SDC Engineering Journal and Notes

Database Benchmarking

Which two DBMS did you test?

1. MongoDB / Mongoose
2. PostgreSQL

10/13/2020

Performance notes: (inherited DBMS = mySQL)

Response time: Only Get request

- http://localhost:3003/api/sites/20
    - **Postman**: < 40 ms
    - **Morgan:** < 1.4 ms
- http://localhost:3003/api/services/20
    - **Postman**: < 49 ms
    - **Morgan:** < 1.2 ms
- http://localhost:3003/api/activities/20
    - **Postman**: < 40 ms
    - **Morgan:** < 2.4 ms
- http://localhost:3003/api/attractions/20
    - **Postman**: < 34 ms
    - **Morgan:** < 3 ms
- http://localhost:3003/api/photos/20
    - **Postman**: < 51 ms
    - **Morgan:** < 5 ms
- http://localhost:3003/api/terrain/20
    - **Postman**: < 30 ms
    - **Morgan:** < 1.2 ms

Notes:

Tested all inherited DBMS Read operations multiple times to have a gauge on the response times. Decided to show three different examples of each to also have reference to a variety of response times as well.

Benefits/TradeOffs:

Primary Database MongoDB/Mongoose

Pros:

- MongoDB is a document database in which one collection holds different documents.
- Structure of a single object is clear.
- No complex joins.
- Change-friendly design

Cons:

- Joins not supported
- High memory usage

C.A.R.

Challenges:

- Deciding which DBSM i wanted to choose and figure out why it would be more beneficial.

Actions:

- Choose two DBSM I wanted to use to convert the existing database.
- Chose MongoDB for its easy and document database style and PostgreSQL

Results:

- Implemented some small changes in the database and server.

10/14/2020

Performance notes: (DBMS = MongoDB/Mongoose)

Response time: Only Get request for 100 records

- http://localhost:3003/api/sites/20
  - **Postman**: < 30 ms
  - **Morgan:** < 1.6 ms
- http://localhost:3003/api/services/82
  - **Postman**: < 34 ms
  - **Morgan:** < 2.5 ms
- http://localhost:3003/api/activities/20
  - **Postman**: < 29 ms
  - **Morgan:** < 2.7 ms
- http://localhost:3003/api/terrain/3
  - **Postman**: < 32 ms
  - **Morgan:** < 3.1 ms

C.A.R.

Challenges:

- Setting up the script to generate 10mil data and parse the CSV file so I can seed it into my mongoDB. Attempted to use faker.js to generate this fake data. Tried to use the jquery-csv package to help generate an array of objects to easily seed into my database.
- One challenge I also faced was working with schema's that took in images because they would contain image urls which would end up being quite long to store in a csv file.

Actions:

- Read up on jquery-csv package and continued to try and debug the following errors received. Couldn't create the result of data to allow me to seed into the database.

Results:

- Also tried to generate about 10mil records for my first sites schema and took about 8 minutes to generate all 10mil. Ending up reading a mongoDB import method that allows for direct access to the database and collection I wanted to import my data in. Used the mongoimport method to seed 10mil records inside within 4 minutes.

10/15/2020

Performance notes: (DBMS = MongoDB/Mongoose)

Response time: Only Get request for 10mil records

- http://localhost:3003/api/sites/9999999 (BEFORE INDEXING)
    - "executionTimeMillis" : 5079
- http://localhost:3003/api/sites/9999999 (AFTER INDEXING)
    - "executionTimeMillis" : 0ms

```
db.sites.explain("executionStats").find({id: 9999999})
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "overviewDB.sites",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "id" : {
                                "$eq" : 9999999
                        }
                },
                "winningPlan" : {
                        "stage" : "FETCH",
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "keyPattern" : {
                                        "id" : 1
                                },
                                "indexName" : "id_1",
                                "isMultiKey" : false,
                                "multiKeyPaths" : {
                                        "id" : [ ]
                                },
                                "isUnique" : false,
                                "isSparse" : false,
                                "isPartial" : false,
                                "indexVersion" : 2,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "id" : [
                                                "[9999999.0, 9999999.0]"
                                        ]
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 0,
                "totalKeysExamined" : 1,
                "totalDocsExamined" : 1,
                "executionStages" : {
                        "stage" : "FETCH",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
```

- http://localhost:3003/api/services/9999999 (BEFORE INDEXING)
    - "executionTimeMillis" : 5800
- http://localhost:3003/api/services/9999999 (AFTER INDEXING)
    - "executionTimeMillis" : 9ms

```
> db.services.explain("executionStats").find({ id: { $in: [454545, 7676767] } })
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "overviewDB.services",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "id" : {
                                "$in" : [
                                        454545,
                                        7676767
                                ]
                        }
                },
                "winningPlan" : {
                        "stage" : "FETCH",
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "keyPattern" : {
                                        "id" : 1
                                },
                                "indexName" : "id_1",
                                "isMultiKey" : false,
                                "multiKeyPaths" : {
                                        "id" : [ ]
                                },
                                "isUnique" : false,
                                "isSparse" : false,
                                "isPartial" : false,
                                "indexVersion" : 2,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "id" : [
                                                "[454545.0, 454545.0]",
                                                "[7676767.0, 7676767.0]"
                                        ]
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 2,
                "executionTimeMillis" : 9,
                "totalKeysExamined" : 4,
                "totalDocsExamined" : 2,
                "executionStages" : {
                        "stage" : "FETCH",
                        "nReturned" : 2,
                        "executionTimeMillisEstimate" : 0,
```

- http://localhost:3003/api/activities/9999999 (BEFORE INDEXING)
    - "executionTimeMillis" : 93
- http://localhost:3003/api/activities/9999999 (AFTER INDEXING)
    - "executionTimeMillis" : 10ms

```
db.activities.explain("executionStats").find({ id: { $in: [454545, 7676767, 1919191] } })

      "queryPlanner" : {
              "plannerVersion" : 1,
              "namespace" : "overviewDB.activities",
              "indexFilterSet" : false,
              "parsedQuery" : {
                      "id" : {
                              "$in" : [
                                      454545,
                                      1919191,
                                      7676767
                              ]
                      }
              },
              "winningPlan" : {
                      "stage" : "FETCH",
                      "inputStage" : {
                              "stage" : "IXSCAN",
                              "keyPattern" : {
                                      "id" : 1
                              },
                              "indexName" : "id_1",
                              "isMultiKey" : false,
                              "multiKeyPaths" : {
                                      "id" : [ ]
                              },
                              "isUnique" : false,
                              "isSparse" : false,
                              "isPartial" : false,
                              "indexVersion" : 2,
                              "direction" : "forward",
                              "indexBounds" : {
                                      "id" : [
                                              "[454545.0, 454545.0]",
                                              "[1919191.0, 1919191.0]",
                                              "[7676767.0, 7676767.0]"
                                      ]
                              }
                      }
              },
              "rejectedPlans" : [ ]
      },
      "executionStats" : {
              "executionSuccess" : true,
              "nReturned" : 3,
              "executionTimeMillis" : 10,
              "totalKeysExamined" : 6,
              "totalDocsExamined" : 3,
              "executionStages" : {
                      "stage" : "FETCH",
                      "nReturned" : 3,
                      "executionTimeMillisEstimate" : 0,
```

Notes:

One thing I realized was I was recording the wrong response times since i was benchmarking the API endpoints themselves and not the query times. So now using mongoDB's analyze query performance built in function, this allowed me to see the actual query speeds and record it. Before using indexes for my 10mil data, my query search times were on average around 5000ms which is not as fast as i wanted it to be. But after implementing indexes to my data, it allowed me to reach incredible speeds. The speeds were consistently around 0ms because when you set an index to a collections data, it acts as a key value pair so the search is O(1). In addition to working with 6 schemas I considered combining all the tables into one huge table but realized that it would take too much time to refactor all the tables since it would ultimately affect the client side.

10/16/2020

Benefits/TradeOffs:

Primary Database PostgreSQL

Pros:

- Joins are supported to reduce the amount of redundancy
- Have many different indexing methods
- Schemas have identified relationship with foreign keys

Cons:

- Strict schema

Notes:

Today I worked on only making 3 schemas scripts and tested all the benchmarks and response times. Recorded before indexing and after indexing. Also modified some of the queries to be faster by using the .find method rather than using .aggregate to pick random records. Decided to fix the services script because one of the data entries contained commas and once I would run the mongoimport method on the terminal, it would generate a csv file that contained more headers since every comma would count as a separate value. Thus having to remove the commas and run the script again so my data would resemble the schema. Later I looked into postgreSQL and started researching the pros and cons. Had trouble with postgreSQL syntax and figuring out why I wasn't able to seed a small file to test out the query. Once I tried running the seeding script, I would get an error telling me that the column is not found. Example: (error: column "Sam Lakes" does not exist). Even though "Sam Lakes" is the value I'm trying to store and it is treating it as a column in my table which is confusing. Figured out that using double quotes for postgreSQL to represent that an item is a "string" is not allowed. Using single quotes fixed the issue because using double quotes acts if you are looking up a column not a value.

