Exercise 8

Simple Benchmarking with autocannon

Prior Knowledge
Previous exercises

Objectives
Benchmarking runtimes

Software Requirements

- docker-compose
- autocannon - a simple benchmarking tool

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

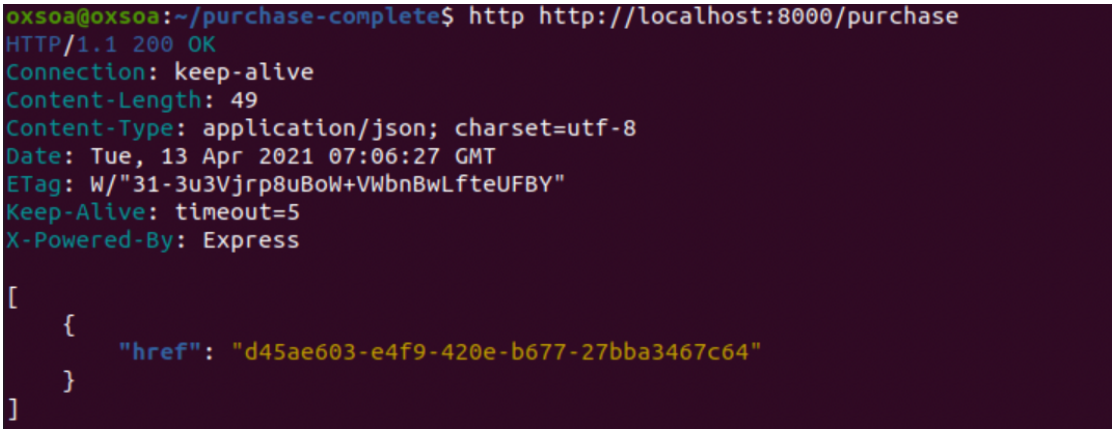
Steps

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
   )
   ```

   Test your service is up and running:
   http http://localhost:8000/purchase



2. Let's install autocannon.

   Autocannon needs a newer version of node than is installed, so let's upgrade:

   ```
   curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash -
   sudo apt install nodejs
   ```

   Now install autocannon:

   ```
   yarn global add autocannon
   ```

   ```
   autocannon --help
   ```

```
Usage: autocannon [opts] URL

URL is any valid http or https url.
If the PORT environment variable is set, the URL can be a path. In that case 'http://localhost:$PORT/path' will be used as the URL.

Available options:

  -c/--connections NUM
        The number of concurrent connections to use. default: 10.
  -p/--pipelining NUM
        The number of pipelined requests to use. default: 1.
  -d/--duration SEC
        The number of seconds to run the autocannon. default: 10.
  -a/--amount NUM
        The amount of requests to make before exiting the benchmark. If set, duration is ignored.
  -S/--socketPath
        A path to a Unix Domain Socket or a Windows Named Pipe. A URL is still required in order to send the correct Host header and path.
  -w/--workers
        Number of worker threads to use to fire requests.
  --on-port
        Start the command listed after -- on the command line. When it starts listening on a port,
        start sending requests to that port. A URL is still required in order to send requests to
        the correct path. The hostname can be omitted, `localhost` will be used by default.
  -m/--method METHOD
        The http method to use. default: 'GET'.
  -t/--timeout NUM
        The number of seconds before timing out and resetting a connection. default: 10
  -T/--title TITLE
        The title to place in the results for identification.
  -b/--body BODY
        The body of the request.
```

3. Now we can run a test:
   `autocannon -c 100 -d 60 -w 4 http://localhost:8000/purchase`

4. This will constantly hit our server with 100 concurrent clients calling over 60 seconds (using 4 worker threads).

5. You should see something like:

