Асинхронное программирование может быт куда сложнее синхронного в написании, понимании и отладке.

Сравним подходы к написанию кода и выясним по конкретным цифрам, стоит ли игра свеч.

Для измерения скорости работы веб-приложений я буду использовать веб-сервис Artillery. Он позволяет проводить нагрузочное тестирование легко и наглядно. Установить его легко, одной командой

Такой конфиг я буду использовать для тестирования:

!@#$

config:

target: 'http://127.0.0.1:8080'

phases:

- duration: 60

arrivalRate: 1

rampTo: 120

plugins:

ensure: {}

apdex: {}

metrics-by-endpoint: {}

apdex:

threshold: 100

scenarios:

- flow:

- loop:

- get:

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

count: 1

!@#$

Параметр **duration** означает, что тестирование будет длиться 60 секунд, параметр **arrivalRate** означает, что будет создать один виртуальный юзер, который выполнит запрос к моему API, а **rampTo** означает, что в течение тестирования количество виртуальных юзеров будет расти постепенно так, чтобы до конца процесса было создано 120 виртуальных юзеров.

Принцип же работы тестирования в ежесекундном увеличении количества виртуальных юзеров, отправляющих запросы на указанный url **'/test/sync\_get\_all\_comments'**. Параметр **count: 1** означает, что каждый виртуальный юзер выполнит 1 запрос и будет уничтожен. В следующую секунду будет создано другое количество юзеров, отправляющих запрос.

Начнём с синхронного кода, для которого я написал синхронную вьюху:

!@#$

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:

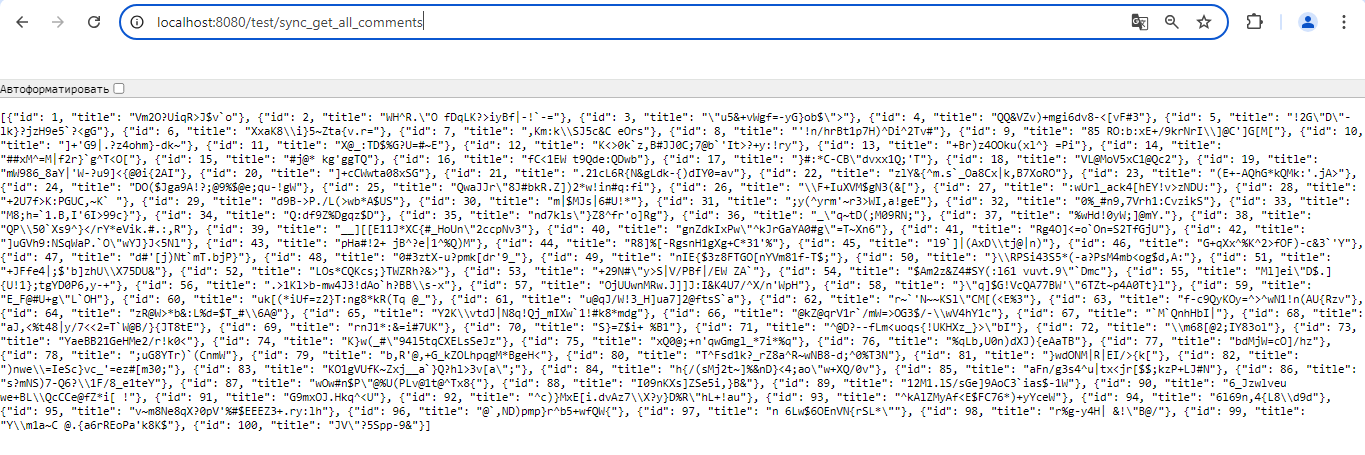
!@#$

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

!@#$

Я буду получать первые 100 записей из БД.

Вывод этого запроса выглядит так:



Я просто сгенерировал случайные строки и добавил их в БД.

