В прошлых статьях я сопоставил синхронные и асинхронные версии Django и FastAPI. Тут я лишь возьму лучшие показатели, уберу у Django мидлвары и еще раз запущу приложения на трёх воркерах.

Напомню, что для замеров я использую веб-сервис Artillery со следующим конфигом:

!@#$

config:

  target: 'http://127.0.0.1:80'

  phases:

    - duration: 60

      arrivalRate: 1

      rampTo: 200

  plugins:

    ensure: {}

    apdex: {}

    metrics-by-endpoint: {}

  apdex:

    threshold: 100

scenarios:

  - flow:

      - loop:

          - get:

              url: '/async\_get\_all\_comments'

        count: 1

!@#$

для FastAPI и

!@#$

config:

  target: 'http://127.0.0.1:8080'

  phases:

    - duration: 60

      arrivalRate: 1

      rampTo: 200

  plugins:

    ensure: {}

    apdex: {}

    metrics-by-endpoint: {}

  apdex:

    threshold: 100

scenarios:

  - flow:

      - loop:

          - get:

              url: '/test/sync\_get\_all\_comments'

        count: 1

!@#$

для Django

Обращу внимание на параметр **rampTo**, равным 200. В прошлых тестах он был 120. Просто я заранее знаю, что 3 воркера легко справятся с 120 запросами, поэтому решил повысить нагрузку.

Начнём с прошлого призёра. Конфиг FastAPI тот же:

Файл **models** с моделями:

!@#$

from sqlalchemy import Integer, String, ForeignKey  
from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from database import Base  
  
  
class UserModel(Base):  
 \_\_tablename\_\_ = "user"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, autoincrement=True)  
 username: Mapped[str] = mapped\_column(String, unique=True) # добавлен unique=True для уникальности  
  
 # Define relationship if needed  
 comments: Mapped[list["CommentsModel"]] = relationship("CommentsModel", back\_populates="author")  
  
  
class ArticleModel(Base):  
 \_\_tablename\_\_ = 'article'  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, autoincrement=True)  
 title: Mapped[str] = mapped\_column(String, unique=True) # добавлен unique=True для уникальности  
  
 # Define relationship if needed  
 comments: Mapped[list["CommentsModel"]] = relationship("CommentsModel", back\_populates="article")  
  
  
class CommentsModel(Base):  
 \_\_tablename\_\_ = 'comment'  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, autoincrement=True)  
 text: Mapped[str] = mapped\_column(String)  
 author\_id: Mapped[int] = mapped\_column(ForeignKey("user.id"))  
 article\_id: Mapped[int] = mapped\_column(ForeignKey("article.id"))  
  
 # Define relationships  
 author: Mapped["UserModel"] = relationship("UserModel", back\_populates="comments")  
 article: Mapped["ArticleModel"] = relationship("ArticleModel", back\_populates="comments")

!@#$

Файл **database**:

!@#$

from sqlalchemy.orm import DeclarativeBase  
from sqlalchemy.ext.asyncio import create\_async\_engine, async\_sessionmaker, AsyncAttrs  
  
  
SQLALCHEMY\_DATABASE\_URL = "postgresql+asyncpg://admin:admin@db:5432/admin"  
engine = create\_async\_engine(  
 SQLALCHEMY\_DATABASE\_URL  
), expire\_on\_commit=False)  
  
  
class Base(AsyncAttrs, DeclarativeBase):  
 pass

!@#$

Файл **dependencies.py** с зависимостями:

!@#$

from database import async\_session  
  
  
async def get\_db():  
 async with async\_session() as session:  
 yield session

!@#$

Вьюха:

!@#$

@app.get("/async\_get\_all\_comments")  
async def async\_get\_all\_comments(db: AsyncSession = Depends(get\_db)):  
 result = await db.execute(  
 select(CommentsModel)  
 .filter(  
 CommentsModel.id < 100  
 )  
 )  
 return result.scalars().all()

!@#$

Запускать приложения я буду в докере с PostgreSQL в качестве БД. Dockerfile выглядит следующим образом:

!@#$

FROM python:3.12  
  
WORKDIR /code  
  
COPY ./requirements.txt /code/requirements.txt  
  
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt  
  
COPY . /code

!@#$

Файл docker-compose.yaml выглядит так:

!@#$

version: "3.0"  
  
services:  
 back:  
 build: .  
 container\_name: "backend"  
 ports:  
 - "80:80"  
 restart: unless-stopped  
 command: sh -c "gunicorn -w 3 -k uvicorn.workers.UvicornWorker main:app --bind 0.0.0.0:80"db:  
 image: postgres:16.3-alpine3.20  
 container\_name: db  
 environment:  
 POSTGRES\_USER: admin  
 POSTGRES\_PASSWORD: admin  
 POSTGRES\_DB: admin  
 volumes:  
 - db\_data:/var/lib/postgresql/data  
 ports:  
 - "5432:5432"  
 restart: unless-stopped  
 depends\_on:  
 - back  
volumes:  
 db\_data:

!@#$

В этот раз запущу gunicorn с тремя воркерами

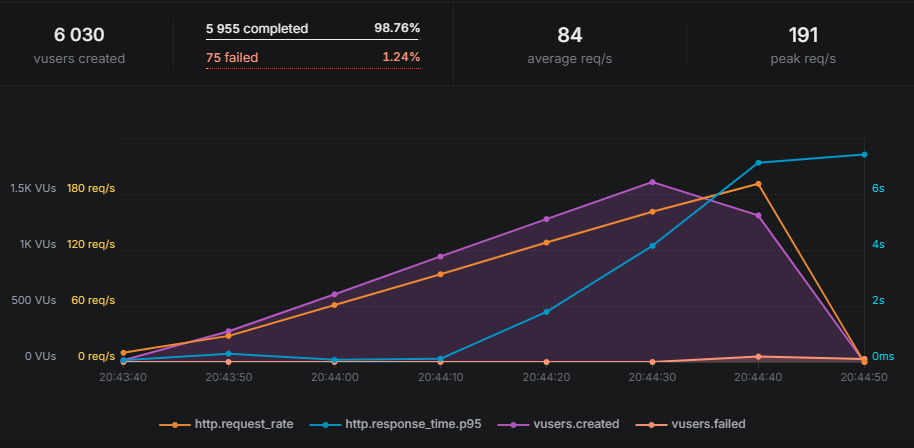
Команда для запуска тестов:

!@#$

artillery run --name async\_fastapi\_3\_worker C:\Users\na2ar\Desktop\asciiart-load-test.yml --record --key <сюда вставить свой ключ>

!@#$

Результат:



Асинхронный FastAPI с 3 воркерами выполнил практически все запросы без ошибок. Лишь на пике при 191 запросе в секунду произошло несколько ошибок.

Запустим лучший конфиг Django из статьи по сравнению синхрона против асинхрона: - синхронный Django. В той статье я предположил, что синхронные вьюхи слишком быстро принимают запросы, что БД, не имея ПО, которое обеспечивало бы асинхронную или какую-либо другую параллельную работу, просто крашится из-за количества подключений. Напомню, что тогда в логах начали появляться следующие ошибки:



Это произошло при использовании адаптера psycopg3, имеющего по заявлению разработчиков поддержку асинхронной работы. Как и в прошлый раз, не исключаю возможность того, что я просто что-то неправильно подключил :)

И так, код вьюхи:

!@#$

class SyncCommentsView(View):  
   
 def get(self, request, \*args, \*\*kwargs):  
 cr = services.get\_all\_comments()  
 response\_data = [{'id': comment.id, 'title': comment.text} for comment in cr]  
 return JsonResponse(response\_data, safe=False)

!@#$

метод **get\_all\_comments()** из файла services:

!@#$

def get\_all\_comments():  
 return CommentsModel.objects.filter(id\_\_lte=100)

!@#$

Мидлвары в файле **Settings.py** выглядят так:



Оставшиеся 2 мидлвара отключить не получалось, даже закомментировав практически весь файл

Это позволит хоть немного уравнять силы Django перед FastAPI

В файле **docker-compose.yaml** в сервис бекаввел**:**

!@#$

command: sh -c "python manage.py makemigrations &&  
 python manage.py migrate &&  
 gunicorn -w 3 config.wsgi:application --bind 0.0.0.0:8080"