10/17/2020

Performance notes: (DBMS = PostgreSQL)

Response time: Only Get request for 10mil records

- http://localhost:3003/api/sites/9999999
    - Execution Time: 0.050 ms

```
timcamp=# EXPLAIN ANALYZE SELECT * FROM sites WHERE id = 999999;
                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Index Scan using sites_pkey on sites  (cost=0.43..8.45 rows=1 width=57) (actual time=0.014..0.015 rows=1 loops=1)
   Index Cond: (id = 999999)
 Planning Time: 0.098 ms
 Execution Time: 0.050 ms
(4 rows)
```

- http://localhost:3003/api/services/9999999 (DIDN'T USE SERIAL PRIMARY KEY)
    - Execution Time: 554 ms
- http://localhost:3003/api/services/9999999 (AFTER USING SERIAL PRIMARY KEY)
    - Execution Time: 0.028 ms

```
timcamp=# EXPLAIN ANALYZE SELECT * FROM services WHERE id IN (452519, 235332);
                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Index Scan using services_pkey on services  (cost=0.43..12.90 rows=2 width=229) (actual time=0.014..0.020 rows=2 loops=1)
   Index Cond: (id = ANY ('{452519,235332}'::integer[]))
 Planning Time: 0.064 ms
 Execution Time: 0.031 ms
(4 rows)
```

- http://localhost:3003/api/activities/9999999 (DIDN'T USE SERIAL PRIMARY KEY)
    - Execution Time: 556 ms
- http://localhost:3003/api/activities/9999999 (AFTER USING SERIAL PRIMARY KEY)
    - Execution Time: 0.036ms

```
timcamp=# EXPLAIN ANALYZE SELECT * FROM activities WHERE id IN (100239, 99999);
                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Index Scan using activities_pkey on activities  (cost=0.43..12.90 rows=2 width=83) (actual time=0.020..0.023 rows=2 loops=1)
   Index Cond: (id = ANY ('{100239,99999}'::integer[]))
 Planning Time: 0.046 ms
 Execution Time: 0.032 ms
(4 rows)
```

Notes:

Today I managed to import 10mil data from a CSV file into my postgreSQL database. After figuring out that postgreSQL has a really strict schema and query syntax I finally imported the data. It was interesting to find that postgreSQL doesn't like camel casing and would lowercase all camel cased tables. I was able to benchmark test the query I used for the sites table and it was relatively fast. Another thing I realized was that postgreSQL has some default indexing I believe? The search time for an id is almost similar to mongoDB search time after indexing but mongoDB seems a bit faster. Also with mongoDB the id's I had to manually add was not sorted in the database so an id with 9999 can be the first item in the database so I couldn't really tell if I was searching "deep" or in the "last 10 percent" range of the database. I also managed to test a record search in a 10mil dataset without using the SERIAL PRIMARY KEY for the tables schema and found that it was much slower to make your own id and increment.

Example of inside the sites table in the postgreSQL shell:

Indexes: "sites_pkey" PRIMARY KEY, btree (id)

A: Specifying a primary key or a unique within a CREATE TABLE statement causes PostgreSQL to create B-Tree indexes.

10/18/2020

Notes:

One thing I noticed when working with postgreSQL is that it is extremely fast when importing a data file of 10 mil records. It only took around 30 seconds or less to finish all 3 imports and of sizes of 1.1GB ~ 2.7GB average. Photo example of size for PostgreSQL:

```
timcamp=# SELECT pg_size_pretty( pg_total_relation_size('sites') );
 pg_size_pretty
----------------
 1103 MB
(1 row)

timcamp=# SELECT pg_size_pretty( pg_total_relation_size('services') );
 pg_size_pretty
----------------
 2751 MB
(1 row)

timcamp=# SELECT pg_size_pretty( pg_total_relation_size('activities') );
 pg_size_pretty
----------------
 1331 MB
(1 row)
```

MongoDB would take a rather extensive time to import all the data into the collection. Once retrieving all the response times for 3 routes for both DBMS, I compared the times and really saw no difference other than postgreSQL is a tad bit slower then mongoDB (after indexing). Furthermore, I decided to use postgreSQL since it has the ability to set a primary key (id), which allows for query for an id much easier since mongoDB while it does have a default id for every record, it's in a huge string. In addition, I liked the import speeds when working with postgreSQL and the default indexing when using a serial primary key. I believe it is using a default B-tree indexing to speed up id search times. Then I tested the response times between a premade serial primary key id (~0.030ms) and just a regular id integer (550.00ms), which was a huge difference in response time.

C.A.R.

Challenges:

- One challenge I faced was that I think I was recording the wrong response times because once you make an initial query it takes about a couple milliseconds to query and if you query the exact same thing then it is super fast. My hypothesis is that it caches it and saves it in memory so the next time the same query is made it delivers instantly. I was recording the times when it already stored the query search in memory.

Actions:

- So I decided to go back and record the response times again so that my metrics aren't incorrect or misleading.

Results:

- Now that I have the correct data I can now correctly compare which DBMS I want to choose.

10/21/2020

Performance notes:

Stress tested with artillery with 100 requests to just test it out first!

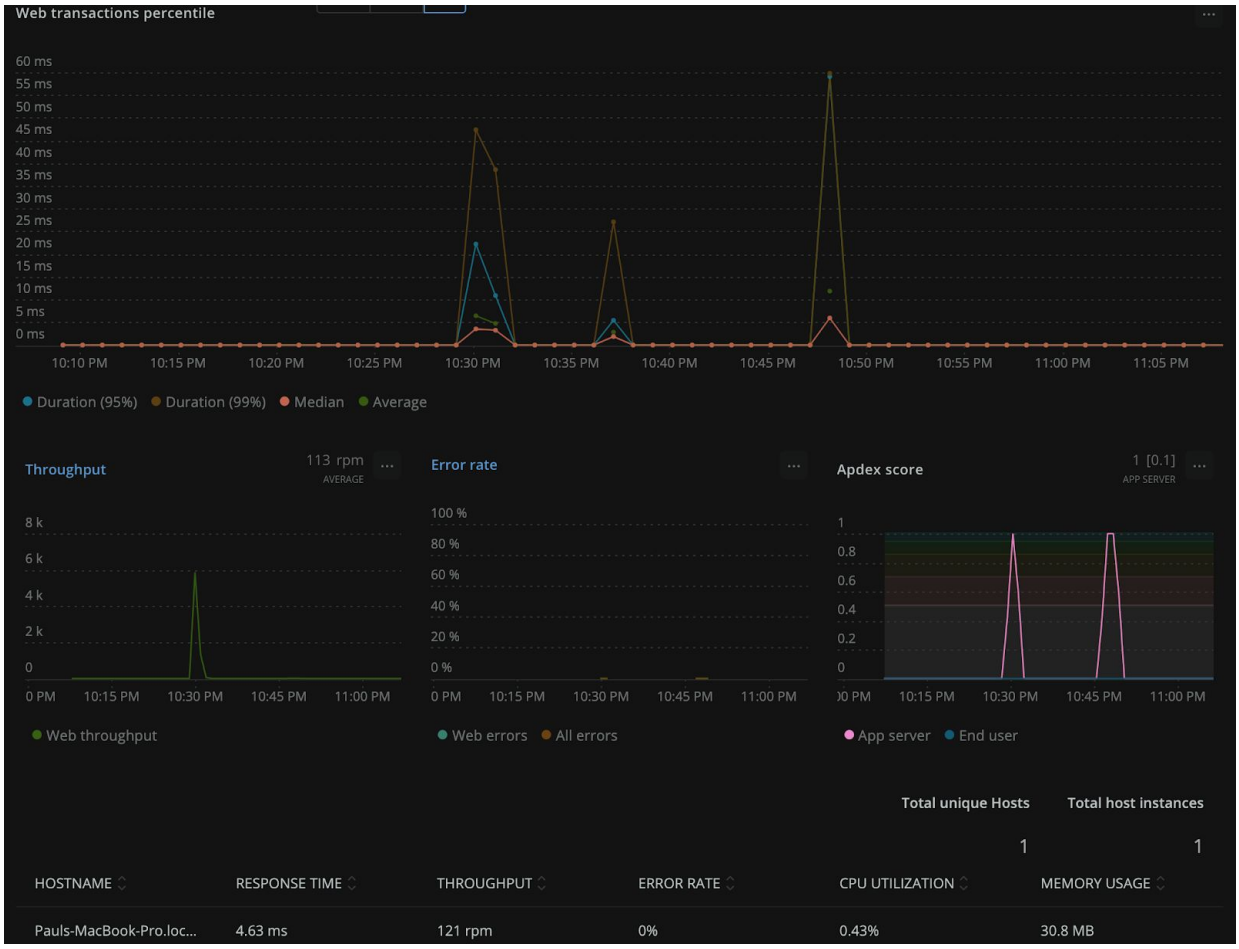
```
Pauls-MacBook-Pro:SDC-Overview paulchoi$ artillery quick --count 1 -n 100 http://localhost:3003
/api/sites/1000
Started phase 0, duration: 1s @ 12:15:30(-0700) 2020-10-21
Report @ 12:15:32(-0700) 2020-10-21
Elapsed time: 1 second
  Scenarios launched:  1
  Scenarios completed: 1
  Requests completed:  100
  Mean response/sec: 212.77
  Response time (msec):
    min: 1.2
    max: 28.4
    median: 1.7
    p95: 2.7
    p99: 15.8
  Codes:
    200: 100

All virtual users finished
Summary report @ 12:15:32(-0700) 2020-10-21
  Scenarios launched:  1
  Scenarios completed: 1
  Requests completed:  100
  Mean response/sec: 208.33
  Response time (msec):
    min: 1.2
    max: 28.4
    median: 1.7
    p95: 2.7
    p99: 15.8
  Scenario counts:
    0: 1 (100%)
  Codes:
    200: 100
```

Then proceeded to test with 1000 requests to see the speed of the response times.

