⚡ Практическое занятие

Метрики производительности и CAP-теорема

ФИО студента:

Алексеева Анастасия Андреевна

Группа:

ИМО-426

Дата:

21.10.2025

📊 Основные метрики производительности

Percentiles (Процентили)

- **P50 (медиана):** 50% запросов быстрее этого времени
- **P95:** 95% запросов быстрее
- **P99:** 99% запросов быстрее
- **P99.9:** 99.9% запросов быстрее

⚖️ CAP-теорема

Правило: В распределенной системе можно гарантировать только 2 из 3 свойств:

Практические компромиссы:

- **CP системы:** MongoDB, Redis Cluster (жертвуют доступностью)
- **AP системы:** Cassandra, DynamoDB (жертвуют согласованностью)
- **CA системы:** PostgreSQL, MySQL (не устойчивы к разделению)

🔒 ACID свойства

D - Durability (Долговечность)
Зафиксированные изменения сохраняются навсегда
Пример: После подтверждения транзакции данные не теряются даже при сбое

📈 Availability (Доступность)

Формула: Availability = (Uptime / Total Time) × 100%

SLA уровни:

🧪 Практические задания

Требования к выполнению:

- выполняются на ОС **Linux** (любой дистрибутив)
- пакеты устанавливаются в **виртуальное окружение venv**
- для создания серверов можно использовать **LLM** (желательно Claude Sonnet подключить в IDE)

Задание 1: Установка и использование Apache Bench

Установка Apache Bench на Linux:

```
# Ubuntu/Debian
sudo apt-get install apache2-utils

# CentOS/RHEL/Fedora
sudo yum install httpd-tools
# или
sudo dnf install httpd-tools

# Проверка установки
ab -V
```

Тестирование производительности:

```
# Запуск 1000 запросов с 10 одновременными соединениями
ab -n 1000 -c 10 http://httpbin.org/get
```

Результаты тестирования:

Время выполнения теста:

119.15 секунд

Requests per second:

8.39 запросов/сек

Time per request (mean):

1191.53 мс

Статистика соединений (мс):

	min	mean	median	max
Connect:	128	156	142	1223
Processing:	130	1019	794	5740

Перцентили времени ответа:

P50: 949 мс P90: 2270 мс P95: 2715 мс P99: 3740 мс

Описание полученных результатов:

Сервер продемонстрировал среднюю производительность и заметные колебания времени отклика. Среднее время ответа составило около 1,2 секунды, что указывает на умеренную задержку при обработке запросов. Скорость обработки — 8,39 запросов в секунду — сравнительно низкая, а наличие 17 неудачных и неуспешных запросов говорит о возможных ограничениях или нестабильности под нагрузкой. При этом большая часть запросов (до 90%) была выполнена в пределах 2–2,3 секунд, что свидетельствует о приемлемом уровне стабильности для публичного API, но не для высоконагруженного продакшен-сервиса.

📊 Подробный анализ результатов Apache Bench

Общая информация о тесте:

Утилита: ApacheBench 2.3
URL: http://httpbin.org/get
Количество запросов: 1000
Уровень параллелизма: 10 одновременных соединений

Параметры сервера:

```
Server Software: awselb/2.0 ← AWS Elastic Load Balancer
Server Hostname: httpbin.org
Server Port: 80 ← HTTP порт
Document Path: /get ← Тестируемый endpoint
Document Length: 162 bytes ← Размер ответа
```

Расшифровка статистики соединений:

- **Connect:** Время установки TCP-соединения
- **Processing:** Время обработки запроса сервером
- **Waiting:** Время от отправки запроса до получения первого байта
- **Total:** Общее время запроса

Анализ качества:

Failed requests: 17 (ошибок нет) Transfer rate: 3.94 Kbytes/sec

Выводы о производительности:

```
Стабильность:
Довольно высокое стандартное отклонение (≈800 мс) говорит о
колебаниях времени отклика. Это может быть вызвано сетевой
задержкой или нестабильностью сервера.

Производительность:
Показатель 8.39 запросов/сек — относительно низкий. Для теста с 10
параллельными соединениями это говорит о высокой задержке на
стороне сервера или маршрута.

Задержки:
Среднее время ответа 1.2 секунды — значительно выше, чем в
эталонном тесте (≈300 мс). Это может быть связано с географической
удалённостью httpbin.org и ограничениями публичного API.

Ошибки:
17 неудачных запросов (ошибки длины или статуса). Это указывает на
то, что при высокой нагрузке часть ответов некорректна — возможно,
сервер ограничивает количество обращений или отбрасывает часть
запросов.

Масштабируемость:
90% запросов обслуживаются за ≤2.3 секунды, 95% — до 2.7 секунды.
Для публичного API это приемлемо, но для реальных продакшен-
сервисов показатель высокий
```

Рекомендации для улучшения:

- Оптимизировать можно за счет кеширования
- Рассмотреть CDN для уменьшения задержек
- Увеличить параллелизм для проверки пределов сервера

