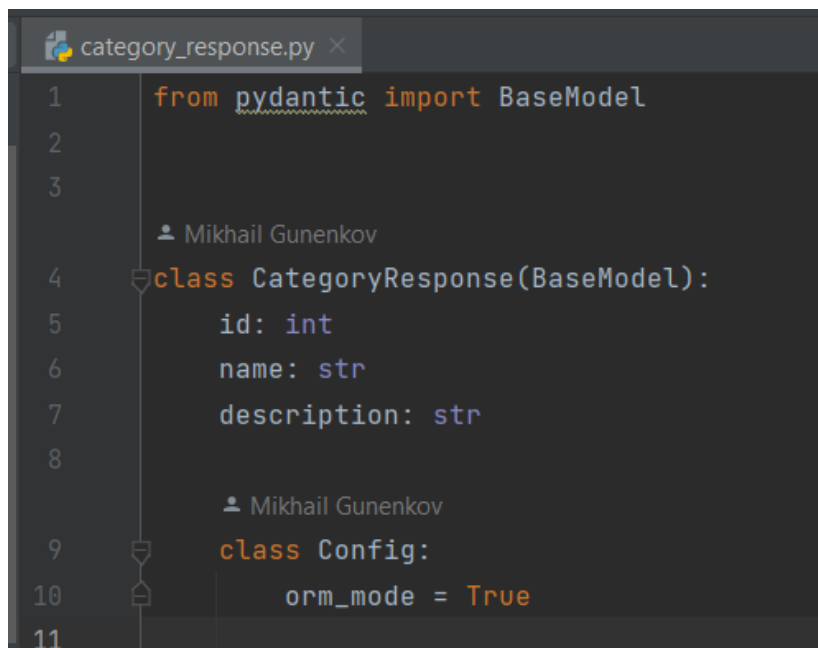


```

1  from typing import Optional
2  from pydantic import BaseModel
3
4
5  class CategoryRequest(BaseModel):
6      name: str
7      description: Optional[str]
8

```

В файле category_response опишем структуру данных о категории, которую мы отдаем клиенту.



```

1  from pydantic import BaseModel
2
3
4  class CategoryResponse(BaseModel):
5      id: int
6      name: str
7      description: str
8
9  class Config:
10     orm_mode = True
11

```

Для корректной работы pydantic с sqlalchemy при преобразовании типов следует добавить дополнительный флаг orm_mode=True у структур данных, которые планируется отдавать клиенту.

Теперь мы можем доработать наш сервис, добавив методы для добавления и обновления категории.

```

35 Mikhail Gunenkov
36 def add(self, category_schema: CategoryRequest) -> Category:
37     category = Category(**category_schema.dict())
38     self.session.add(category)
39     self.session.commit()
40     return category

41 Mikhail Gunenkov
42 def update(self, category_id: int, category_schema: CategoryRequest) -> Category:
43     category = self.get(category_id)
44     for field, value in category_schema:
45         setattr(category, field, value)
46     self.session.commit()
47     return category

```

Выполнение `session.commit()` непосредственно сохраняет данные в БД. Без выполнения коммита, данные в БД не изменятся. Мы указывали `autocommit` и `autoflush` как `False`! При обновлении информации о категории мы пользуемся тем, что объекты в `pydantic` итерируемы и мы можем получить доступ к парам <ключ, значение>. Также видно, что специализированного метода для обновления объекта в ORM нет. ORM следит за изменениями полученного из БД объекта и автоматически обновляет значения полей при выполнении коммита.

Не забудем и о удалении объектов.

```

Mikhail Gunenkov
def delete(self, category_id: int):
    category = self.get(category_id)
    self.session.delete(category)
    self.session.commit()

```

Ну здесь прямо совсем просто! 😊

Создание эндпоинтов. Документирование. Исключения

Теперь, собственно, разработаем главную часть API – энпоинты, к которым клиент сможет обращаться для выполнения действий. В FastAPI принято эндпоинты разбивать по роутерам. Один роутер работает с одной сущностью (одной моделью БД) либо решает один класс задач (принцип единственной ответственности).