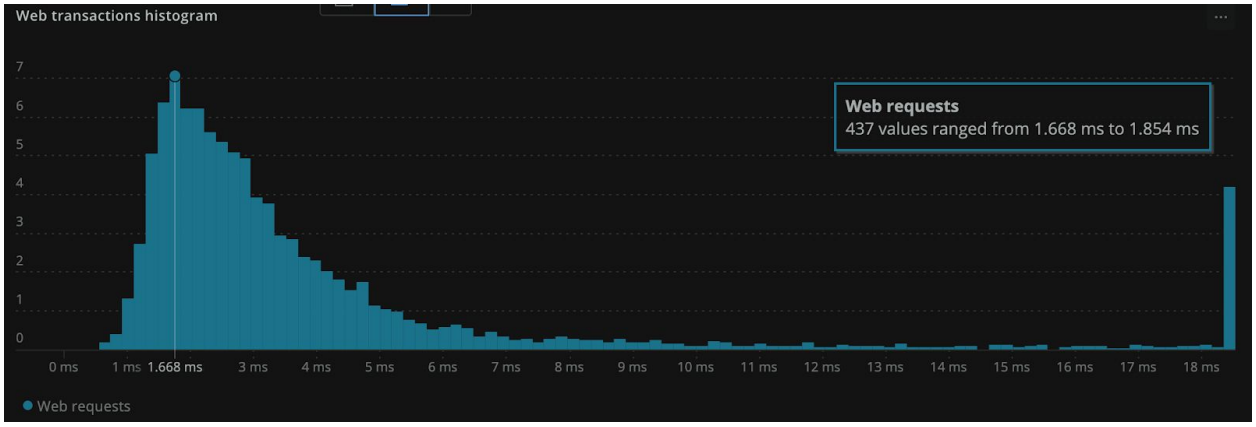
```
Pauls-MacBook-Pro:SDC-Overview paulchoi$ artillery quick --count 1 -n 1000 http://localhost:300
3/api/sites/1000
Started phase 0, duration: 1s @ 12:19:31(-0700) 2020-10-21
Report @ 12:19:36(-0700) 2020-10-21
Elapsed time: 5 seconds
  Scenarios launched:  1
  Scenarios completed: 1
  Requests completed:  1000
  Mean response/sec: 250.63
  Response time (msec):
    min: 1
    max: 12.7
    median: 1.7
    p95: 2.9
    p99: 3.9
  Codes:
    200: 1000

All virtual users finished
Summary report @ 12:19:36(-0700) 2020-10-21
  Scenarios launched:  1
  Scenarios completed: 1
  Requests completed:  1000
  Mean response/sec: 250
  Response time (msec):
    min: 1
    max: 12.7
    median: 1.7
    p95: 2.9
    p99: 3.9
  Scenario counts:
    0: 1 (100%)
  Codes:
    200: 1000
```

Notes:

While trying to figure out how to use Loader.io in the beginning it was difficult setting up the “Target Host” and it seems a little vague on how to set it up properly. So I opted for Artillery.io and used a simple terminal command that allows me to stress test a specific endpoint given as well the amount of requests. In addition, I finally got New Relic set up so that I will be able to retrieve some data once I run an Artillery stress test. Here are the results of that data:



I wasn't able to really make sense of all the data at the moment but going to research more about throughput and the metrics in more detail.



10/22/2020

Notes:

Today I learned that while working with K6 you can set different options and stages to simulate different scenarios your application will be in. So I decided to first stress test my application by setting a few options and without any stages involved:

```
vus: 1000,
 rps: 1000,
 duration: '1m',
```

Vus: 1000 = 1000 virtual users

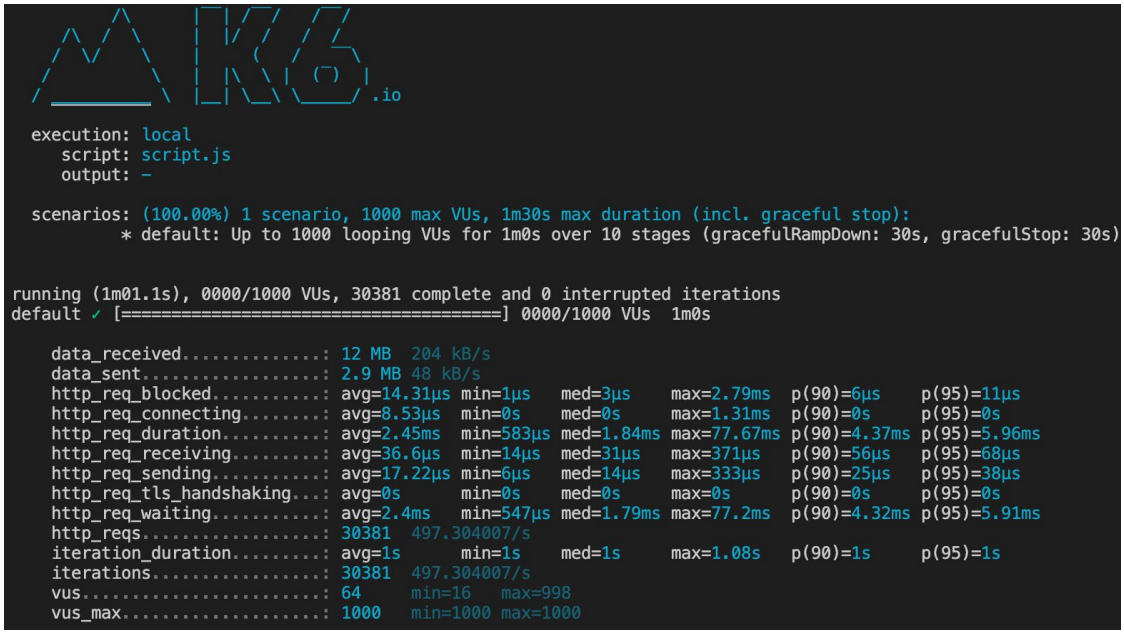
Rps: 1000 = 1000 requests per second

Duration: '1m' = Run for 1 minute.

Data retrieved back from K6:

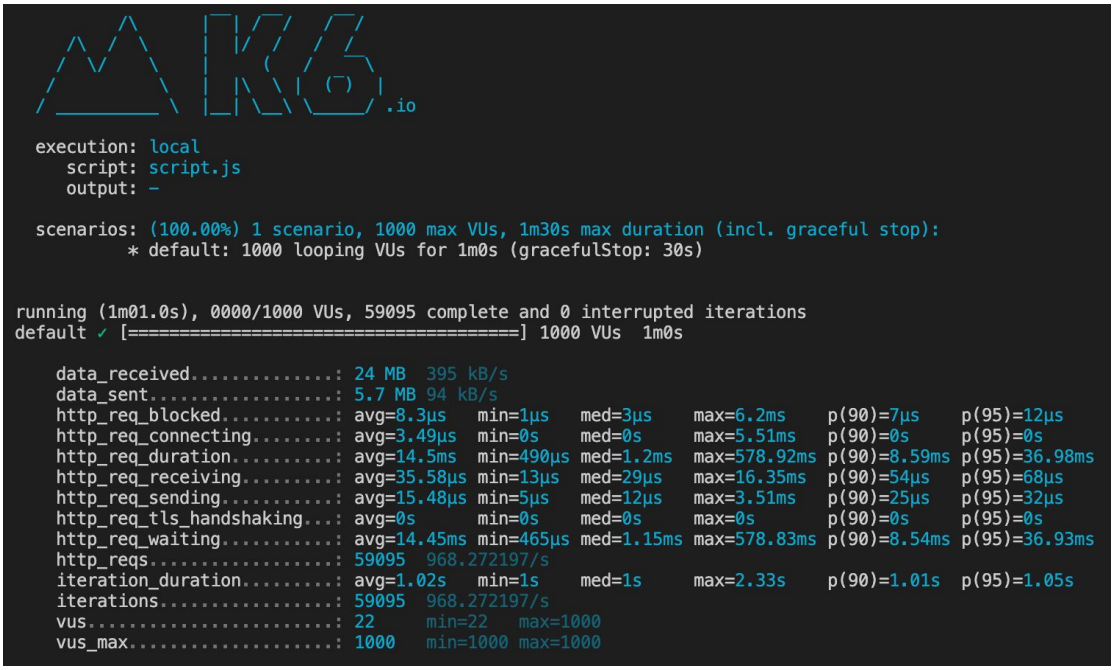
```
          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: script.js
     output: -

  scenarios: (100.00%) 1 scenario, 1000 max VUs, 1m30s max duration (incl. graceful stop):
           * default: 1000 looping VUs for 1m0s (gracefulStop: 30s)


running (1m01.1s), 0000/1000 VUs, 58717 complete and 0 interrupted iterations
default ✓ [======================================] 1000 VUs  1m0s

     data_received..............: 24 MB  388 kB/s
     data_sent..................: 5.4 MB 88 kB/s
     http_req_blocked...........: avg=10.43µs min=1µs   med=4µs    max=5.58ms   p(90)=10µs    p(95)=16µs
     http_req_connecting........: avg=4.26µs  min=0s    med=0s     max=4.51ms   p(90)=0s      p(95)=0s
     http_req_duration..........: avg=11.98ms min=536µs med=1.31ms max=427.26ms p(90)=11.21ms p(95)=23.83ms
     http_req_receiving.........: avg=43.6µs  min=16µs  med=36µs   max=684µs    p(90)=70µs    p(95)=87µs
     http_req_sending...........: avg=19.93µs min=6µs   med=15µs   max=1.37ms   p(90)=34µs    p(95)=45µs
     http_req_tls_handshaking...: avg=0s      min=0s    med=0s     max=0s       p(90)=0s      p(95)=0s
     http_req_waiting...........: avg=11.92ms min=491µs med=1.25ms max=427.11ms p(90)=11.14ms p(95)=23.79ms
     http_reqs..................: 58717  961.463202/s
     iteration_duration.........: avg=1.03s   min=1s    med=1.01s  max=2.36s    p(90)=1.03s   p(95)=1.05s
     iterations.................: 58717  961.463202/s
     vus........................: 126    min=126  max=1000
     vus_max....................: 1000   min=1000 max=1000
```

One thing I realized with doing this and not using stages is that for a whole minute, the application is going through a stress test with 1000 virtual users and 1000 requests every 1 second. But when I stress test with stages included, so within 10 stages for every 6 seconds it ramps up 100 vus and 100 rps. This gave back around 500 rps back. **For example:**