!@#$

Так я запущу 3 воркера, используя спецификацию wsgi. Обращу внимание, что я не указал в качестве воркера класс ювикорна. Это сделано из-за того, что wsgi не может работать с асинхронными воркерами. Такая же команда запустит дефолтные синхронные воркеры гуникорна

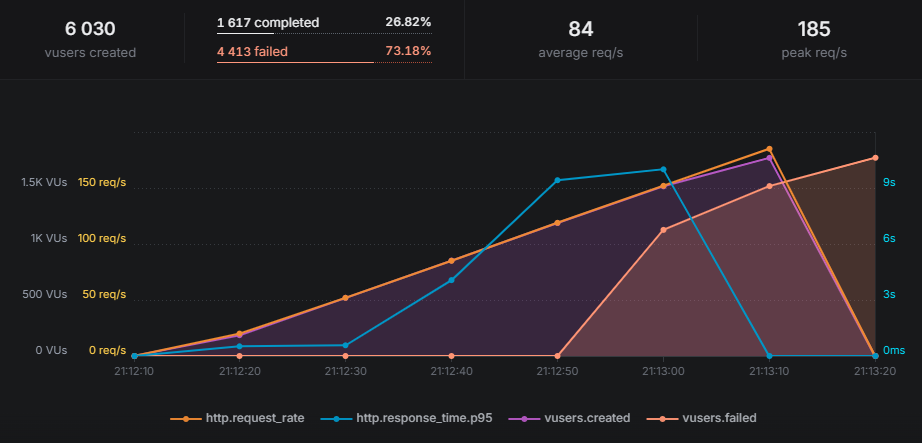
Запускаю тест.

!@#$

artillery run --name sync\_django\_3\_worker C:\Users\na2ar\Desktop\asciiart-load-test.yml --record --key a9\_aFHDUaLZgOtZsTFxB1sP6yf5fiOWxgp0

!@#$

Результат:



Что ж, Django смог успешно достичь обработки 119 запросов в секунду до того, как начал сыпать ошибки. Момент повышения индикатора vusers.failed можно считать конечной точкой, т.к. в этом месте некоторые запросы начинают возвращаться с ошибкой

Сравнивая с FastAPI:

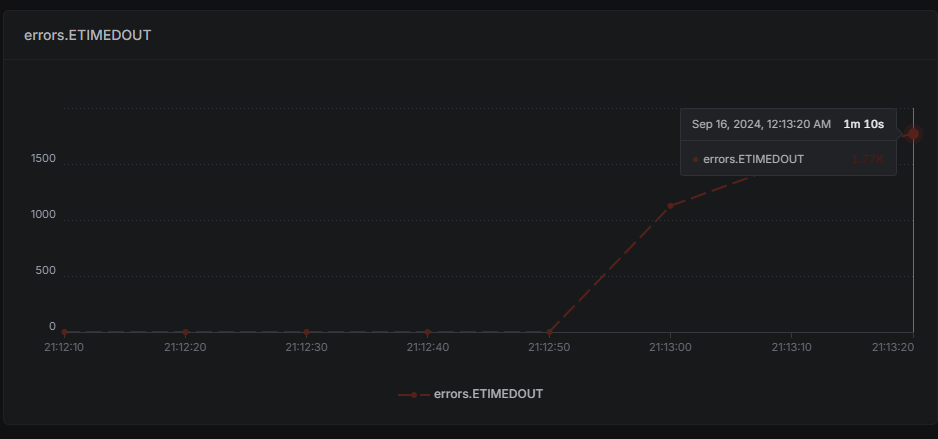
Максимальный RPS Django: 119 req/s

Максимальный RPS FastAPI: 161 req/s

Среднее время ответа 95% ответов Django на пике нагрузки: 9.4 сек

Среднее время ответа 95% ответов FastAPI на пике нагрузки: 4.1 сек

Количество ошибок ETIMEDOUT (Слишком долгое время ожидания ответа от сервера) на пике нагрузки Django (спойлер: 100%):



Количество ошибок ETIMEDOUT (Слишком долгое время ожидания ответа от сервера) на пике нагрузки FastAPI:

