Exercise 8

Simple Benchmarking with wrk

Prior Knowledge
Previous exercises

Objectives
Benchmarking runtimes

Software Requirements

- docker-compose
- wrk - a simple benchmarking tool

Overview
We will look at using a benchmarking tool to call our APIs very fast and see how they react.

Steps

1. Start your service using "docker-compose up"

   (If you don't have a working service but want to try this anyway then do this:

   ```
   cd ~
   git clone https://github.com/pzfreo/purchase-complete.git
   cd purchase-complete
   yarn install
   docker-compose up --build
   )
   ```

2. wrk should already be installed.

   Test that the binary is there. Start a new terminal window and type:
   `wrk --help`

```
oxsoa@oxsoa:~$ wrk --help
Usage: wrk <options> <url>
  Options:
    -c, --connections <N>  Connections to keep open
    -d, --duration    <T>  Duration of test
    -t, --threads     <N>  Number of threads to use

    -s, --script      <S>  Load Lua script file
    -H, --header      <H>  Add header to request
        --latency          Print latency statistics
        --timeout     <T>  Socket/request timeout
    -v, --version          Print version details

  Numeric arguments may include a SI unit (1k, 1M, 1G)
  Time arguments may include a time unit (2s, 2m, 2h)
```

3. Now we can run a test:
   `wrk -c 100 -d 1m -t 10 http://localhost:8000/purchase`

4. This will constantly hit our server with 100 concurrent clients calling over 1 minute (using 10 threads).

5. typeorm and postgres are meant to implement connection pooling automatically. However, the first time I run this I get connection pooling issues in both the postgres container and the purchase container:

```
2021-03-28 14:46:36.503 UTC [3003] FATAL:  sorry, too many clients already
purchase_1  | error: sorry, too many clients already
```

6. This shows up in the wrk output

```
wrk -c 100 -d 2m -t 10 http://localhost:8000/purchase
Running 1m test @ http://localhost:8000/purchase
  10 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    59.67ms   89.02ms   1.98s    95.32%
    Req/Sec   208.74     60.48   300.00     84.82%
  123949 requests in 1.00m, 80.22MB read
  Socket errors: connect 0, read 0, write 0, timeout 21
  Non-2xx or 3xx responses: 1357
Requests/sec:   2056.85
Transfer/sec:      1.33MB
```

7. Notice the 1357 error responses I got back.

8. If you rerun this however, everything seems warmed up and ready to work reliably:

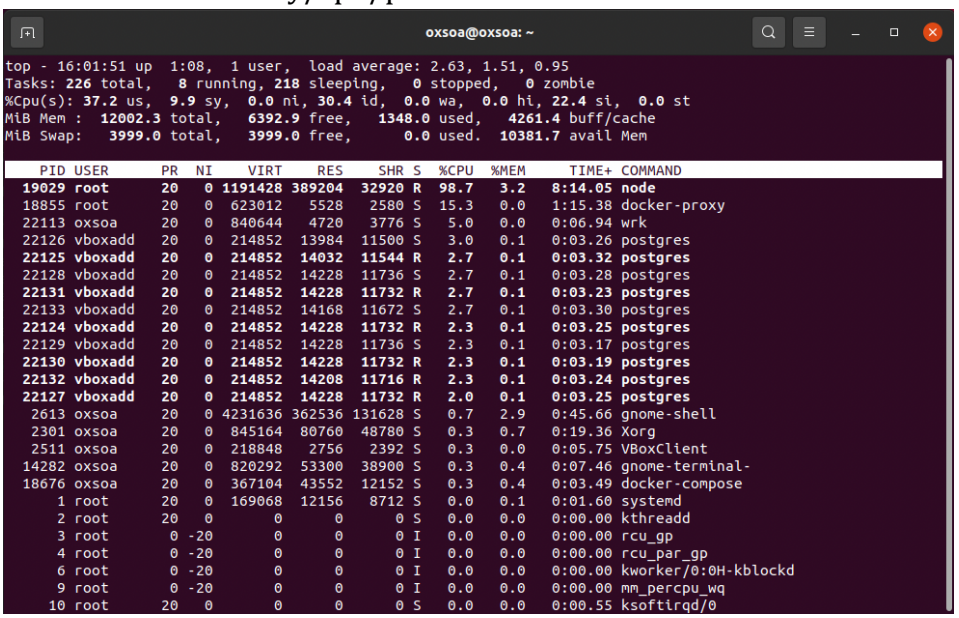
```
wrk -c 100 -d 1m -t 10
http://localhost:8000/purchase
Running 1m test @ http://localhost:8000/purchase
  10 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    49.31ms    9.97ms 183.09ms   92.12%
    Req/Sec   204.88     41.52   303.00     73.40%
  122170 requests in 1.00m, 78.06MB read
Requests/sec:   2034.45
Transfer/sec:      1.30MB
```

9. I think 2000+ requests per second accessing a database is reasonable. You may get different results on your setup of course.

10. While it is running you can monitor the CPU.
    Extend the time to run longer (e.g. 5m) and rerun

    Open up a new terminal window and type:
    `top`

11. You will see a memory/cpu/process monitor.



    If you want to read more about load averages, this is a good read:
    http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html

12. You will see that there is only one node process here. Node is single-threaded, so on a multi-core system you might want to run more. In a Kubernetes environment you would run multiple replicas behind a load-balancer. In other systems you can use tools like pm2 to automatically scale up node instances:
    https://github.com/Unitech/pm2

13. Note that this is not a real performance analysis. Ideally the servers would be on a separate machine from the client load drivers (siege engines!). Also, microservices are designed to be run in parallel in multiple containers with load balancing across them, so this model is not the recommended way of running either deployment.

14. That's all for this lab!