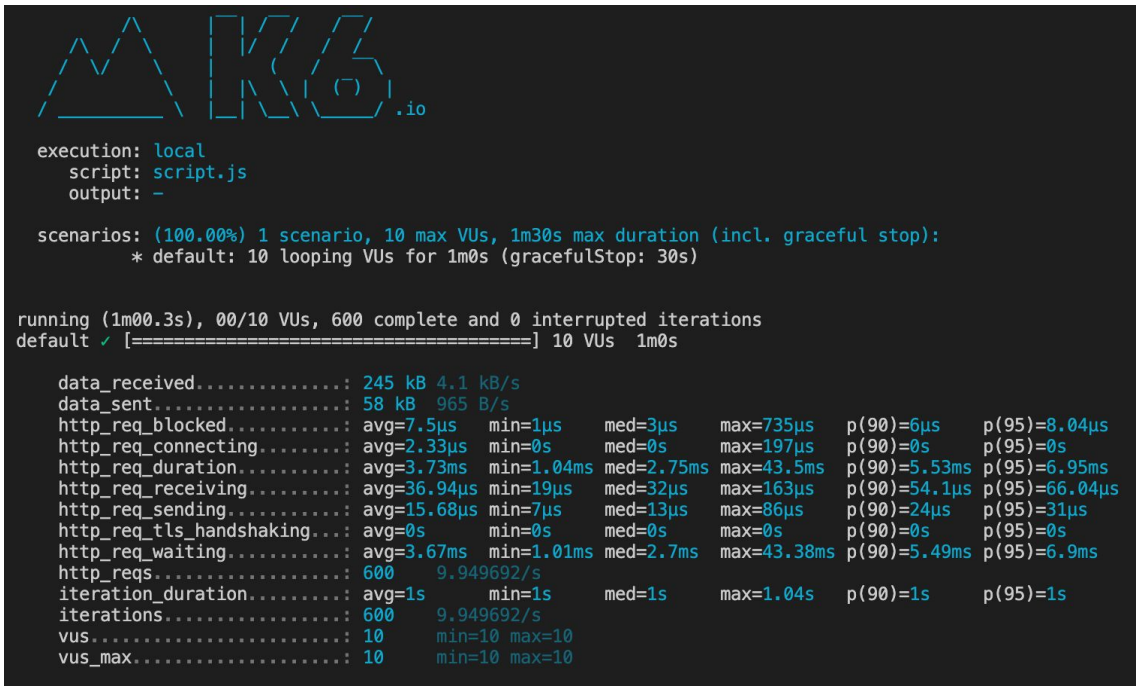
```
  execution: local
     script: script.js
     output: -

  scenarios: (100.00%) 1 scenario, 1000 max VUs, 1m30s max duration (incl. graceful stop):
           * default: Up to 1000 looping VUs for 1m0s over 10 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (1m01.1s), 0000/1000 VUs, 30381 complete and 0 interrupted iterations
default ✓ [======================================] 0000/1000 VUs  1m0s

     data_received..............: 12 MB  204 kB/s
     data_sent..................: 2.9 MB 48 kB/s
     http_req_blocked...........: avg=14.31µs min=1µs   med=3µs    max=2.79ms  p(90)=6µs    p(95)=11µs
     http_req_connecting........: avg=8.53µs  min=0s    med=0s     max=1.31ms  p(90)=0s     p(95)=0s
     http_req_duration..........: avg=2.45ms  min=583µs med=1.84ms max=77.67ms p(90)=4.37ms p(95)=5.96ms
     http_req_receiving.........: avg=36.6µs  min=14µs  med=31µs   max=371µs   p(90)=56µs   p(95)=68µs
     http_req_sending...........: avg=17.22µs min=6µs   med=14µs   max=333µs   p(90)=25µs   p(95)=38µs
     http_req_tls_handshaking...: avg=0s      min=0s    med=0s     max=0s      p(90)=0s     p(95)=0s
     http_req_waiting...........: avg=2.4ms   min=547µs med=1.79ms max=77.2ms  p(90)=4.32ms p(95)=5.91ms
     http_reqs..................: 30381  497.304007/s
     iteration_duration.........: avg=1s      min=1s    med=1s     max=1.08s   p(90)=1s     p(95)=1s
     iterations.................: 30381  497.304007/s
     vus........................: 64     min=16   max=998
     vus_max....................: 1000   min=1000 max=1000
```

Another bottleneck I believe I found was when I tried to increase the amount of rps (request per second) in the options for K6, my theory was that I was going to hit more than my average 960~970 rps because I was increasing the number of requests so I thought it was parallel to the amount I was going to receive. **For example:**



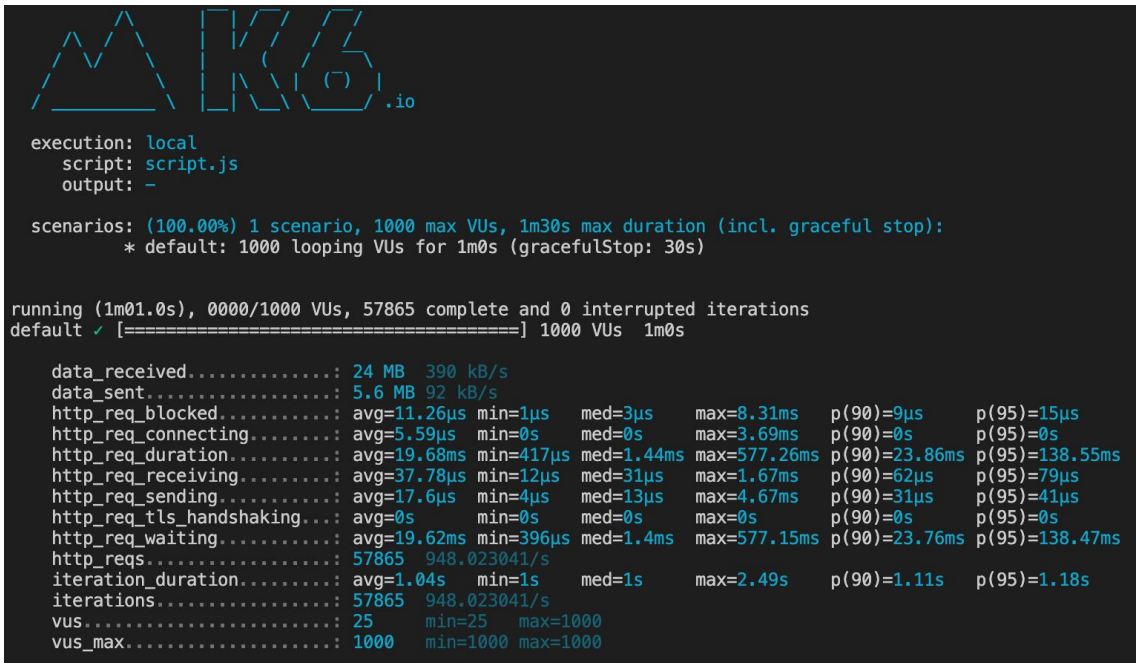
```
  execution: local
     script: script.js
     output: -

  scenarios: (100.00%) 1 scenario, 1000 max VUs, 1m30s max duration (incl. graceful stop):
           * default: 1000 looping VUs for 1m0s (gracefulStop: 30s)


running (1m01.0s), 0000/1000 VUs, 59095 complete and 0 interrupted iterations
default ✓ [======================================] 1000 VUs  1m0s

     data_received..............: 24 MB  395 kB/s
     data_sent..................: 5.7 MB 94 kB/s
     http_req_blocked...........: avg=8.3µs   min=1µs   med=3µs    max=6.2ms    p(90)=7µs    p(95)=12µs
     http_req_connecting........: avg=3.49µs  min=0s    med=0s     max=5.51ms   p(90)=0s     p(95)=0s
     http_req_duration..........: avg=14.5ms  min=490µs med=1.2ms  max=578.92ms p(90)=8.59ms p(95)=36.98ms
     http_req_receiving.........: avg=35.58µs min=13µs  med=29µs   max=16.35ms  p(90)=54µs   p(95)=68µs
     http_req_sending...........: avg=15.48µs min=5µs   med=12µs   max=3.51ms   p(90)=25µs   p(95)=32µs
     http_req_tls_handshaking...: avg=0s      min=0s    med=0s     max=0s       p(90)=0s     p(95)=0s
     http_req_waiting...........: avg=14.45ms min=465µs med=1.15ms max=578.83ms p(90)=8.54ms p(95)=36.93ms
     http_reqs..................: 59095  968.272197/s
     iteration_duration.........: avg=1.02s   min=1s    med=1s     max=2.33s    p(90)=1.01s  p(95)=1.05s
     iterations.................: 59095  968.272197/s
     vus........................: 22     min=22   max=1000
     vus_max....................: 1000   min=1000 max=1000
```

In addition to trying to reach over 1000 requests per second I realized that when setting the number of vus (virtual users), that number really changed the requests per second. So for example if I used only 10 vus for an rps of 1000, I would receive a max of 9.9 requests per second. **For example:**



```
  execution: local
     script: script.js
     output: -

  scenarios: (100.00%) 1 scenario, 10 max VUs, 1m30s max duration (incl. graceful stop):
           * default: 10 looping VUs for 1m0s (gracefulStop: 30s)


running (1m00.3s), 00/10 VUs, 600 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  1m0s

     data_received..............: 245 kB 4.1 kB/s
     data_sent..................: 58 kB  965 B/s
     http_req_blocked...........: avg=7.5µs   min=1µs    med=3µs    max=735µs   p(90)=6µs    p(95)=8.04µs
     http_req_connecting........: avg=2.33µs  min=0s     med=0s     max=197µs   p(90)=0s     p(95)=0s
     http_req_duration..........: avg=3.73ms  min=1.04ms med=2.75ms max=43.5ms  p(90)=5.53ms p(95)=6.95ms
     http_req_receiving.........: avg=36.94µs min=19µs   med=32µs   max=163µs   p(90)=54.1µs p(95)=66.04µs
     http_req_sending...........: avg=15.68µs min=7µs    med=13µs   max=86µs    p(90)=24µs   p(95)=31µs
     http_req_tls_handshaking...: avg=0s      min=0s     med=0s     max=0s      p(90)=0s     p(95)=0s
     http_req_waiting...........: avg=3.67ms  min=1.01ms med=2.7ms  max=43.38ms p(90)=5.49ms p(95)=6.9ms
     http_reqs..................: 600    9.949692/s
     iteration_duration.........: avg=1s      min=1s     med=1s     max=1.04s   p(90)=1s     p(95)=1s
     iterations.................: 600    9.949692/s
     vus........................: 10     min=10 max=10
     vus_max....................: 10     min=10 max=10
```