⏱️ Задание 2: Измерение задержек (latency)

Простой HTTP сервер с задержками (Flask):

```
from flask import Flask
import time
import random

app = Flask(__name__)

@app.route('/fast')
def fast_endpoint():
    return {'response': 'fast', 'latency': '10ms'}

@app.route('/slow')
def slow_endpoint():
    time.sleep(random.uniform(0.1, 0.5))  # 100-500ms задержка
    return {'response': 'slow', 'latency': '100-500ms'}

if __name__ == '__main__':
    app.run(port=5000)
```

Измерение latency:

```
# Тестирование быстрого endpoint
ab -n 100 -c 5 http://localhost:5000/fast

# Тестирование медленного endpoint
ab -n 100 -c 5 http://localhost:5000/slow
```

Результаты /fast endpoint:

RPS:

337.96

Средняя задержка:

14.8 мс

Результаты /slow endpoint:

RPS:

14.24

Средняя задержка:

351 мс

Сравнение веб-фреймворков

FastAPI (порт 8000)

Код на Python:

```
from fastapi import FastAPI
import time
import random
import subprocess

app = FastAPI()

@app.get("/fast")
async def fast_endpoint():
    return {"response": "fast", "latency": "10ms"}

@app.get("/slow")
async def slow_endpoint():
    delay = random.uniform(0.1, 0.5)
    time.sleep(delay)  "color: #95a5a6;"># задержка 100-500ms
    return {"response": "slow", "latency": "100-500ms"}

@app.get('/test-api')
def test_api():
    "color: #95a5a6;"># Запускаем ApacheBench с нужными параметрами
    cmd = ['ab', '-n', '100', '-c', '5', 'http://localhost:8000/fast']
    try:
        result = subprocess.run(cmd, capture_output=True, text=True, timeout=30)
        output = result.stdout
        error = result.stderr
        if result.returncode != 0:
            return {"error": "Ошибка при запуске ab", "details": error}
        return {"результаты теста": output}
    except Exception as e:
        return {"error": "Исключение при запуске ab", "details": str(e)}

@app.get('/test-1')
def test_endpoint():
    return {
        'message': 'FastAPI test endpoint',
        'commands': [
            'ab -n 1000 -c 10 http://localhost:8000/fast',
            'ab -n 1000 -c 10 http://localhost:8000/slow',
            'ab -n 1000 -c 10 http://localhost:8000/test'
        ],
        'expected_rps': '~500-1000 for /fast, ~20-50 for /slow'
    }

if __name__ == "__main__":
```

```
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

803.64

Средняя задержка:

6.2 мс

Результаты /slow endpoint:

RPS:

3.26

Средняя задержка:

1533 мс

Краткие выводы:

FastAPI демонстрирует высокую производительность и низкие задержки на быстром endpoint, обрабатывая более 800 запросов в секунду. При работе с медленным endpoint скорость падает ожидаемо из-за искусственных пауз, но сервер остаётся стабильным и без ошибок. В сравнении с Flask, FastAPI показывает лучшую пропускную способность и меньшие задержки, что подтверждает его эффективность

Django (порт 8001)

Код на Python:

```
import os
import django
from django.conf import settings
from django.http import JsonResponse
from django.urls import path
from django.core.wsgi import get_wsgi_application
import time
import random

"color: #95a5a6;"># Минимальная конфигурация Django
settings.configure(
    DEBUG=True,
    SECRET_KEY='test-key',
    ROOT_URLCONF=__name__,
    ALLOWED_HOSTS=['*'],
)
django.setup()

def fast_endpoint(request):
    return JsonResponse({'response': 'fast', 'latency': '10ms'})

def slow_endpoint(request):
    time.sleep(random.uniform(0.1, 0.5))  "color: #95a5a6;"># 100-500ms задержка
```

```
    return JsonResponse({'response': 'slow', 'latency': '100-500ms'})

urlpatterns = [
    path('fast/', fast_endpoint),
    path('slow/', slow_endpoint),
]

if __name__ == '__main__':
    from django.core.management import execute_from_command_line
    execute_from_command_line(['manage.py', 'runserver', '0.0.0.0:8001'])
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

253.10

Средняя задержка:

19.8 мс

Результаты /slow endpoint:

RPS:

229.31

Средняя задержка:

21.8 мс

Краткие выводы:

Django показал высокую стабильность и относительно низкое время отклика как на быстрых, так и на медленных endpoint-ах. Однако производительность (RPS) ниже, чем у FastAPI, что ожидаемо из-за синхронной модели обработки запросов. В целом, Django остаётся надёжным, но менее быстрым решением по сравнению с асинхронным FastAPI.

