**sync\_requests.py**

# Классический синхронный подход

import datetime

import requests

def get\_people(person\_id):

    return requests.get(f'http://127.0.0.1:5000/press/{person\_id}').json()

    # return requests.get(f'http://swapi.py4e.com/api/people/{person\_id}').json()

def main():

    response\_1 = get\_people(1)

    response\_2 = get\_people(2)

    response\_3 = get\_people(3)

    response\_4 = get\_people(4)

    print(response\_1, response\_2, response\_3, response\_4)

start = datetime.datetime.now()

main()

print(datetime.datetime.now() - start)

**async\_requests.py**

# Асинхронный подход

import asyncio # Для запуска асинхронного кода в приложении

async def get\_people(people\_id):

    return 42

# get\_people\_coroutine = get\_people(4)

# print(f'{get\_people\_coroutine=}')

# out: объект асинхронной функции - КОРУТИНА <coroutine object get\_people at 0x000000000067A4C0>

# Чтобы функции выполнялись асинхронно их нужно

# вызывать в event loop (цикле событий)

# Можно сказать, что корутины помещаются в цикл событий

# Чтобы НЕМЕДЛЕННО извлечь из

# результата асинхронной функции (корутины) ее значение

# нужно использовать БЛОКИРУЮЩЕЕ ключевое слово await

async def main():

    response = await get\_people(4) # значение корутины

    print(f'{response=}')

asyncio.run(main()) # run() - это метод создающий цикл событий event loop

#  в цикл событий передается main() для вызова асинхронных функций

# out:

# response=42

**async\_requests\_2.py**

# Асинхронный подход (продолжение)

import asyncio

async def get\_people(people\_id):

    return 42

async def main():

    # так код всеравно будет выполняться синхронно

    # потому что await не только извлекает значение объекта корутины, но и

    # одновременно выступает как БЛОКИРУЮЩЕЕ слово, т.е. пока

    # асинхронная функция не отработает следующая выполняться не начнет

    response\_1 = await get\_people(1) # начинает выполняться

    response\_2 = await get\_people(2) # ждет завершения предыдущей и начинает выполняться

    response\_3 = await get\_people(3) # ждет завершения предыдущей и начинает выполняться

    response\_4 = await get\_people(4) # ждет завершения предыдущей и начинает выполняться

    print(response\_1, response\_2, response\_3, response\_4)

    # так код будет выполняться асинхронно,

    # выполнение всех функций СТАРТУЕТ практически ОДНОМОМЕНТНО,

    # при этом есть риск, что мы можем не дождаться окончания выполнения

    # некоторых особо долгих из них, продолжив выполнять код дальше и

    # в конечном итоге получить ошибку (например не успеть записать

    # предыдущий результат в БД, см. пояснения в async\_requests\_4\_orm.py)

    coroutine\_1 = get\_people(1) # начинает выполняться

    coroutine\_2 = get\_people(2) # начинает выполняться

    coroutine\_3 = get\_people(3) # начинает выполняться

    coroutine\_4 = get\_people(4) # начинает выполняться

    # gather() в цикле событий event loop

    # проверяет выполнение АСИНХРОННЫХ ФУНКЦИЙ,

    # и КОНКУРЕНТНО, по мере ГАРАНТИРОВАННОГО завершения

    # их выполнения, складывает результат -

    # значения их объектов КОРУТИН в список.

    result = await asyncio.gather(coroutine\_1, coroutine\_2, coroutine\_3, coroutine\_4)

    # Поскольку gather() - это тоже асинхронная функция

    # чтобы извлечь из ее корутины результат (список значений)

    # нужно применить блокирующую операцию await

    print(result)

    # out: [42, 42, 42, 42]

# Передаем main() в цикл событий event loop,

# который создается с помощью метода run()

asyncio.run(main())

**async\_requests\_3.py**

# Асинхронный подход (продолжение)

import asyncio

# Библиотеку requests использовать мы не можем, т.к.

# чтобы асинхронные функции давали выигрыш

# они должны использоваться с другими асинхронными ф-ями,

# которые поставляются в специальных библиотеках

# аналогом библиотеки requests является библиотека aiohttp

import aiohttp

import datetime

from more\_itertools import chunked

async def get\_people(people\_id, session):

    # Отправляем запрос и получаем значение корутины с пом. await

    # response = await session.get(f'http://swapi.py4e.com/api/people/{people\_id}')

    response = await session.get(f'http://127.0.0.1:5000/press/{people\_id}')

    # Десериализуем. Получаем json. Ф-я тоже асинхронная (вообще

    # все ф-ии работы с сетью асинхронные)

    json\_data = await response.json()

    return json\_data

# Разделяем всю последовательность запросов на подпоследовательности

# чтобы не переполнить оперативку и не задедосить сервак

MAX\_COROTINS = 5

async def main():