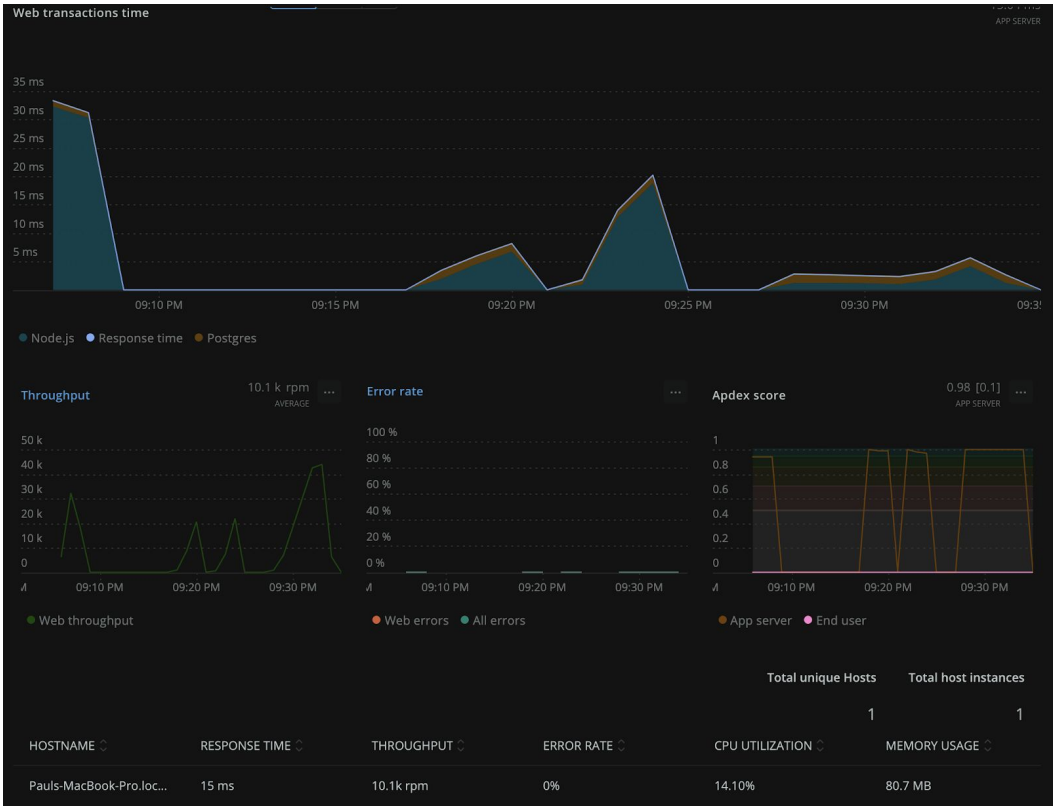
Finally I decided to make a random number function that would generate a random number from 1-10mil so during my stress test I can include this random number at the endpoint api route. Because before I would only test a single endpoint id and I was curious on the differences if the id was always a random id. **For example:**



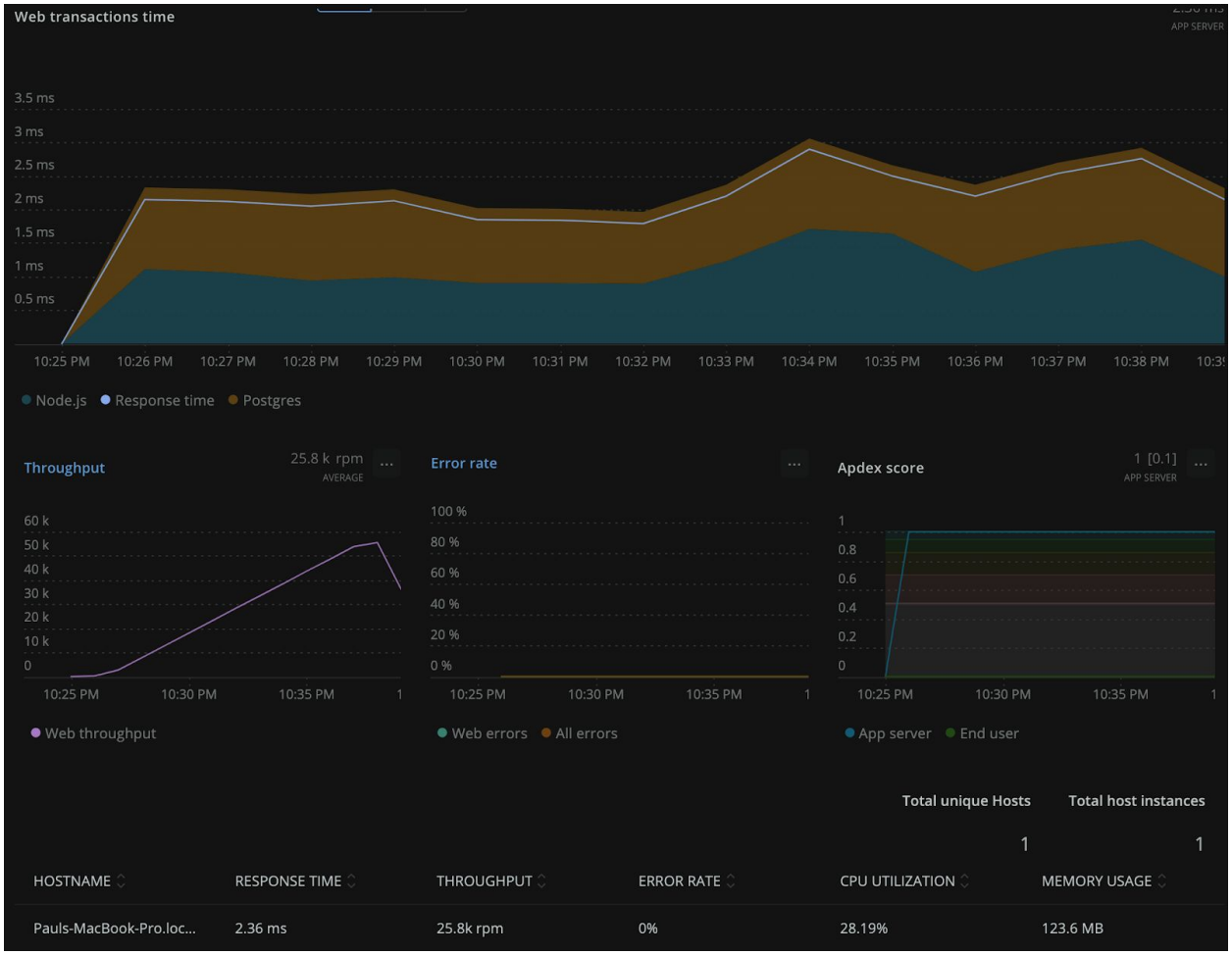
```
  execution: local
     script: script.js
     output: -

  scenarios: (100.00%) 1 scenario, 1000 max VUs, 1m30s max duration (incl. graceful stop):
           * default: 1000 looping VUs for 1m0s (gracefulStop: 30s)


running (1m01.0s), 0000/1000 VUs, 57865 complete and 0 interrupted iterations
default ✓ [======================================] 1000 VUs  1m0s

     data_received..............: 24 MB  390 kB/s
     data_sent..................: 5.6 MB 92 kB/s
     http_req_blocked...........: avg=11.26µs min=1µs   med=3µs    max=8.31ms   p(90)=9µs     p(95)=15µs
     http_req_connecting........: avg=5.59µs  min=0s    med=0s     max=3.69ms   p(90)=0s      p(95)=0s
     http_req_duration..........: avg=19.68ms min=417µs med=1.44ms max=577.26ms p(90)=23.86ms p(95)=138.55ms
     http_req_receiving.........: avg=37.78µs min=12µs  med=31µs   max=1.67ms   p(90)=62µs    p(95)=79µs
     http_req_sending...........: avg=17.6µs  min=4µs   med=13µs   max=4.67ms   p(90)=31µs    p(95)=41µs
     http_req_tls_handshaking...: avg=0s      min=0s    med=0s     max=0s       p(90)=0s      p(95)=0s
     http_req_waiting...........: avg=19.62ms min=396µs med=1.4ms  max=577.15ms p(90)=23.76ms p(95)=138.47ms
     http_reqs..................: 57865  948.023041/s
     iteration_duration.........: avg=1.04s   min=1s    med=1s     max=2.49s    p(90)=1.11s   p(95)=1.18s
     iterations.................: 57865  948.023041/s
     vus........................: 25     min=25   max=1000
     vus_max....................: 1000   min=1000 max=1000
```

