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

Урок Асин. работа

Асинхронная работа с базами данных

1. Введение

Мы с вами уже не новички, если дело касается баз данных, и прекрасно знаем, что некоторые запросы могут выполняться довольно долго. И, кажется, желание, чтобы наше приложение не простаивало в момент ожидания ответа от СУБД, вполне естественно. Это время можно потратить на что-то более полезное, например начать обрабатывать запросы от следующих клиентов (если наше приложение многопользовательское).

Ожидаемо, что таким желанием загорелись не только мы, ожидаемо, что решение задачи уже есть и нам надо только им воспользоваться.

2. Асинхронные драйвера для разных СУБД

Асинхронные библиотеки для Python существуют для всех популярных СУБД. Например, для PostgreSQL это `asyncpg`, для MySQL — `aiomysql`, для SQLite — `aiosqlite`. Все они работают по принципу, который мы уже знаем: вместо блокирования потока ожидания ответа от СУБД, мы запускаем запрос и ждем, когда он завершится.

Давайте посмотрим примеры для каждой из библиотек. Понятно, что для не встраиваемых СУБД, сначала будет необходимо поставить сервер, но мы эту часть

опустим и сосредоточимся непосредственно на питонячьем коде.

Пример для MySQL

Для начала, установим библиотеку `aiomysql`.

```
pip install aiomysql
```

Соберем базовый пример, который предполагает, что на нашем локальном компьютере установлен и запущен на порту 3306 процесс MySQL, а также существует база данных `test_db` с таблицей `my_table`, к которой есть доступ у пользователя `user` с паролем `password`.

```
import aiomysql
import asyncio

async def main():
    conn = await aiomysql.connect(host='localhost', port=3306, use
    async with conn.cursor() as cur:
        await cur.execute("SELECT * FROM my_table;")
        result = await cur.fetchall()
        print(result)
    conn.close()

asyncio.run(main())
```

Как видно из примера, процесс практически не отличается от того, что мы видели при синхронной работе, кроме использования асинхронного контекстного менеджера, и вызова функций в асинхронной обвязке. Вся работа сводится к нескольким обязательным пунктам:

1. Создание соединения с СУБД. На этом шаге необходимо передавать необходимые для подключения параметры: адрес сервера, порт, логин и пароль пользователя, имя базы данных.
2. Создание курсора. Некоторые библиотеки по работе с СУБД позволяют пропускать этап создания курсора и выполнять запросы непосредственно из объекта подключения к базе данных, но мы такое крайне не рекомендуем и советуем все равно создавать объект курсора, потому что единообразная работа с базой данных позволит избежать дополнительных проблем, если наш проект будет переноситься на другую СУБД, библиотека для которой подход без создания курсора не поддерживает.

3. Выполнение запросов и получение результатов. Тут мы делаем всякое, необходимое для работы приложения.

4. Закрытие соединения. Стандартный подход, если открыл — закрой. Каждая СУБД имеет некоторый пул подключений, который выдает новым клиентам, если мы будем создавать подключения и не закрывать их, то может возникнуть ситуация (в текущих версиях СУБД она скорее гипотетическая), когда у нашего сурвера накопится большое количество "спящих" соединений, которые еще не отвалились по таймауту и новым клиентам будет просто отказано в обслуживании.

Пример для PostgreSQL

Также установим библиотеку `asyncpg`.

```
pip install asyncpg
```

Соберем базовый пример, который предполагает, что на нашем локальном компьютере установлен и запущен на порту по умолчанию процесс PostgreSQL, а также существует база данных `test_db` с таблицей `my_table`, к которой есть доступ у пользователя `user` с паролем `password`.

```
import asyncpg
import asyncio

async def main():
    # Подключение к базе данных
    conn = await asyncpg.connect(user='user', password='password',

    # Создание курсора и выполнение запроса
    async with conn.transaction():
        async for record in conn.cursor("SELECT * FROM my_table;")
            print(record)

    # Закрытие соединения
    await conn.close()

asyncio.run(main())
```

Практически тоже самое, со скидкой на некоторые особенности синтаксиса самой библиотеки. `asyncpg` — не единственная асинхронная библиотека для работы с PostgreSQL, например очень популярная `Psycopg`, начиная с третьей версии также поддерживает асинхронные вызовы.