Django REST Framework (порт 8002)

Код на Python:

```
import os
import django
from django.conf import settings
import time
import random

"color: #95a5a6;"># Минимальная конфигурация Django + DRF
settings.configure(
    DEBUG=True,
    SECRET_KEY='test-key',
    ROOT_URLCONF=__name__,
    ALLOWED_HOSTS=['*'],
```

```
    INSTALLED_APPS=[
        'django.contrib.contenttypes',
        'django.contrib.auth',
        'rest_framework',
    ],
    REST_FRAMEWORK={
        'DEFAULT_RENDERER_CLASSES': [
            'rest_framework.renderers.JSONRenderer',
        ]
    }
)
django.setup()

from django.urls import path
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET'])
def fast_endpoint(request):
    return Response({'response': 'fast', 'latency': '10ms'})

@api_view(['GET'])
def slow_endpoint(request):
    time.sleep(random.uniform(0.1, 0.5))  "color: #95a5a6;"># 100-500ms задержка
    return Response({'response': 'slow', 'latency': '100-500ms'})

@api_view(['GET'])
def test_api(request):
    return Response({'результаты теста': '123'})

@api_view(['GET'])
def api_root(request):
    return Response({
        'endpoints': {
            'fast': '/api/fast/',
            'slow': '/api/slow/',
            'test-api': '/api/test-api/'
        }
    })

from django.urls import include

urlpatterns = [
    path('api/', api_root),
    path('api/', include([
        path('fast/', fast_endpoint),
        path('slow/', slow_endpoint),
        path('test-api/', test_api),
    ])),
]

if __name__ == '__main__':
```

```
    from django.core.management import execute_from_command_line
    execute_from_command_line(['manage.py', 'runserver', '0.0.0.0:8002'])
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

236.97

Средняя задержка:

21.1 мс

Результаты /slow endpoint:

RPS:

205.85

Средняя задержка:

24.3 мс

Краткие выводы:

Django REST Framework демонстрирует стабильную работу и умеренную производительность. Несмотря на небольшие задержки, сервер остаётся надёжным при одинаковом уровне параллельности. По сравнению с “чистым” Django, DRF работает немного медленнее из-за дополнительного слоя сериализации и обработки данных, однако разница незначительна. FastAPI по-прежнему остаётся самым быстрым из протестированных фреймворков.

LiteStar (порт 8003)

Код на Python:

```
import time
import random
from litestar import Litestar, get

@get("/fast")
def fast_endpoint() -> dict:
    return {"response": "fast", "latency": "10ms"}

@get("/slow")
def slow_endpoint() -> dict:
    time.sleep(random.uniform(0.1, 0.5))  "color: #95a5a6;"># задержка 100-500ms
    return {"response": "slow", "latency": "100-500ms"}

app = Litestar(route_handlers=[fast_endpoint, slow_endpoint])

if __name__ == "__main__":
```

```
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8003)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

1297.99

Средняя задержка:

3.85 мс

Результаты /slow endpoint:

RPS:

3.28

Средняя задержка:

1522 мс

Краткие выводы:

LiteStar показал наилучшие показатели производительности среди всех протестированных фреймворков на быстром endpoint – более 1200 запросов в секунду при минимальной задержке менее 4 мс. При медленном endpoint производительность аналогична FastAPI, с предсказуемыми задержками из-за sleep. LiteStar демонстрирует высокую эффективность асинхронной обработки и подходит для систем, требующих максимальной скорости и минимальных накладных расходов.

Общие выводы о производительности веб-фреймворков:

На основе проведённых нагрузочных тестов можно сделать следующие выводы:

LiteStar — показал наивысшую производительность: более 1200 запросов в секунду и минимальные задержки (около 4 мс). Благодаря асинхронной архитектуре и лёгкому ядру, LiteStar подходит для высоконагруженных и real-time приложений.

FastAPI — также продемонстрировал отличную скорость (≈800 RPS, задержка ~6 мс) и стабильность. Это современный, асинхронный фреймворк, хорошо сбалансированный между производительностью и удобством разработки.

Django — работает медленнее, чем асинхронные решения (≈250 RPS), однако отличается стабильностью и предсказуемыми результатами. Его синхронная природа ограничивает пиковую производительность, но компенсируется зрелой экосистемой и надёжностью.

Django REST Framework (DRF) — показал чуть более низкую скорость (≈230 RPS) из-за дополнительного слоя сериализации и обработки данных. При этом остаётся устойчивым и предсказуемым инструментом для построения API-сервисов.

Flask — обеспечивает среднюю производительность (≈338 RPS для fast-endpoint) и подходит для лёгких приложений и прототипов, но не для высокой нагрузки.

📝 Домашнее задание

Задача: Доделать задания

Срок сдачи: К следующему занятию

Что нужно сделать:

- Протестировать 3 разных сайта с помощью Apache Bench
- Сравнить их производительность
- Построить график зависимости RPS от количества одновременных соединений
- Создать HTTP серверы на всех указанных фреймворках
- Провести сравнительное тестирование производительности

📊 Критерии оценки

📚 Литература

- Клеппман М. "Высоконагруженные приложения"
- Фаулер М. "Архитектура корпоративных программных приложений"
- Документация Apache Bench
- Официальная документация веб-фреймворков