Практическая работа №8. Подключение к нереляционной базе данных

В предыдущих работах были рассмотрены темы структуризации приложения FastAPI, реализации некоторых маршрутов и моделей приложения, а также протестированы конечные точки.

Однако приложение по-прежнему использует внутреннюю базу данных для хранения событий. Необходимо модифицировать приложение для использованияправильной базы данных.

Базу данных можно просто назвать хранилищем данных. В этом контексте база данных позволяет нам хранить данные постоянно, в отличие от встроенной в приложение базы данных, которая стирается при любом перезапуске или сбое приложения. **База данных — это таблица, содержащая столбцы, называемые полями, и строки, называемые записями.**

Цель работы - овладеть умениями и навыками подключения приложениеFastAPI к базе данных. Рассматривается подключение к базе данных **MongoDB** с помощью **Beanie**.

Решаемые задачи:

- Настройка MongoDB
- CRUD операции в MongoDB с помощью Beanie

1. Настройка MongoDB

Существует ряд библиотек, которые позволяют нам интегрировать MongoDB в нашеприложение FastAPI. Однако мы будем использовать **Beanie**, асинхронную библиотеку **Object Document Mapper (ODM)**, для выполнения операций с базой данных из нашего приложения.

Давайте установим библиотеку `beanie`, выполнив следующую команду:

```
(venv)$ pip install beanie
```

Прежде чем погрузиться в интеграцию, давайте рассмотрим некоторые методы избиблиотеки Beanie, а также то, как создаются таблицы базы данных в этом разделе.

1.1. Документ

В SQL данные, хранящиеся в строках и столбцах, содержатся в таблице. В базе данных NoSQL это называется документом. Документ представляет, как данные будут храниться в коллекции базы данных. Документы определяются так же, как имодель Pydantic, за исключением того, что вместо этого наследуется класс `Document` из библиотеки Beanie.

Пример документа определяется следующим образом:

```
from beanie import Document

class Event(Document):name: str
    location: str

    class Settings: name =
        "events"
```

Подкласс `Settings` определен, чтобы указать библиотеке создать имя коллекции,переданное в базе данных MongoDB.

Теперь, когда мы знаем, как создать документ, давайте рассмотрим методы,используемые для выполнения CRUD операций:

- `.insert()` и `.create()`: Методы `.insert()` и `.create()` вызываютсяэкземпляром

документа для создания новой записи в базе данных. Вы также можете использовать метод `.insert_one()` для добавления отдельной записи в базу данных.
Чтобы вставить много записей в базу данных, вызывается метод `.insert_many()`, который принимает список экземпляров документа, например:

```
event = Event(name="Packt office launch",location="Hybrid")
await event.create()
await Event.insert_one(event)
```

- `.find()` и `.get()`: Метод `.find()` используется для поиска списка документов, соответствующих критериям поиска, переданным в качестве аргумента метода. Метод `.get()` используется для получения одного документа, соответствующего предоставленному идентификатору. Отдельныйдокумент, соответствующий критерию поиска, можно найти с помощью метода `.find_one()`, например следующего:

```
event = await Event.get("74478287284ff")
event = await Event.find(Event.location == "Hybrid").to_
list()
# Returns a list of matching items
event = await.find_one(Event.location == "Hybrid") #
Returns a single event
```

- `.save()`, `.update()`, и `.upsert()`: Для обновления документа можно использовать любой из этих методов. Метод `.update()` принимает запрос на обновление, а метод `.upsert()` используется, когда документ не соответствует критериям поиска. В этой работе мы будем использовать метод `.update()`. Запрос на обновление — это инструкция,за которой следует база данных MongoDB, например, следующая:

```
event = await Event.get("74478287284ff")
update_query = {"$set": {"location": "virtual"}}
await event.update(update_query)
```

В этом блоке кода мы сначала извлекаем событие, а затем создаем запрос наобновление, чтобы установить для поля `location` в коллекции событий значение `virtual`.

- `.delete()`: Этот метод отвечает за удаление записи документа из базыданных, например:

```
event = await Event.get("74478287284ff")
await event.delete()
```

После изучения методов, содержащиеся в библиотеке Beanie, инициализируем базу данных в приложении планировщика событий, определим наши документы и реализуем CRUD операции.

1.2. Инициализация базы данных

Изучим шаги, необходимые чтобы сделать это:

1. В папке базы данных создайте модуль `connection.py`:

```
(venv)$ touch connection.py
```

Pydantic позволяет нам читать переменные среды, создавая дочерний классродительского класса `BaseSettings`. При создании веб-API стандартной практикой является хранение переменных конфигурации в файле среды.