Пример для sqlite

Вернемся к нашей любимой встраиваемой базе данных sqlite. В отличие от синхронного драйвера, который идет в стандартной библиотеке Python, асинзронный надо устанавливать дополнительно, но его использование ничем не отличается от уже знакомого нам синтаксиса модуля `sqlite3`, у библиотеки `aiosqlite` даже в самом начале документации написано, что она полностью копирует интерфейс стандартного модуля, просто добавляя щепотку асинхронности.

```
pip install aiosqlite
```

```
import aiosqlite
import asyncio

async def main():
    conn = await aiosqlite.connect('test.db')
    async with conn.cursor() as cur:
        await cur.execute("SELECT * FROM my_table;")
        result = await cur.fetchall()
        print(result)
    conn.close()

asyncio.run(main())
```

3. Асинхронные ORM

SQLAlchemy

Мы с вами уже должны быть на текущий момент знакомы с библиотекой `slqalchemy`. Но использовали ее до этого только в синхронном режиме. Оказывается, что в недрах библиотеки скрыт модуль `sqlalchemy.ext.asyncio`, который позволяет подключать асинхронные драйвера. Давайте посмотрим пример:

```
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSessi
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Column, Integer, String, select
from sqlalchemy.ext.declarative import declarative_base
import asyncio
```

```
Base = declarative_base()

class MyModel(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

async def main():
    engine = create_async_engine('sqlite+aiosqlite:///test.db')
    async_session = sessionmaker(engine, expire_on_commit=False, c
    async with async_session() as session:
        async with engine.begin() as conn:
            await conn.run_sync(Base.metadata.create_all)

        async with session.begin():
            session.add(MyModel(name='example'))

        result = await session.execute(select(MyModel))
        for row in result.scalars():
            print(row)

    await engine.dispose()

asyncio.run(main())
```

Всего нескольно строчек, а именно создание асинхронного движка и создание асинхронной сессии (класса `AsyncSession`) и все готово!

Tortoise ORM

Понятно, что для решения любой частой проблемы в Python есть несколько способов. Поэтому существует сильно больше одной ORM, которые поддерживают асинхронное взаимодействие с базами данных. Но мы рассмотрим еще одну — Tortoise ORM. Такой выбор хочется обосновать тем, что эта библиотека создается с оглядкой на синтаксис суперпопулярной Django ORM (такая популярная она потому, что является неотъемлемой частью самого Django), поэтому разобравшись в ней можно убить сразу и второго зайца — изучить синтаксис Django ORM. Вторая причина в том, что эту ORM достаточно часто выбирают к связке fast-api + pydantic, с которыми мы познакомились на прошлом уроке.

Установим библиотеку:

```
pip install tortoise-orm
```

Конфигурацию к `tortoise-orm` принято хранить либо в json-файле, либо в python-модуле. Давайте создадим файл `config.py` и запишем в него следующее:

```
TORTOISE_ORM = {
    "connections": {
        "default": "sqlite://db.sqlite3"  # или "postgres://user:p
    },
    "apps": {
        "models": {
            "models": ["models"],  # Путь к вашим моделям
            "default_connection": "default",
        }
    },
}
```

Модели в Tortoise ORM определяются как классы, наследующиеся от `tortoise.models.Model`. Для примера давайте создадим модель в файле `models.py` (иммено этот файл мы указали в конфигурации выше), которая хранит информацию о пользователе.

```
from tortoise import fields, models

class User(models.Model):
    id = fields.IntField(pk=True)
    username = fields.CharField(max_length=50)
    email = fields.CharField(max_length=100)
    is_active = fields.BooleanField(default=True)

    class Meta:
        table = "user"

    def __str__(self):
        return self.username
```