В пакете `api` создадим файл `base_router` – базовый роутер.

```
1  from fastapi import APIRouter
2
3  router = APIRouter()
4
```

Теперь мы сразу включим базовый роутер в наше приложение. Перейдем в файл `app`.

```
1  from fastapi import FastAPI
2
3  from src.api.base_router import router
4
5
6  app = FastAPI(
7      title='Мое первое приложение FastAPI',
8      description='Это мое первое приложение Fast API!',
9      version='0.0.1'
10 )
11
12 app.include_router(router)
13
```

С помощью `include_router` мы подключаем базовый роутер. И с помощью этого же метода мы будем к базовому роутеру подключать наши новые роутеры. Создадим роутер для категорий (в пакете `api` файл `categories`). И включим его в базовый роутер.

```
base_router.py x categories.py x
1  from fastapi import APIRouter
2
3  router = APIRouter(
4      prefix='/categories',
5      tags=['categories'],
6  )
7
```

```
base_router.py categories.py x
1  from fastapi import APIRouter
2
3  from src.api import categories
4
5  router = APIRouter()
6  router.include_router(categories.router)
7
```

Создавая роутер, мы указываем префикс у его эндпоинтов (в данном случае, все эндпоинты для работы с категориями будут начинаться с /categories). Также мы указываем список тегов. Это из части документирования. Интерфейс, который мы видим при переходе по роуту /docs называется SwaggerUI. Когда мы создадим эндпоинты, он будет их показывать в соответствии с тегами роутера. То есть роутеру необходимо задать хотя бы один тег. Если задать несколько, то его эндпоинты будут отображаться в нескольких разделах сваггера. Обычно, я задаю один тег одному эндпоинту. Далее можно будет добавить тегам описание, чтобы улучшить интерактивную документацию. Но пока давайте создадим эндпоинты. Начнем с получения списка всех категорий.

```
Mikhail Gunenkov
12  @router.get('/all', response_model=List[CategoryResponse], name="Получить все категории")
13  def get(categories_service: CategoriesService = Depends()):
14      """
15      Получить все категории. Более подробное описание.
16      """
17      return categories_service.all()
18
```

Эндпоинты создаются с помощью декораторов, в которых мы указываем метод (в данном случае GET-запрос), URL эндпоинта (учитывая префикс роутера, чтобы получить список всех категорий нужно будет обратиться по URL <http://localhost:11000/categories/all>), модель ответа (в данном случае – список категорий; мы предварительно создали эту схему данных) и имя эндпоинта (оно отображается в SwaggerUI). Посмотрим, как изменился SwaggerUI.

Мое первое приложение FastAPI 0.0.1 OAS3

OpenAPI JSON

Это мое первое приложение Fast API

categories

GET /categories/all Получить Все Категории

Schemas

CategoryResponse >

Видим, что SwaggerUI отображает подробную информацию о нашем эндпоинте. Мы можем попробовать выполнить запрос.

The screenshot shows the SwaggerUI interface for the 'categories' endpoint. The 'GET /categories/all' method is selected. The 'Parameters' section is empty. The 'Execute' button is highlighted. The 'Responses' section shows a 200 status code. The 'Curl' section displays the command: `curl -X 'GET' \ 'http://localhost:11000/categories/all' \ -H 'accept: application/json'`. The 'Request URL' is `http://localhost:11000/categories/all`. The 'Server response' section shows a 200 status code and a JSON response body:

```
{
  "id": 4,
  "name": "string",
  "description": "string"
},
{
  "id": 3,
  "name": "string",
  "description": "string"
},
{
  "id": 2,
  "name": "string",
  "description": "string"
}
```

Мы получили 3 записи о категориях в БД, которые я предварительно добавил. На данном этапе, если вы не добавляли записей и получили пустой список – значит все сделано правильно. Если есть ошибки – они допущены на этапе подключения БД.

Обратите внимание, что и здесь мы используем внедрение зависимостей. Мы внедряем объект нашего сервиса. Ранее мы использовали функцию, которая возвращает объект. А теперь мы просто говорим объект какого класса хотим получить и IoC-контейнер резолвит нам объект, выполняя конструктор. А внутри конструктора

соответственно, как мы помним, резолвится сессия. В итоге, мы используем в роутере сервис, и он работает с БД.

Теперь напишем запрос для получения одной категории.

```
Mikhail Gunenkov
@router.get('/get/{category_id}', response_model=CategoryResponse, name="Получить одну категорию")
def get(category_id: int, categories_service: CategoriesService = Depends()):
    return get_with_check(category_id, categories_service)

Mikhail Gunenkov
def get_with_check(category_id: int, categories_service: CategoriesService):
    result = categories_service.get(category_id)
    if not result:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Категория не найдена")
    return result
```

Здесь нам необходимо обработать ситуацию, если в БД нет записи с таким id. В этом случае, принято возвращать клиенту ошибку 404. Возвращение клиенту ошибок при исключениях – важный момент. Мы всегда можем использовать HTTPException, где укажем статус-код ошибки и детальную информацию об исключении (детальная информация в общем случае – словарь, строку сделал для упрощения). Импортируется все это из fastapi.

```
1 from typing import List
2 from fastapi import APIRouter, Depends, HTTPException, status
3 from src.models.schemas.category.category_response import CategoryResponse
4 from src.services.categories import CategoriesService
5
```

Теперь в сваггере попробуем получить существующую запись – все хорошо будет.