    # Создаем сессию

    # session = aiohttp.ClientSession()

        # for coroutins\_chunk in chunked(range(1, 100), MAX\_COROTINS):

        #     coroutins = [get\_people(i, session) for i in coroutins\_chunk]

        #     result = await asyncio.gather(\*coroutins)

        #     print(result)

    # await session.close()

    # Так как объект сессии является асинхронным менеджером контекста

    # можно заменить так

    # Создаем сессию

    async with aiohttp.ClientSession() as session:

        # выгружаем 100 персонажей асинхронно (конкурентно)

        # разделяем последовательность из 100 элементов на подпоследовательности по 5 элементов

        for coroutins\_chunk in chunked(range(1, 100), MAX\_COROTINS):

            coroutins = [get\_people(i, session) for i in coroutins\_chunk]

            result = await asyncio.gather(\*coroutins)

            print(result)

start = datetime.datetime.now()

asyncio.run(main())

print(datetime.datetime.now() - start)

**models.py**

import os

# Используем асинхронный движок для работы с БД

from sqlalchemy.ext.asyncio import create\_async\_engine

# Используем фабрику асинхронных сессий

from sqlalchemy.ext.asyncio import async\_sessionmaker

# Используем базовый класс для создания моделей

from sqlalchemy.orm import DeclarativeBase

# Импортируем класс АИНХРОННЫЕ АТРИБУТЫ

from sqlalchemy.ext.asyncio import AsyncAttrs

from sqlalchemy.orm import mapped\_column, Mapped # колонки и типизация

# Импортируем типы данных

from sqlalchemy import Integer, JSON

POSTGRES\_USER = os.getenv("POSTGRES\_USER", "postgres")

POSTGRES\_PASSWORD = os.getenv("POSTGRES\_PASSWORD", "admin")

POSTGRES\_DB = os.getenv("POSTGRES\_DB", "test")

POSTGRES\_HOST = os.getenv("POSTGRES\_HOST", "127.0.0.1")

POSTGRES\_PORT = os.getenv("POSTGRES\_PORT", "5432")

# DSN строка отлична для работы в АСИНХРОННОМ РЕЖИМЕ

PG\_DSN = f"postgresql+asyncpg://{POSTGRES\_USER}:{POSTGRES\_PASSWORD}@" \

    f"{POSTGRES\_HOST}:{POSTGRES\_PORT}/{POSTGRES\_DB}"

# Создаем асинхронный движок для работы с БД

engine = create\_async\_engine(PG\_DSN)

# Создаем фабрику асинхронных сессий

Session = async\_sessionmaker(bind=engine, expire\_on\_commit=False)

# expire\_on\_commit=False - отключаем закрытие сессии после первого же коммита

# Создаем базовый класс для создания моделей

class Base(DeclarativeBase, AsyncAttrs):

    # мы хотим чтобы некоторые атрибуты нашей модели

    # стали асинхронными поэтому делаем

    # множественное наследование от класса AsyncAttrs

    pass

# Создаем нашу модель

class SwapiPeople(Base):

    \_\_tablename\_\_ = "swapi\_people"

    # Смотрим наши JSON (те которые мы получаем при обращении на сервак)

    # Задаем поля сопоставляя питоновский тип с постгрессовским

    id: Mapped[int] = mapped\_column(Integer, primary\_key=True)

    json: Mapped[dict] = mapped\_column(JSON, nullable=True) # nullable=True - обязательное поле

# Запросы к БД будут уходить асинхронно,

# соответственно они должны быть внутри event loop

# Инициализируем ORM

async def init\_orm():

    # для этого из engine с помощью менеджера контекста with

    # вырвем одно подключение и в нем с помощью

    # "запускатора" run.sync() выполним запрос в базу на создание таблиц,

    # которые привязаны к модели, что создаст таблицу "swapi\_people"

    async with engine.begin() as connect:

        # Решаем проблему НАСЛОЕНИЯ

        # Удаляем предыдущие записи

        await connect.run\_sync(Base.metadata.drop\_all)

        await connect.run\_sync(Base.metadata.create\_all)

# Функция, закрывающая ORM

async def close\_orm():

    await engine.dispose()

**async\_requests\_4\_orm.py**

# Асинхронный подход (продолжение:выгружаем в ORM)

import asyncio

import aiohttp

import datetime

from more\_itertools import chunked

# Импортируем из models.py инициализацию и закрытие БД

# фабрику сессий и нашу модель

from models import init\_orm, close\_orm, Session, SwapiPeople

MAX\_COROTINS = 5

async def get\_people(people\_id, session):

    # response = await session.get(f'http://swapi.py4e.com/api/people/{people\_id}')

    response = await session.get(f'http://127.0.0.1:5000/press/{people\_id}')

    json\_data = await response.json()

    return json\_data

# Пишем функцию для вставки в БД значений списка result gather-а,

# содержащего пять вложенных JSON (это будут отдельные экземпляры ORM-модели)

async def insert\_to\_database(list\_json: [dict]):