2. В `connection.py`, добавьте следующее:

```
from beanie import init_beanie
from motor.motor_asyncio import AsyncIOMotorClient
from typing import Optional
from pydantic import BaseSettings

class Settings(BaseSettings):
    DATABASE_URL: Optional[str] = None

    async def initialize_database(self):
        client = AsyncIOMotorClient(self.DATABASE_URL)
        await init_beanie(
        database=client.get_default_database(),
                      document_models=[])

    class Config:
        env_file = ".env"
```

В этом блоке кода код начинается с импорта зависимостей, необходимых для инициализации базы данных. Затем определяется класс `Settings`, которыйимеет значение `DATABASE_URL`, которое считывается из среды `env`, определенной в подклассе `Config`. В коде также определяется метод `initialize_database` для инициализации базы данных.

Метод `init_beanie` принимает клиент базы данных, который представляетсобой версию движка mongo, созданную в разделе SQLModel, и список документов.

3. Обновим файлы модели в каталоге моделей, чтобы включитьдокументы MongoDB. В `models/events.py`, замените содержимое следующим:

```
from beanie import Document
from typing import Optional, List

class Event(Document):
    title: str
    image: str
    description: str
    tags: List[str]
    location: str

    class Config:
        schema_extra = {
            "example": {
                "title": "FastAPI Book Launch",
                "image": "https:
                //linktomyimage.com/image.png",
                "description": "We will be discussing
                the contents of the FastAPI book in
                this event. Ensure to come with your
                own copy to win gifts!",
                "tags": ["python", "fastapi", "book",
                "launch"],
                "location": "Google Meet"
            }
        }
    class Settings:
        name = "events"
```

4. Давайте создадим модель Pydantic для операций `UPDATE`:

```
class EventUpdate(BaseModel):title:
    Optional[str] image: Optional[str]
    description: Optional[str]tags:
    Optional[List[str]]location:
    Optional[str]

    class Config:
        schema_extra = {"example": {
                "title": "FastAPI Book Launch","image":
                "https:
                //linktomyimage.com/image.png", "description": "We will be
                discussingthe contents of the FastAPI book in this event.
                Ensure to come with yourown copy to win gifts!",
                "tags": ["python", "fastapi", "book","launch"],
                "location": "Google Meet"
            }
        }
```

5. В `model/users.py`, замените содержимое модуля следующим:

```
from typing import Optional, List
from beanie import Document, Link


from pydantic import BaseModel, EmailStr
from models.events import Event
```

```
class User(Document):
    email: EmailStr
    password: str
    events: Optional[List[Link[Event]]]

    class Settings:
        name = "users"

    class Config:
        schema_extra = {
            "example": {
                "email": "fastapi@packt.com",
                "password": "strong!!!",
                "events": [],
            }
        }

class UserSignIn(BaseModel):
    email: EmailStr
    password: str
```

6. Теперь, когда мы определили документы, давайте обновим поле `document_models` в `connection.py`:

```
from models.users import User
from models.events import Event

async def initialize_database(self):
        client = AsyncIOMotorClient(self.DATABASE_URL)
        await init_beanie(
        database=client.get_default_database(),
                        document_models=[Event, User])
```

7. Наконец, давайте создадим файл среды, `.env`, и добавим URL-адрес базыданных, чтобы завершить этап инициализации базы данных:

```
(venv)$ touch .env
(venv)$ echo DATABASE_URL=mongodb://localhost:27017/
planner >> .env
```

Теперь, когда мы успешно добавили блоки кода для инициализации базыданных, давайте приступим к реализации методов для CRUD операций.

2. CRUD операции

В `connection.py`, создайте новый класс `Database`, который принимает модельв качестве аргумента во время инициализации:

```
from pydantic import BaseSettings, BaseModel
from typing import Any, List, Optional


class Database:
    def __init__(self, model):
        self.model = model
```

Модель, передаваемая во время инициализации, представляет собой классмодели документа `Event` или `User`.

2.1. Создать

Давайте создадим метод в классе `Database`, чтобы добавить запись вколлекцию базы данных:

```
    async def save(self, document) -> None:
        await document.create()
            return
```

В этом блоке кода мы определили метод `save` для получения документа, который будет экземпляром документа, переданного в экземпляр `Database` вмомент создания экземпляра.

2.2. Читать

Давайте создадим методы для извлечения записи базы данных или всех записей,присутствующих в коллекции базы данных.:

```
    async def get(self, id: PydanticObjectId) -> Any:
        doc = await self.model.get(id)
        if doc:
            return doc
        return False


    async def get_all(self) -> List[Any]:
        docs = await self.model.find_all().to_list()
        return docs
```

Первый метод, `get()`, принимает идентификатор в качестве аргумента метода и возвращает соответствующую запись из базы данных, в то время как метод `get_all()` не принимает аргументов и возвращает список всех записей, имеющихся в базе данных.