GET /categories/get/{category_id} Получить Одну Категорию

Parameters Cancel

Name	Description
category_id * required	
integer (path)	4

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:11000/categories/get/4' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:11000/categories/get/4
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 4, "name": "string", "description": "string" }</pre> <p>Download</p>

Response headers

А если попросим несуществующую – будет ошибка.

GET /categories/get/{category_id} Получить Одну Категорию

Parameters Cancel

Name	Description
category_id * required	
integer (path)	6

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:11000/categories/get/6' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:11000/categories/get/6
```

Server response

Code	Details
404 Undocumented	<p>Error: Not Found</p> <p>Response body</p> <pre>{ "detail": "Категория не найдена" }</pre> <p>Download</p>

Наконец, добавим эндпоинты для добавления, обновления и удаления категорий.

```

Mikhail Gunenkov
@router.post('/', response_model=CategoryResponse, status_code=status.HTTP_201_CREATED, name="Добавить категорию")
def add(category_schema: CategoryRequest, categories_service: CategoriesService = Depends()):
    return categories_service.add(category_schema)

Mikhail Gunenkov
@router.put('/{category_id}', response_model=CategoryResponse, name="Обновить информацию о категории")
def put(category_id: int, category_schema: CategoryRequest, categories_service: CategoriesService = Depends()):
    get_with_check(category_id, categories_service)
    return categories_service.update(category_id, category_schema)

Mikhail Gunenkov
@router.delete('/{category_id}', status_code=status.HTTP_204_NO_CONTENT, name="Удалить категорию")
def delete(category_id: int, categories_service: CategoriesService = Depends()):
    get_with_check(category_id, categories_service)
    return categories_service.delete(category_id)

```

Мы также можем указать в декораторе ожидаемый `status_code` (когда добавляется новый объект, принято возвращать код 201, когда ничего не приходит, но все хорошо – статус 204, когда просто все хорошо в общем случае – статус 200).

Теперь сваггер выглядит следующим образом.

Мое первое приложение FastAPI 0.0.1 OAS3

[/openapi.json](#)

Это мое первое приложение FastAPI!

categories		^
GET	/categories/all	Получить Все Категории
GET	/categories/get/{category_id}	Получить Одну Категорию
POST	/categories/	Добавить Категорию
PUT	/categories/{category_id}	Обновить Информацию О Категории
DELETE	/categories/{category_id}	Удалить Категорию

В файле `app` мы можем дополнительно улучшить документацию, добавив для каждого тега описание.

```

5 tags_dict = [
6     {
7         'name': 'categories',
8         'description': 'Категории статей',
9     },
10    {
11        'name': 'articles',
12        'description': 'Статьи (Без реализации)',
13    }
14 ]
15
16
17 app = FastAPI(
18     title='Мое первое приложение FastAPI',
19     description='Это мое первое приложение Fast API!',
20     version='0.0.1',
21     openapi_tags=tags_dict
22 )

```

Посмотрим, что изменилось.

Мое первое приложение FastAPI 0.0.1 OAS3

/openapi.json

Это мое первое приложение Fast API!

categories Категории статей

- GET** /categories/all Получить Все Категории
- GET** /categories/get/{category_id} Получить Одну Категорию
- POST** /categories/ Добавить Категорию
- PUT** /categories/{category_id} Обновить Информацию О Категории
- DELETE** /categories/{category_id} Удалить Категорию

articles Статьи (Без реализации)

Таким образом, в этой части спринта мы познакомились с фреймворком FastAPI, поработали с ORM SQLAlchemy, вспомнили про конфигурирование приложений.

Научились запускать приложения, создавать интерактивную документацию с помощью SwaggerUI.

Попробовали внедрение зависимостей в FastAPI, научились декомпозировать проект на модели, сервисы и роутеры.

В следующей части спринта мы научимся создавать и выполнять миграции в базу данных (подход CodeFirst), научимся использовать авторизацию в ынашем API (с помощью JWT) и посмотрим, как можно загружать и скачивать файлы.