K6 (6 second stages) → Throughput, Response time (latency), error rate

K6 (30second stages)→ Throughput, Response time (latency), error rate



K6 (1min 10second stages)→ Throughput, Response time (latency), error rate



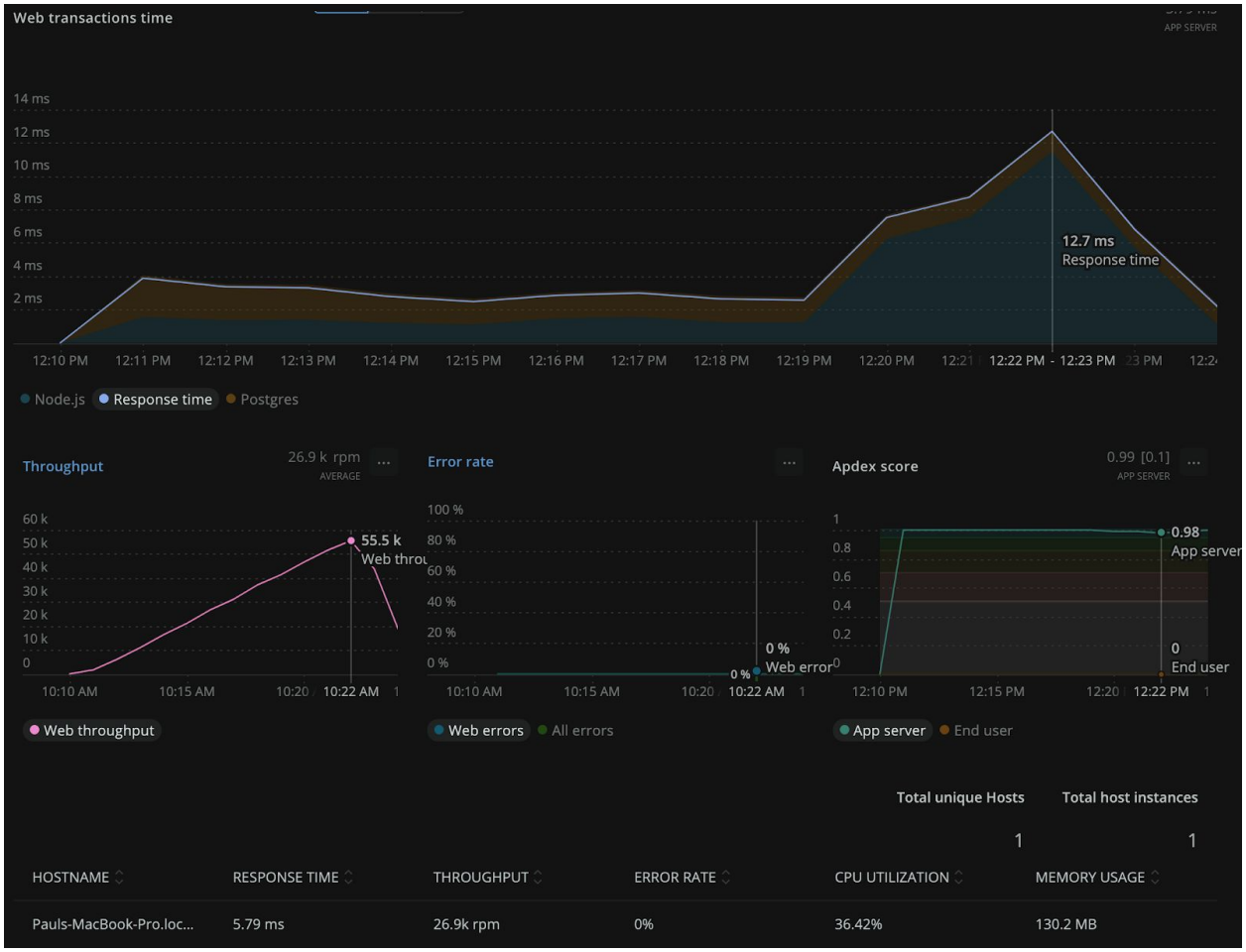
Notes:

One thing I realized was once I increased the duration during each stage in my K6 stress test, it allowed the throughput to increase in size. I was making an educated guess that if we allow more time during each stage that it gives more time for a response to get back.

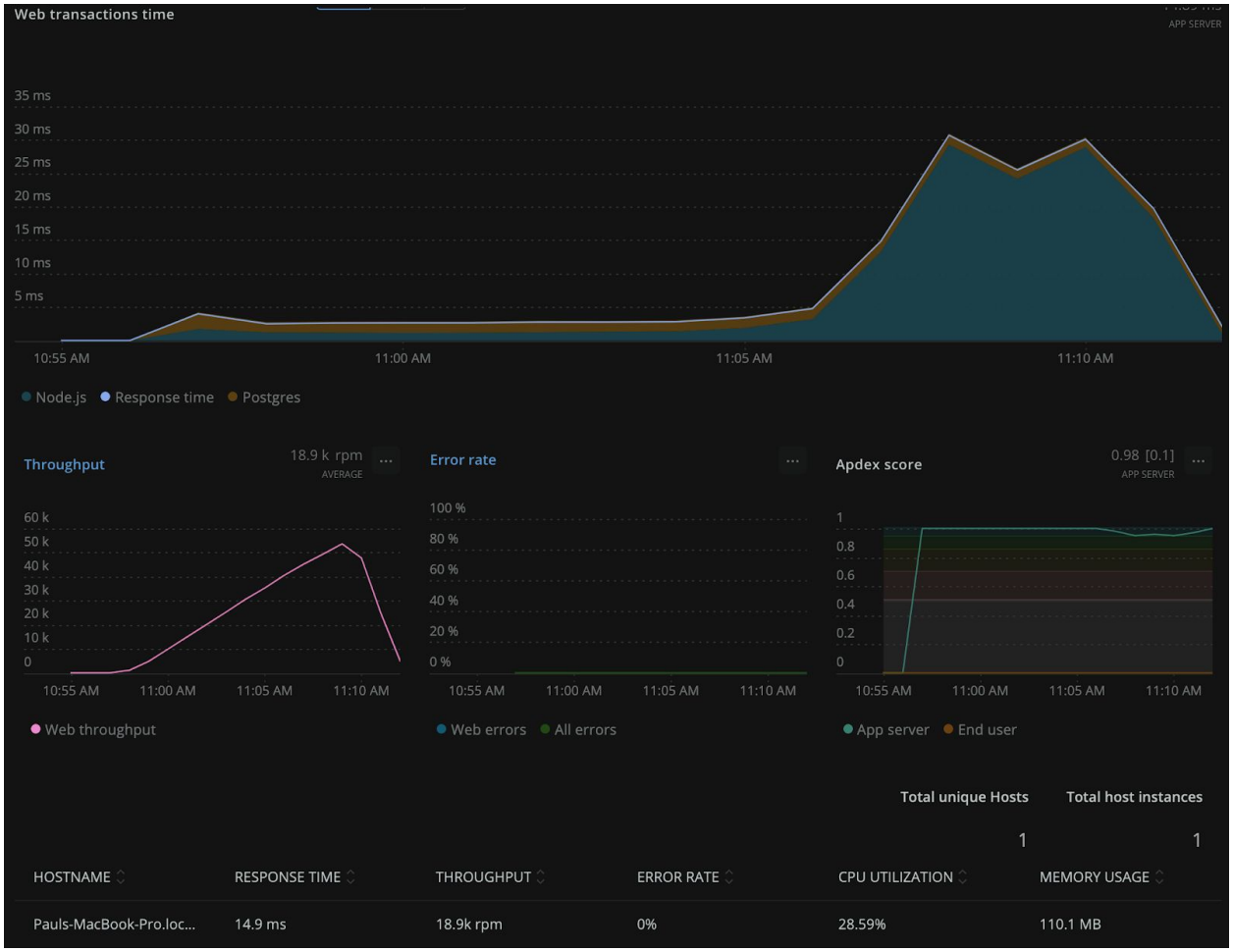
10/23/2020

Notes:

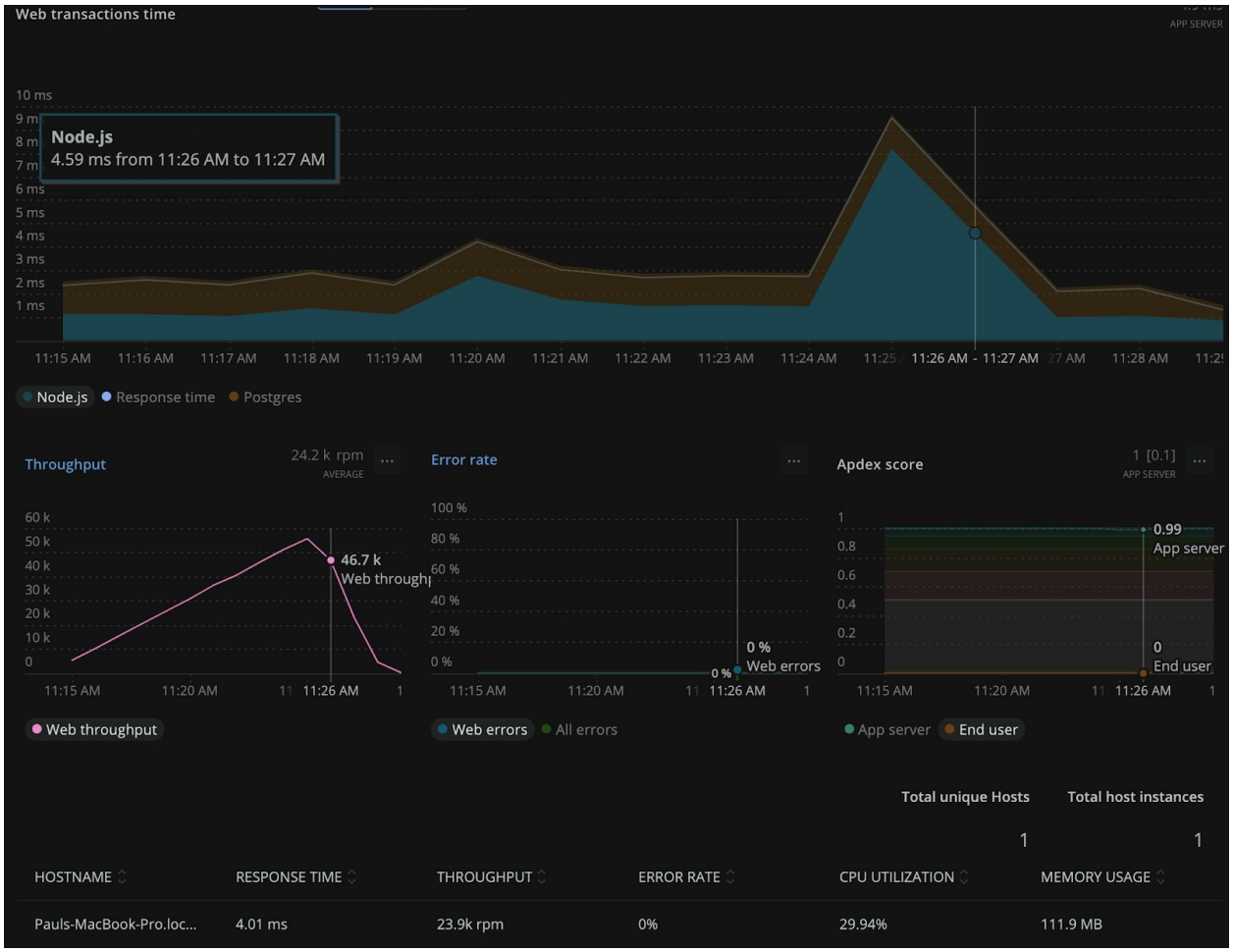
Today I wanted to see if I can optimize my response times to be under 10ms since during my K6 stress testing, it sometimes spikes around 10ms. Here are the results before any optimization:



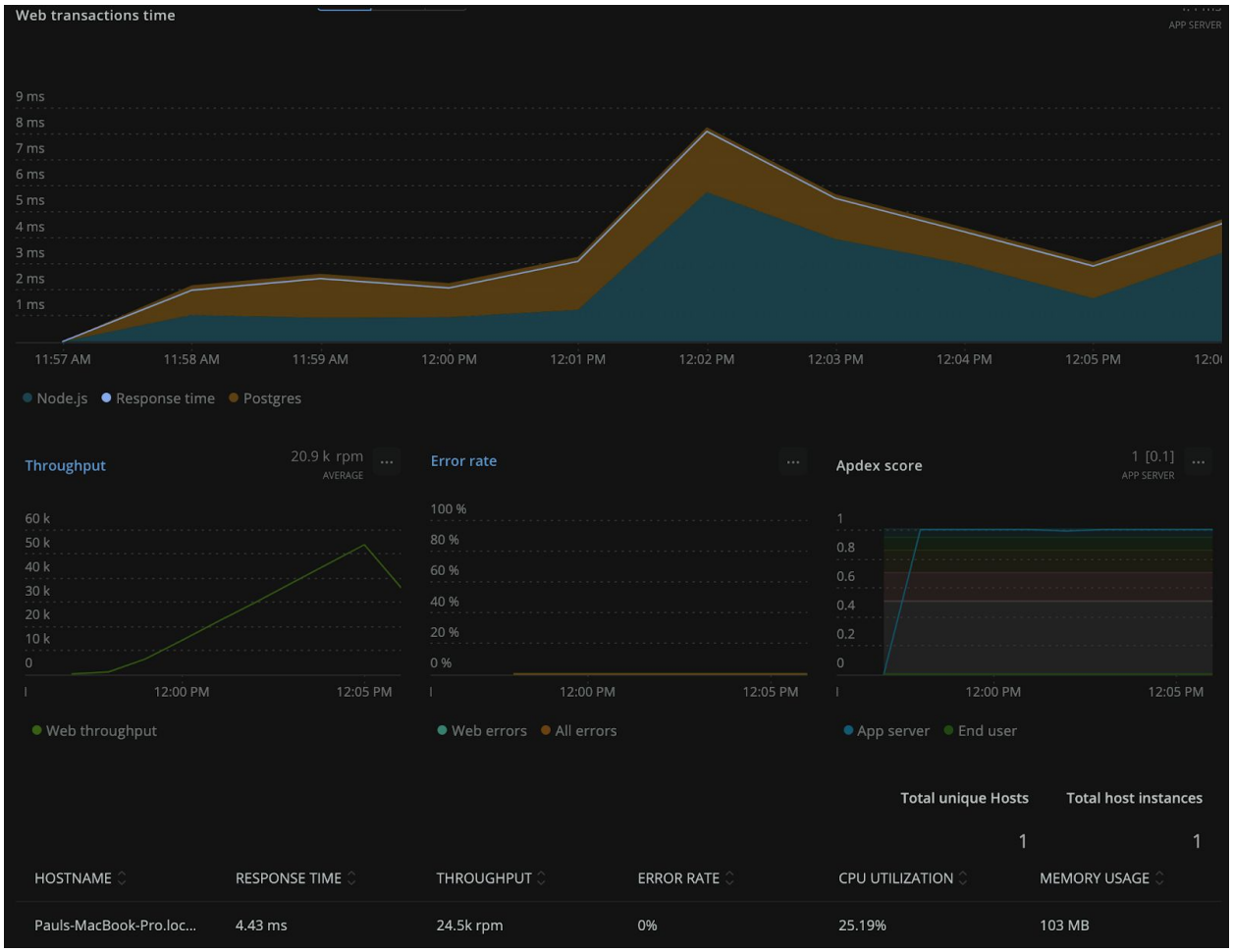
I then tried to switch my GET request to be asynchronous functions and use promises to see if that would help the speed of my node response times but ended up hurting my response times and getting longer times. Here are the results:



So after changing back to my original non-promise GET request, I was able to lower the response times by around 20 ms. Another thing I realized was that once I hit around 50k throughput, my node response times spike dramatically from 1.5ms average to 8-10ms average. I believe this is because there are too many requests and maybe it can't handle that many requests at once. Here are the results of the non-promise GET requests:



One last stress test I ran was changing the durations of each stage to be lower times and saw that it helped with the spike and lowered the node.js response time by almost half:

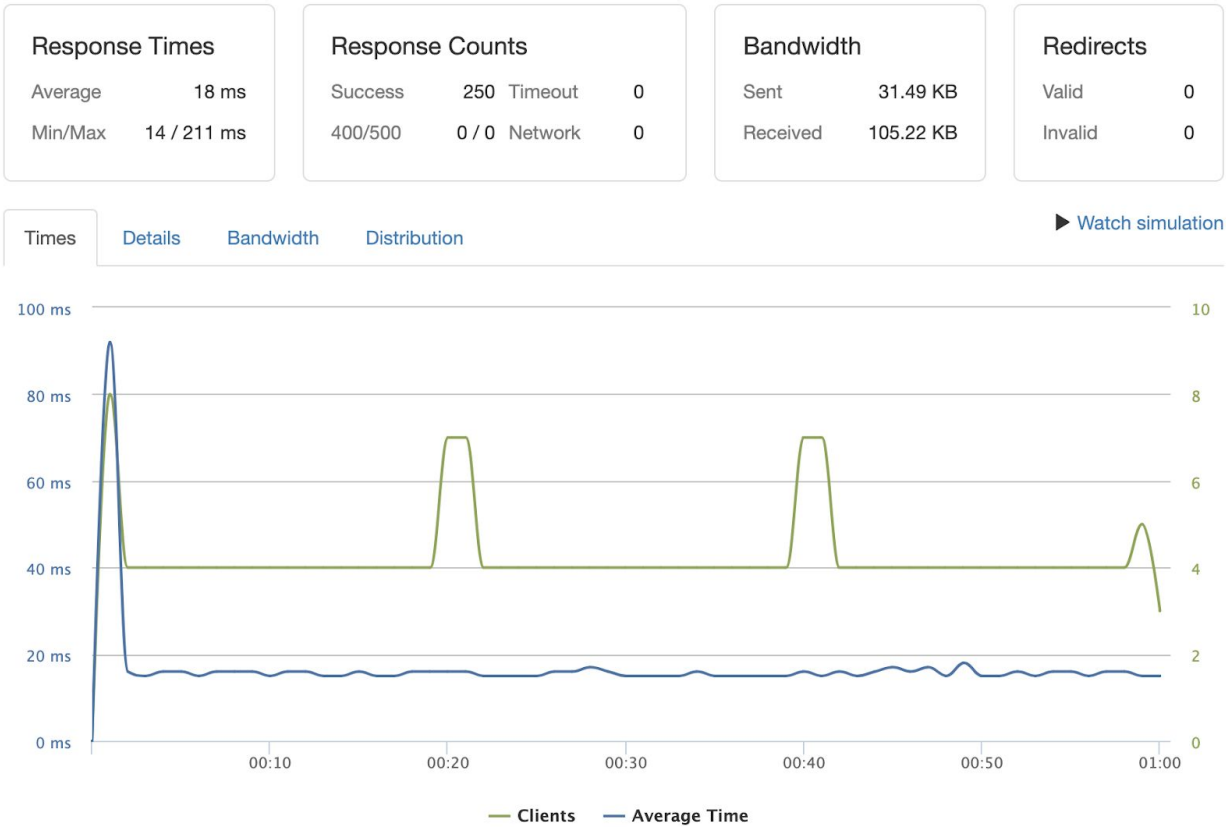


Notes:

Still trying to figure out why node.js spikes when it reaches the 10th stage of 1000 targets.