```
oxsoa@oxsoa:~/purchase-complete$ autocannon -c 100 -d 60 -w 4 http://localhost:
000/purchase
Running 60s test @ http://localhost:8000/purchase
100 connections
4 workers
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	33 ms	45 ms	109 ms	129 ms	50.94 ms	19.88 ms	282 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	860	897	2055	2751	1943.79	508.75	860
Bytes/Sec	244 kB	255 kB	584 kB	781 kB	552 kB	144 kB	244 kB

```
Req/Bytes counts sampled once per second.

117k requests in 60.04s, 33.1 MB read
```

6. In my run I had zero errors.
   If there are errors there will be a line like:

   ```
   120274 2xx responses, 97 non 2xx responses
   ```

7. Rerun it now everything is warmed up.

```
Running 60s test @ http://localhost:8000/purchase
100 connections
4 workers
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	33 ms	39 ms	66 ms	88 ms	41.27 ms	10.95 ms	215 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1077	1143	2547	2791	2393.99	381.85	1077
Bytes/Sec	306 kB	325 kB	723 kB	793 kB	680 kB	108 kB	306 kB

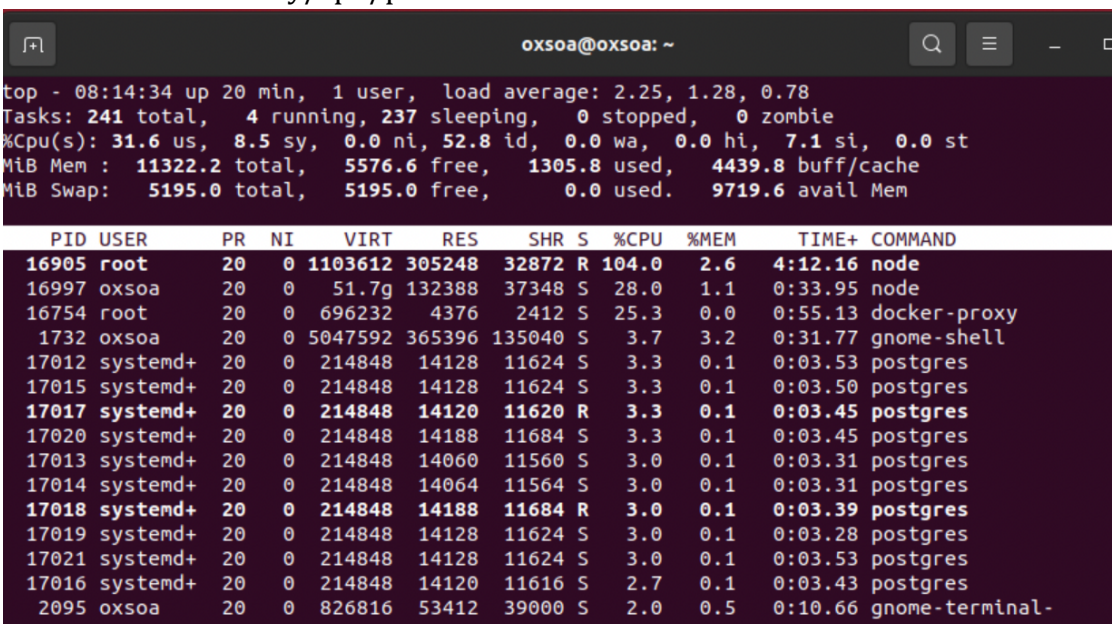
```
Req/Bytes counts sampled once per second.

144k requests in 60.03s, 40.8 MB read
```

I think ~2400 requests per second accessing a database is reasonable. You may get different results on your setup of course. Overall this shows we have built a highly performant (and hopefully scalable) application.

8. While it is running you can monitor the CPU.
   Extend the time to run longer (e.g. 300s) and rerun

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

9. 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

10. You will see that there is are two node processes here.
    One for the autocannon and one for the server

    Node is single-threaded, so on a multi-core system you might want to run more processes for the server. 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

11. 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.

12. That's all for this lab!

Extension

If you want to try a similar benchmarking tool written in go, try:

https://github.com/codesenberg/bombardier

It should be pretty similar to get running.