В прошлой статье я сравнил синхронный и асинхронный Django. Теперь проверим FastAPI

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

Конфиг для тестирования я использую тот же, изменив лишь порт и url:

!@#$

config:

target: 'http://127.0.0.1:80'

phases:

- duration: 60

arrivalRate: 1

rampTo: 120

plugins:

ensure: {}

apdex: {}

metrics-by-endpoint: {}

apdex:

threshold: 100

scenarios:

- flow:

- loop:

- get:

url: '/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 **'/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:

!@#$

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

!@#$

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:

!@#$

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 выглядит так:

!@#$

from sqlalchemy import create\_engine  
from sqlalchemy.orm import DeclarativeBase, sessionmaker  
  
  
SQLALCHEMY\_DATABASE\_URL = "postgresql://admin:admin@db:5432/admin"  
engine = create\_engine(  
 SQLALCHEMY\_DATABASE\_URL  
)  
  
sync\_session = sessionmaker(bind=engine, expire\_on\_commit=False)  
  
  
class Base(DeclarativeBase):  
 pass

!@#$

Файл с зависимостями:

!@#$

from database import sync\_session  
  
  
def get\_db():  
 db = sync\_session()  
 try:  
 yield db  
 finally:  
 db.close()

!@#$

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 1 -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:

!@#$

При тестировании синхронного django, я не использовал для gunicorn асинхронный uvicorn воркер, но FastAPI не поддерживает синхронные воркеры, которые gunicorn использует по умолчанию. Также в прошлой статье я говорил я писал, что gunicorn включает в себя модуль **fctnl**, который не поддерживает винду, из-за чего придется запускать приложение в докере.

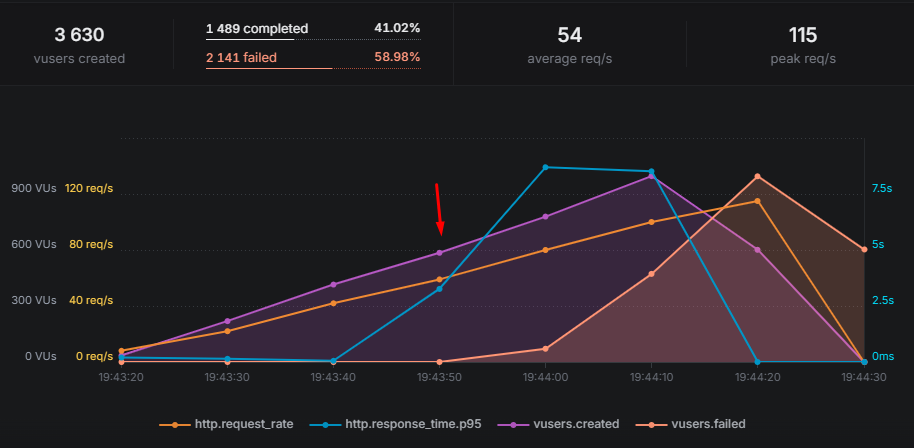
С помощью команды:

!@#$

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

!@#$

Запускаю тестирование. Через 73 секунды получаю результат:



Видно, что со временем количество запросов, юзеров и время ответа росли. Самый важный на мой взгляд этап начался в точке, обозначенной 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 3.2 сек на пике скоростью. Сверху видно, что успешно выполнились 41.02% юзеров.

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

Вью выглядит так:

!@#$

@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()

!@#$

В файле **database** я изменил диалект бд на **asyncpg**. Для создания сессий использую **async\_sessionmaker**:

!@#$  
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 с зависимостями такой:

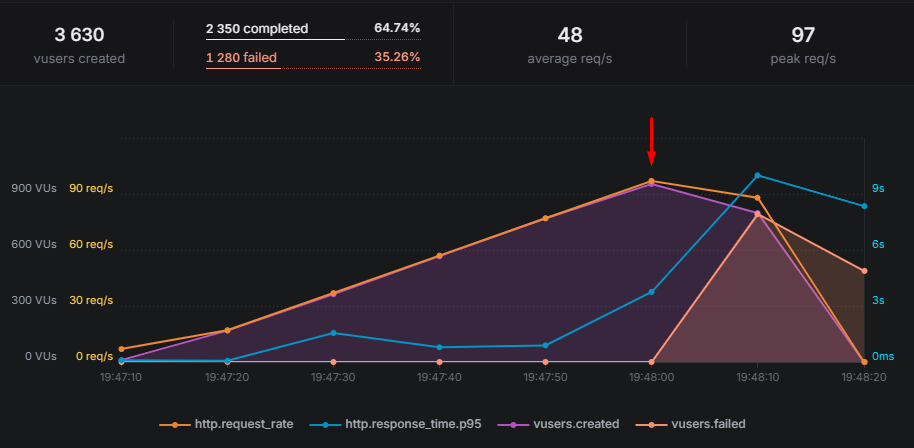
!@#$

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

!@#$

Файл docker-compose.yaml выглядит также

Далее запускаю тестирование всё той же командой и получаю результат:



Результат асинхронного кода превзошел синхронный наглядно. Если синхрон смог достичь успешной обработки лишь 59 запросов в секунду до начала смерти юзеров (ответов с ошибкой), то асинхрон достиг 97 успешно обработанных запросов в секунду. То как росло время ответа (голубая линия) по сравнению в синхронным кодом – занавес! Асинхронный код держал время ответа ниже секунды (кроме одной точки) практически до конца. Синхронный же сдался через 20 секунд и заставил приложение грузиться столько, сколько человек не будет ждать и закроет его

Вывод напрашивается сам собой – асинхронность в контексте FastAPI даёт очевидное преимущество в производительности

Сравнивая результаты с джанго – выбор очевиден. Однако в следующей статье я сравню лучший конфиг Django против лучшей сборки FastAPI