Сами модели выглядят так:

!@#$

UserModel = get\_user\_model()

class ArticleAuthorModel(models.Model):  
 author = models.ForeignKey(UserModel, models.CASCADE)  
 article = models.ForeignKey("ArticleModel", models.CASCADE)  
  
  
class ArticleModel(models.Model):  
 title = models.CharField(max\_length=80)  
 email = models.CharField(max\_length=50)  
 author = models.ManyToManyField(UserModel, "articles", through=ArticleAuthorModel)  
  
 class Meta:  
 db\_table = "ArticleModel"  
  
  
class CommentsModel(models.Model):  
 text = models.TextField()  
 article = models.ForeignKey(ArticleModel, models.CASCADE, "comments")  
 author = models.ForeignKey(UserModel, models.CASCADE, "comments")  
  
 class Meta:  
 db\_table = "CommentsModel"

!@#$

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"  
 command: sh -c "python manage.py makemigrations && python manage.py migrate && gunicorn config.wsgi:application --bind 0.0.0.0:8080"  
 ports:  
 - "8080:8080"  
 restart: unless-stopped  
 depends\_on:  
 - db  
  
 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  
  
  
volumes:  
 db\_data:

!@#$

Важно отметить, что в качестве сервера приложений я использую не встроенный тестовый сервер джанги, а gunicorn. Разработчики джанги сами настоятельно советуют не использовать их сервер в чем угодно кроме отладки и разработки. Еще важно отметить, что gunicorn не запустится на виндусе, т.к. он включает в себя модуль **fctnl**, не поддерживающий винду.

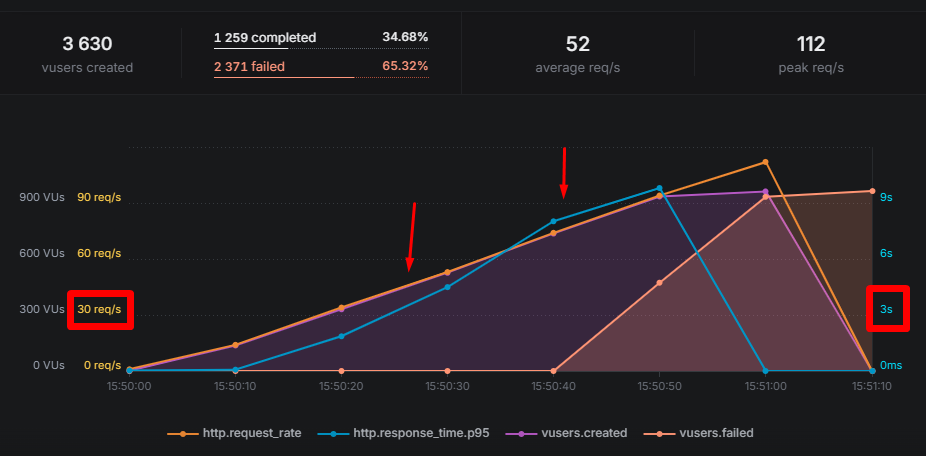
Командой

!@#$

artillery run --name sync\_django\_1\_worker C:\Users\na2ar\Desktop\asciiart-load-test.yml --record --key <сюда вставить свой ключ>

!@#$

Запускаю тестирование. Через 72 секунды получаю результат:



Видно, что со временем количество запросов, юзеров и время ответа росли. Самый важный на мой взгляд этап начался в точке, обозначенной второй красной стрелкой. Здесь выше нуля стал показатель vusers.failed. Он означает, что юзер создался, отправил запрос, и, не получив ответа, умер. Т.е. произошла ошибка и ответ не вернулся совсем. До него все запросы обрабатывались с ответом 200, но с возрастающей до 9.8 сек на пике скоростью. Сверху видно, что успешно выполнились лишь 34.68% юзеров.