2.3.Обновить

Давайте создадим метод для обработки процесса обновления существующей записи:

```
    async def update(self, id: PydanticObjectId, body:
    BaseModel) -> Any:
        doc_id = id
        des_body = body.dict()
        des_body = {k:v for k,v in des_body.items() if v is
        not None}
        update_query = {"$set": {
            field: value for field, value in
            des_body.items()
        }}

        doc = await self.get(doc_id)
        if not doc:
            return False
        await doc.update(update_query)
        return doc
```

В этом блоке кода метод `update` принимает ID и ответственную схему Pydantic, которая будет содержать поля, обновленные из запроса `PUT` отправленного клиентом. Обновленное тело запроса сначала анализируется в словаре, а затем фильтруется для удаления значений `None`.

Как только это будет сделано, он вставляется в запрос на обновление, который, наконец, выполняется методом `update()` Beanie.

2.4. Удалить

2.5Наконец, давайте создадим метод для удаления записи из базы данных:

```
    async def delete(self, id: PydanticObjectId) -> bool:
        doc = await self.get(id)
        if not doc:
            return False
        await doc.delete()
        return True
```

В этом блоке кода метод проверяет, существует ли такая запись, прежде чем приступить к ее удалению из базы данных.

Теперь, когда мы заполнили наш файл базы данных нужными методами, необходимыми для выполнения CRUD операций, давайте также обновим маршруты.

2.5. routes/events.py

Начнем с обновления импорта и создания экземпляра `database`:

```
from beanie import PydanticObjectId
from fastapi import APIRouter, HTTPException, status
from database.connection import Database

from models.events import Event
from typing import List
event_database = Database(Event)
```

Имея импорт и экземпляр базы данных, давайте обновим все маршруты. Начните с обновления маршрутов `GET`:

```
@event_router.get("/", response_model=List[Event])
async def retrieve_all_events() -> List[Event]:
    events = await event_database.get_all()
    return events

@event_router.get("/{id}",   response_model=Event)
async def retrieve_event(id: PydanticObjectId) -> Event:
    event = await event_database.get(id)
    if not event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"

        )

    return event
```

В маршрутах `GET` мы вызываем методы, которые мы определили ранее в модуле базы данных. Давайте обновим `POST` маршруты:

```
@event_router.post("/new")
async def create_event(body: Event) -> dict:
    await event_database.save(body)
    return {
        "message": "Event created successfully"
    }
```

Давайте создадим маршрут `UPDATE`:

```
@event_router.put("/{id}",   response_model=Event)
async def update_event(id: PydanticObjectId, body: EventUpdate)
-> Event:
    updated_event = await event_database.update(id, body)
    if not updated_event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
    return updated_event
```

Наконец, давайте обновим маршрут `DELETE`:

```
@event_router.delete("/{id}")
async def delete_event(id: PydanticObjectId) -> dict:
    event = await event_database.delete(id)
    if not event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
    return {

        "message": "Event deleted successfully."
```

Теперь, когда мы реализовали CRUD операции для наших маршрутов событий, давайте реализуем маршруты для регистрации пользователя и входа пользователя.

2.6. routes/users.py

Начнем с обновления импорта и создания экземпляра базы данных:

```
from fastapi import APIRouter, HTTPException, status
from database.connection import Database
from models.users import User, UserSignIn
user_router = APIRouter(
    tags=["User"],
)

user_database = Database(User)
```

Затем обновите маршрут `POST` для подписи новых пользователей:

```
@user_router.post("/signup")
async def sign_user_up(user: User) -> dict:
    user_exist = await User.find_one(User.email ==
    user.email)
    if user_exist:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail="User with email provided exists
            already."
        )
    await user_database.save(user)
    return {
        "message": "User created successfully"
    }
```

В этом блоке кода мы проверяем, существует ли такой пользователь с переданным адресом электронной почты, прежде чем добавлять его в базуданных. Давайте добавим маршрут для входа пользователей:

```
@user_router.post("/signin")
async def sign_user_in(user: UserSignIn) -> dict:
    user_exist = await User.find_one(User.email ==
    user.email)
    if not user_exist:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="User with email does not exist."
        )
    if user_exist.password == user.password:
        return {
            "message": "User signed in successfully."
        }
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Invalid details passed."
    )
```

В этом определенном маршруте мы сначала проверяем, существует ли пользователь, прежде чем проверять действительность его учетных данных. Используемый здесь метод аутентификации является базовым и *не рекомендуется* в производственной среде.

Теперь, когда мы реализовали маршруты, давайте запустим экземпляр MongoDB, а также наше приложение. Создайте папку для размещения нашейбазы данных MongoDB и запустите экземпляр

MongoDB:

```
(venv)$ mkdir store
(venv)$ mongod --dbpath store
```

Далее в другом окне запускаем приложение:

```
(venv)$ python main.py
INFO: Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
INFO: Started reloader process [3744] using statreload
INFO: Started server process [3747]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Давайте протестируем маршруты событий:

1. Создайте событие:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/event/new' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "title": "FastAPI Book Launch",
  "image": "https://linktomyimage.com/image.png",
  "description": "We will be discussing the contents
  of the FastAPI book in this event. Ensure to come
  with your own copy to win gifts!",
  "tags": [
    "python",
    "fastapi",
    "book",
    "launch"
  ],
  "location": "Google Meet"
}'
```

Вот ответ от предыдущей операции:

```
{
  "message": "Event created successfully"
}
```

2. Получить все события:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/' \
  -H 'accept: application/json'
```

Предыдущий запрос возвращает список событий:

```
[
  {
    "_id": "624daab1585059e8a3fa77ac",
    "title": "FastAPI Book Launch",
    "image": "https://linktomyimage.com/image.png",
    "description": "We will be discussing the contents
    of the FastAPI book in this event. Ensure to come
    with your own copy to win gifts!",
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
  }
]
```

3. Получить событие:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac' \
  -H 'accept: application/json'
```

Эта операция возвращает событие, соответствующее предоставленному ID:

```
{
  "_id": "624daab1585059e8a3fa77ac",
  "title": "FastAPI Book Launch",
  "image": "https://linktomyimage.com/image.png",
  "description": "We will be discussing the contents
  of the FastAPI book in this event. Ensure to come
  with your own copy to win gifts!",
  "tags": [
    "python",
    "fastapi",
    "book",
    "launch"
  ],
  "location": "Google Meet"
}
```

4. Обновим локацию события на `Hybrid`:

```
(venv)$ curl -X 'PUT' \
  'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac'
  \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "location": "Hybrid"
}'

{
  "_id": "624daab1585059e8a3fa77ac",
  "title": "FastAPI Book Launch",
  "image": "https://linktomyimage.com/image.png",
  "description": "We will be discussing the contents
  of the FastAPI book in this event. Ensure to come
  with your own copy to win gifts!",
  "tags": [
    "python","fastapi",
    "book",
    "launch"
  ],
  "location": "Hybrid"
}
```

5. Наконец, давайте удалим событие:

```
(venv)$ curl -X 'DELETE' \
  'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac'
  \
  -H 'accept: application/json'
```

Вот ответ, полученный на запрос:

```
{
  "message": "Event deleted successfully."
}
```

6. Теперь, когда мы протестировали маршруты для событий, давайтесоздадим нового пользователя, а затем войдем в систему:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signup' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "email": "fastapi@packt.com",
  "password": "strong!!!",
  "events": []
}'
```

Запрос возвращает ответ:

```
{
  "message": "User created successfully"
}
```

Повторный запуск запроса возвращает ошибку `HTTP 409`, указывающую наконфликт:

Изначально мы разработали маршрут для проверки существующихпользователей, чтобы избежать

```
{
  "detail": "User with email provided exists already."
}
```

дублирования.

7. Теперь давайте отправим `POST` запрос для входа только что созданномупользователю:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signin' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "email": "fastapi@packt.com",
  "password": "strong!!!"
}'
```

Запрос возвращает сообщение об успешном завершении `HTTP 200`:

```
{
  "message": "User signed in successfully."
}
```

Мы успешно реализовали CRUD-операции с помощью библиотеки Beanie.

Выводы

В этой работы мы освоили технику добавления баз данных SQL и NoSQL с помощью SQLModel и Beanie соответственно. Мы использовали все наши знания из предыдущих работ. Мы также проверили маршруты, чтобы убедиться, что ониработают по плану.

Задания

Описанные выше действия опираются на вариант их реализации в среде Linux/Mac. В среде Windows необходимы небольшие изменения в действиях или синтаксисе команд.

При выполнении работы обратите внимание, что код сервера и код клиента желательно запускать в разных экземплярах командной строки (в разных окнах).

- Освоить основные компоненты подключения и настройки MongoDB.
- Освоить операции в MongoDB с помощью Beanie.
- В соответствии с персональной темой создать структуру БД в рамках простого приложения FastAPI. Сформировать набора команд по действиям с записями в БД для MongoDB.
- Подготовить отчет и загрузить в СДО.
- Записать модель use case для этого варианта.
- Сформировать описание основных сценариев вариантов использования
- Для этих вариантов использования сформировать BPMN схемы