Для инициализации базы данных и создания схемы используем следующие команды.

```
import asyncio
from tortoise import Tortoise

async def init():
    await Tortoise.init(config=TORTOISE_ORM)
```

```
    await Tortoise.generate_schemas()

asyncio.run(init())
```

А теперь несколько примеров на базовые операции с записями. Создание пользователя

```
import asyncio
from models import User
from tortoise import Tortoise, run_async
from config import TORTOISE_ORM

async def create_user():
    await Tortoise.init(config=TORTOISE_ORM)
    await Tortoise.generate_schemas()
    user = await User.create(username="john_doe", email="john@exam
    await user.save()  # Аналог commit()
    print(f"User created: {user}")

run_async(create_user())
```

Получение записи

```
import asyncio
from models import User
from tortoise import Tortoise, run_async
from config import TORTOISE_ORM

async def get_user():
    await Tortoise.init(config=TORTOISE_ORM)
    user = await User.get(username="john_doe")
    print(f"User retrieved: {user}")

run_async(get_user())
```

Редактирование записи

```
import asyncio
from models import User
from tortoise import Tortoise, run_async
from config import TORTOISE_ORM
```

```python
async def update_user():
    await Tortoise.init(config=TORTOISE_ORM)
    user = await User.get(username="john_doe")
    user.email = "john_doe@example.com"
    await user.save()
    print(f"User updated: {user}")

run_async(update_user())
```

Удаление записи

```python
import asyncio
from models import User
from tortoise import Tortoise, run_async
from config import TORTOISE_ORM

async def delete_user():
    await Tortoise.init(config=TORTOISE_ORM)
    user = await User.get(username="john_doe")
    await user.delete()
    print("User deleted")

run_async(delete_user())
```

4. Пример использования Tortoise ORM в связке с fast-api и Pydantic

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from tortoise.contrib.fastapi import register_tortoise
from models import User

app = FastAPI()

# Моделька для пользователя на вход
class UserIn(BaseModel):
    username: str
```

```
    email: str
    is_active: bool = True

# Моделька для пользователя на выход
class UserOut(BaseModel):
    id: int
    username: str
    email: str

# получаем из входящих параметров объект пользователя
# автоматичеки сериализуем пользователя в json на выходе после соз
@app.post("/users/", response_model=UserOut)
async def create_user(user: UserIn):
    user_obj = await User.create(**user.dict())
    return user_obj

# и тут
@app.get("/users/{user_id}", response_model=UserOut)
async def get_user(user_id: int):
    user = await User.get(id=user_id)
    if user:
        return user
    else:
        raise HTTPException(status_code=404, detail="User not foun

@app.put("/users/{user_id}", response_model=UserOut)
async def update_user(user_id: int, user: UserIn):
    user_obj = await User.get(id=user_id)
    if user_obj:
        user_obj.username = user.username
        user_obj.email = user.email
        user_obj.is_active = user.is_active
        await user_obj.save()
        return user_obj
    else:
        raise HTTPException(status_code=404, detail="User not foun

@app.delete("/users/{user_id}")
async def delete_user(user_id: int):
    user = await User.get(id=user_id)
    if user:
        await user.delete()
        return {"detail": "User deleted"}
    else:
```

```
        raise HTTPException(status_code=404, detail="User not foun

register_tortoise(
    app,
    db_url='sqlite://db.sqlite3',
    modules={'models': ['models']},
    generate_schemas=True,
    add_exception_handlers=True,
)

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

5. Заключение

В рамках этого блока уроков мы немного познакомились с концепцией асинхронного программирования и ее реализации в Python. Посмотрели, когда есть смысл в это влезать, а когда лучше посмотреть в сторону других парадигм программирования. А кроме того, прикоснулись к нескольким популярным библиотекам для асинхронной работы с вебом и базами данных.

Понятно, что это только базовое знакомство, но оно позволяет понять в какую сторону копать и на какие вещи обратить более пристальное внимание.

Надеюсь, что уроки были интересными и полезными. Хорошего вам асинхронного (и не только) программирования. Вы восхитительны!