

# 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.

```

top - 16:01:51 up 1:08, 1 user, load average: 2.63, 1.51, 0.95
Tasks: 226 total, 8 running, 218 sleeping, 0 stopped, 0 zombie
%Cpu(s): 37.2 us, 9.9 sy, 0.0 ni, 30.4 id, 0.0 wa, 0.0 hi, 22.4 si, 0.0 st
MiB Mem : 12002.3 total, 6392.9 free, 1348.0 used, 4261.4 buff/cache
MiB Swap: 3999.0 total, 3999.0 free, 0.0 used, 10381.7 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
19029 root        20   0 1191428 389204 32920 R  98.7   3.2   8:14.05 node
18855 root        20   0 623012   5528   2580 S  15.3   0.0   1:15.38 docker-proxy
22113 oxsoa       20   0 840644   4720   3776 S   5.0   0.0   0:06.94 wrk
22126 vboxadd  20   0 214852  13984  11500 S   3.0   0.1   0:03.26 postgres
22125 vboxadd  20   0 214852  14032  11544 R   2.7   0.1   0:03.32 postgres
22128 vboxadd  20   0 214852  14228  11736 S   2.7   0.1   0:03.28 postgres
22131 vboxadd  20   0 214852  14228  11732 R   2.7   0.1   0:03.23 postgres
22133 vboxadd  20   0 214852  14168  11672 S   2.7   0.1   0:03.30 postgres
22124 vboxadd  20   0 214852  14228  11732 R   2.3   0.1   0:03.25 postgres
22129 vboxadd  20   0 214852  14228  11736 S   2.3   0.1   0:03.17 postgres
22130 vboxadd  20   0 214852  14228  11732 R   2.3   0.1   0:03.19 postgres
22132 vboxadd  20   0 214852  14208  11716 R   2.3   0.1   0:03.24 postgres
22127 vboxadd  20   0 214852  14228  11732 R   2.0   0.1   0:03.25 postgres
2613 oxsoa       20   0 4231636 362536 131628 S   0.7   2.9   0:45.66 gnome-shell
2301 oxsoa       20   0 845164  80760  48780 S   0.3   0.7   0:19.36 Xorg
2511 oxsoa       20   0 218848   2756   2392 S   0.3   0.0   0:05.75 VBoxClient
14282 oxsoa       20   0 820292  53300  38900 S   0.3   0.4   0:07.46 gnome-terminal-
18676 oxsoa       20   0 367104  43552  12152 S   0.3   0.4   0:03.49 docker-compose
  1 root        20   0 169068   12156   8712 S   0.0   0.1   0:01.60 systemd
  2 root        20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
  3 root        0 -20      0      0      0 I   0.0   0.0   0:00.00 rcu_gp
  4 root        0 -20      0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
  6 root        0 -20      0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
  9 root        0 -20      0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
 10 root        20   0      0      0      0 S   0.0   0.0   0:00.55 ksoftirqd/0

```

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!