Далее пробуем синхронный подход. Меняю в конфиге Artillery параметр **url** наurl: **'/test/async\_get\_all\_comments'**

Вью выглядит так:

!@#$

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

!@#$

функция так:

!@#$

async def async\_get\_all\_comments():  
 return await sync\_to\_async(get\_all\_comments)()

!@#$

Файл docker-compose.yaml выглядит так:

!@#$

version: "3.0"  
  
services:  
 back:  
 build: .  
 container\_name: "backend"command: sh -c "python manage.py makemigrations &&  
 python manage.py migrate &&  
 gunicorn -w 1 -k uvicorn.workers.UvicornWorker config.asgi:application --bind 0.0.0.0:8080"  
 ports:  
 - "8080:8080"  
 restart: unless-stopped  
 depends\_on:  
 - db  
  
 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  
  
  
volumes:  
 db\_data:

!@#$

Я изменил команду запуска сервера приложений:

!@#$

gunicorn -w 1 -k uvicorn.workers.UvicornWorker config.asgi:application --bind 0.0.0.0:8080

!@#$

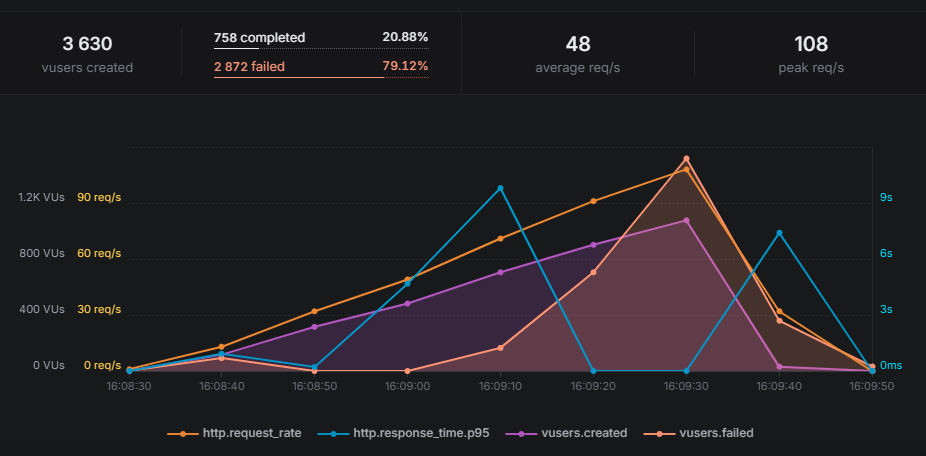
теперь гуникорн запускается с одним асинхронным воркером и использует спецификацию asgi вместо wsgi. Для установки сервера приложений uvicorn, воркеры которого используются для асинхронного выполнения кода, надо установить его командой:

!@#$

pip install uvicorn

!@#$

Далее запускаю тестирование всё той же командой и получаю результат:



Видно, что количество отправленных запросов стало меньше. На пике – 108 запросов в секунду против 112 в синхронной версси. К тому же, количество убитых юзеров было не равно нуля практически весь период тестирования. Пик времени ответа пришелся на 40 секунду и дальше упал до нуля, в то время как показатель смертей юзеров (ответы с ошибкой) достигли пика. Еще один факт – во время тестирования в логах бекенда я получил много таких сообщений:



База данных просто не смогла справиться с количеством запросов и возвращала ошибки. Это учитывая, что я использовал psycopg3, который должен поддерживать асинхронность по заявлениям разработчиков. Возможно я что-то не так подключил :)

Таким образом, можно сделать вывод, что асинхронный код быстрее принимал входящие запросы и отправлял запрос в БД, чем синхронный. Однако база оказалась не в состоянии работать асинхронно или как угодно по-другому параллельно обрабатывать запросы. Поэтому выгоду от использования асинхронности можно увидеть в запросах, не использующих БД. В новых версиях джанго было совершенно много шагов в сторону асинхронности, но на фоне FastAPI он до сих пор отстаёт.