    # # открываем сессию

    # async with Session() as session:

        # objects = []

        # for item in list\_json:

        #     # Создаем экземпляр SwapiPeople и кладем в него JSON

        #     swapi = SwapiPeople(json=item)

        #     objects.append(swapi)

        # # Добавляем все объекты в сессию

        # session.add\_all)objects

        # # Комиттимся

        # await session.commit()

    # Перерписываем этот код с помощью компрекейшн выражения

    async with Session() as session:

        objects = [SwapiPeople(json=item) for item in list\_json]

        session.add\_all(objects)

        await session.commit()

async def main():

    # Инициализируем orm

    await init\_orm()

    async with aiohttp.ClientSession() as session:

        for coroutins\_chunk in chunked(range(1, 100), MAX\_COROTINS):

            coroutins = [get\_people(i, session) for i in coroutins\_chunk]

            result = await asyncio.gather(\*coroutins) # ждем завершения формирования пятерки JSON-ов

            # Вставляем result в БД

            # await insert\_to\_database(result)

            # Используя асинхронные инструменты оптимизируем эту часть кода

            # await - это БЛОКИРУЮЩАЯ операция,

            # но ведь мы можем асинхронно начать выгружать с сервера

            # следующую пятерку JSON-ов не дожидаясь вставки

            # в БД предыдущих выгруженных

            # Это можно организовать с помощью объекта пайтона Task (ЗАДАЧА):

            # Задаем в методе create\_task объект корутину insert\_to\_database(result)

            asyncio.create\_task(insert\_to\_database(result)) # НЕ ЖДЕМ ЗАВЕРШЕНИЯ ЗАПИСИ В БД

            # !!! ОПЕРАЦИЯ СОЗДАНИЯ КОРУТИНЫ ТЕПЕРЬ НЕ БЛОКИРУЮЩАЯ !!!

            # !!! И ПЕРЕД asyncio.create\_task() НЕТ БЛОК. ОПЕРАЦИИ await !!!

            # !!! В ОТЛИЧАЕ ОТ await asyncio.gather() !!!

            # Таким образом операция вставки в БД начинает выполняться,

            # но при этом мы продолжаем выполнять код дальше (начинаем

            # формировать сльедующую пятерку JSON-ов)

            # НЕ ДОЖИДАЯСЬ ЗАВЕРШЕНИЯ ВЫПОЛНЕНИЯ ОПЕРАЦИИ ВСТАВКИ В БД

            # print(result)

    # ЕЩЕ ОДНА ПРОБЛЕМА асинхронного кода - последние 5 JSON-ов не попали в БД.

    # Это происходит потому, что последния задача вставки в БД insert\_to\_database(result)

    # не успевает завершиться, а сессия orm уже закрывается,

    # можно попробывать дождаться завершения выполнения

    # последней задачи insert\_to\_database(result) применением await task,

    # await task # ждем завершения последней задачи вставки в БД insert\_to\_database(result)

    # но тогда может не успеть завершиться

    # предпоследняя задача insert\_to\_database(result),

    # которая по времени выполнения будет дольше

    # !!! В этом ОСОБЕННОСТЬ АСИНХРОННОГО КОДА -

    # мы никогда не знаем наверняка когда асинхронная функция завершиться !!!

    # а пайтон никогда не ждет завершения функций

    # КАК ГАРАНТИРОВАТЬ ВЫПОЛНЕНИЕ ЗАДАЧ?

    tasks = asyncio.all\_tasks()

    # Это функция обращается к текущему event loop и забирает из него все незавершенные задачи,

    # т.е. те, что еще выполняются сейчас и выгружает эти задачи в виде множества.

    # print(tasks)

    # out:

    # {

    #     <Task pending name='Task-1' coro=<main() running at async\_requests\_4\_orm.py:96> cb=[\_run\_until\_complete\_cb() at C:\Users\GIS27\AppData\Local\Programs\Python\Python38\lib\asyncio\base\_events.py:184]>,

    #     <Task pending name='Task-239' coro=<insert\_to\_database() running at async\_requests\_4\_orm.py:39>

    #     wait\_for=<Future pending cb=[BaseProtocol.\_on\_waiter\_completed(), <TaskWakeupMethWrapper object at

    #     0x00000000056B8400>()]>>}

    # Дальше мы можем снова закинуть их в гезер и выполнить с завершением конкурентно,

    # но туда же попадет и сама асинхронная функция main() (см. out выше)

    # и наше приложение зацикливается - мы получаем РЕКУРСИЮ

    # Поэтому сначала получаем текущую задачу main()

    current\_task = asyncio.current\_task()

    # и далее из множества задач tasks удаляю ее

    tasks.remove(current\_task)

    # print(tasks)

    # out:

    # {

    #     <Task pending name='Task-239' coro=<insert\_to\_database() running at async\_requests\_4\_orm.py:39> wait\_for=<Future pending cb=[BaseProtocol.\_on\_waiter\_completed(), <TaskWakeupMethWrapper object at 0x00000000056BA460>()]>>}

    # Теперь в гезере мы конкурентно выполняем с завершением

    # все текущие невыполненные задачи кроме main()

    await asyncio.gather(\*tasks)

    # Закрываем сессию orm

    await close\_orm()

start = datetime.datetime.now()

asyncio.run(main())

print(datetime.datetime.now() - start)