10/26/2020

Notes:
Today I worked on setting up the connection from my service instance to my database instance and was able to link them together. It was challenging to configure all the needed requirements such as opening new ports for postgreSQL so that it will be able to listen for other requests other than just localhost requests. Another challenge was seeing how I initially really didn't even need my database connection inside my database instance because I would be making that connection directly from my service instance. I had a twisted thought that the database was going to reach out and connect to my service instance. Also was able to run one test on loader.io on my service instance but wasn't really sure what the metrics exactly meant. Still need to research and ask more questions on what the metrics are on loader.io. Here are the results:



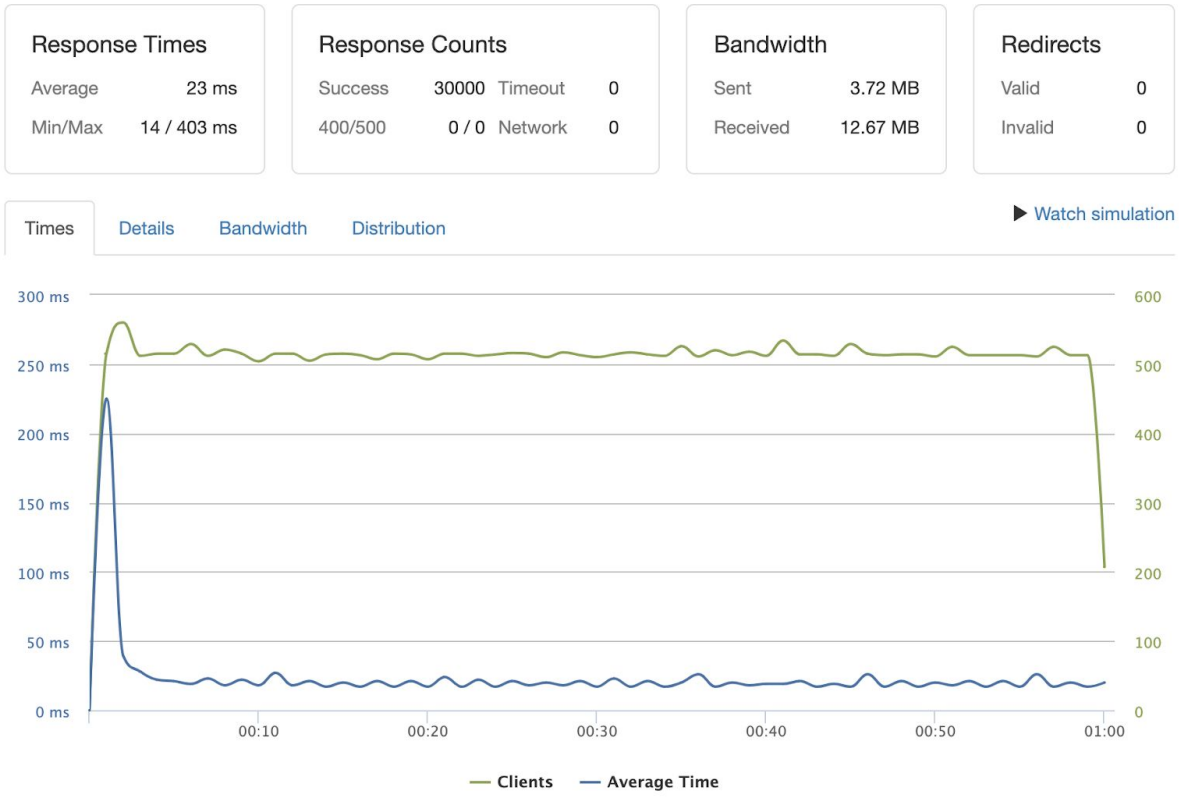
10/27/2020

Notes:

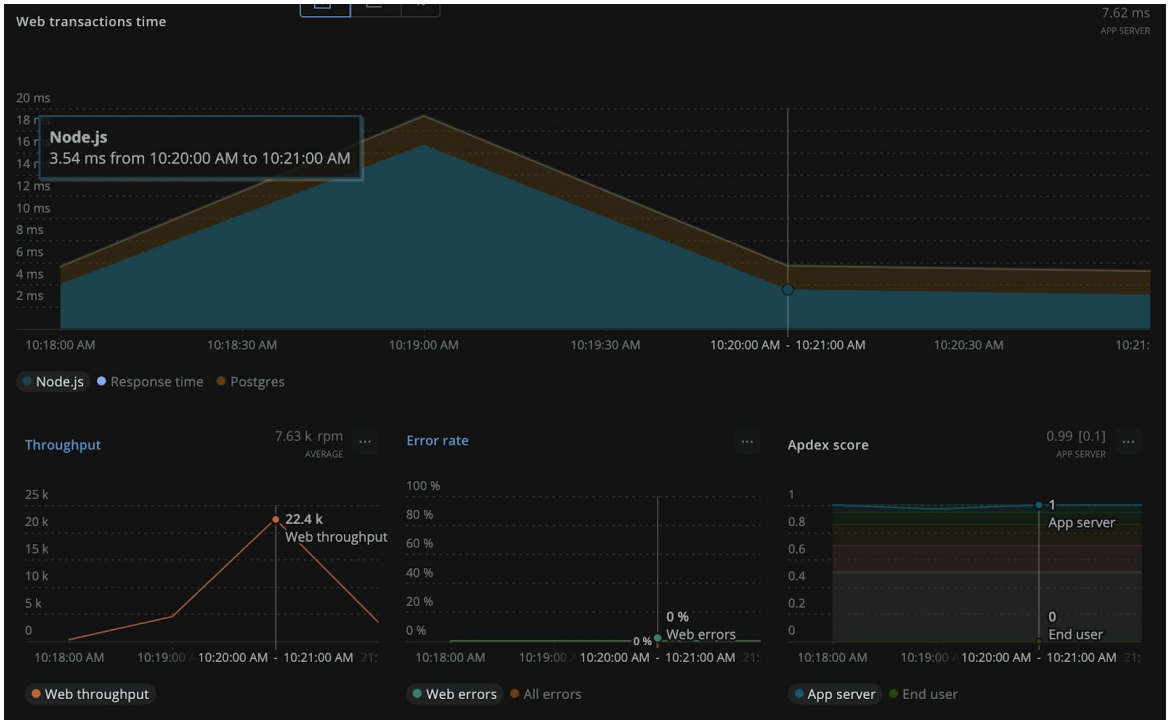
I realized that yesterday I was testing with loader.io and was only testing 250 clients per test which capped the amount of users for the entire test.

I decided to stress test with 500 clients next to hopefully see more results. I was overall getting good results but I would always see a spike in the beginning stages of the climb.

Here are the results: 500 clients per second

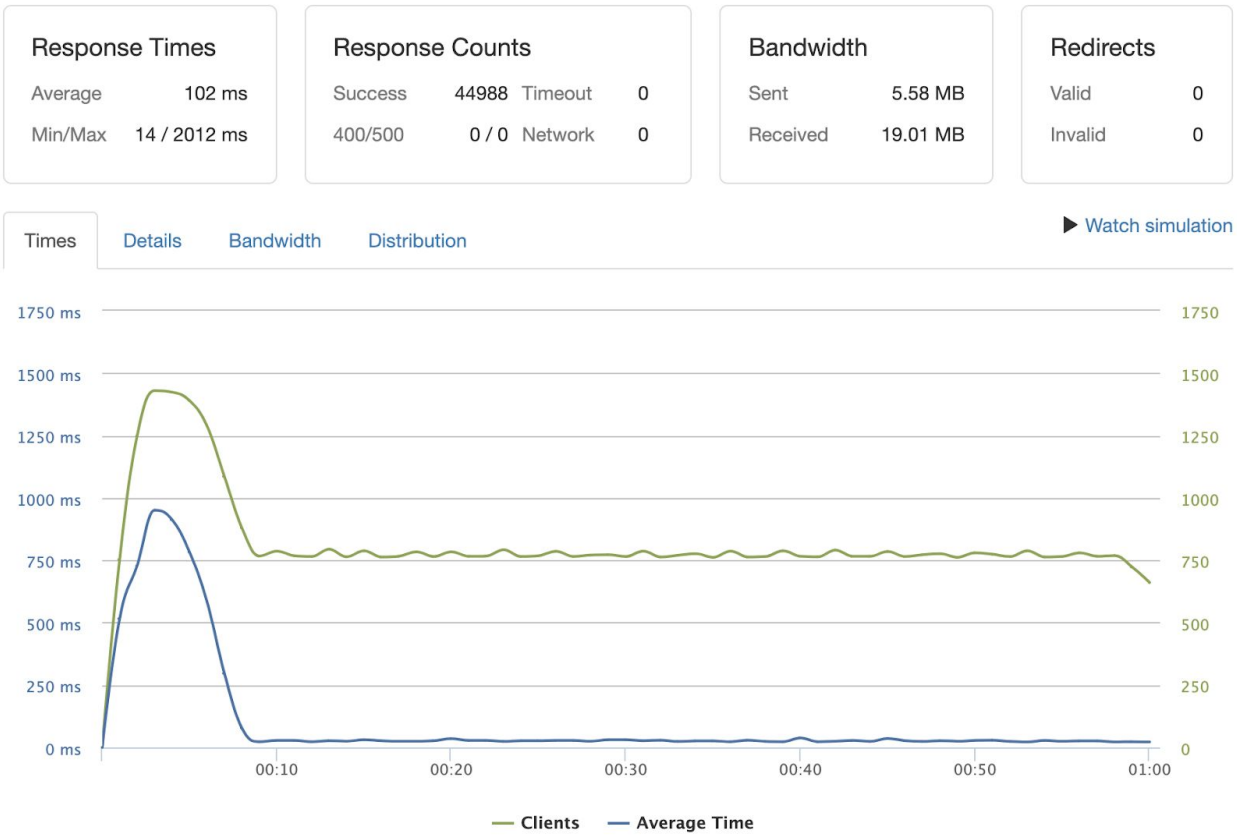


Here are the results of the new relic dashboard: 500 clients per second



In addition I stress tested with 750 clients per second for 1 minute so I can mimic the act of staging within the service to see any differences when increasing the number of clients.

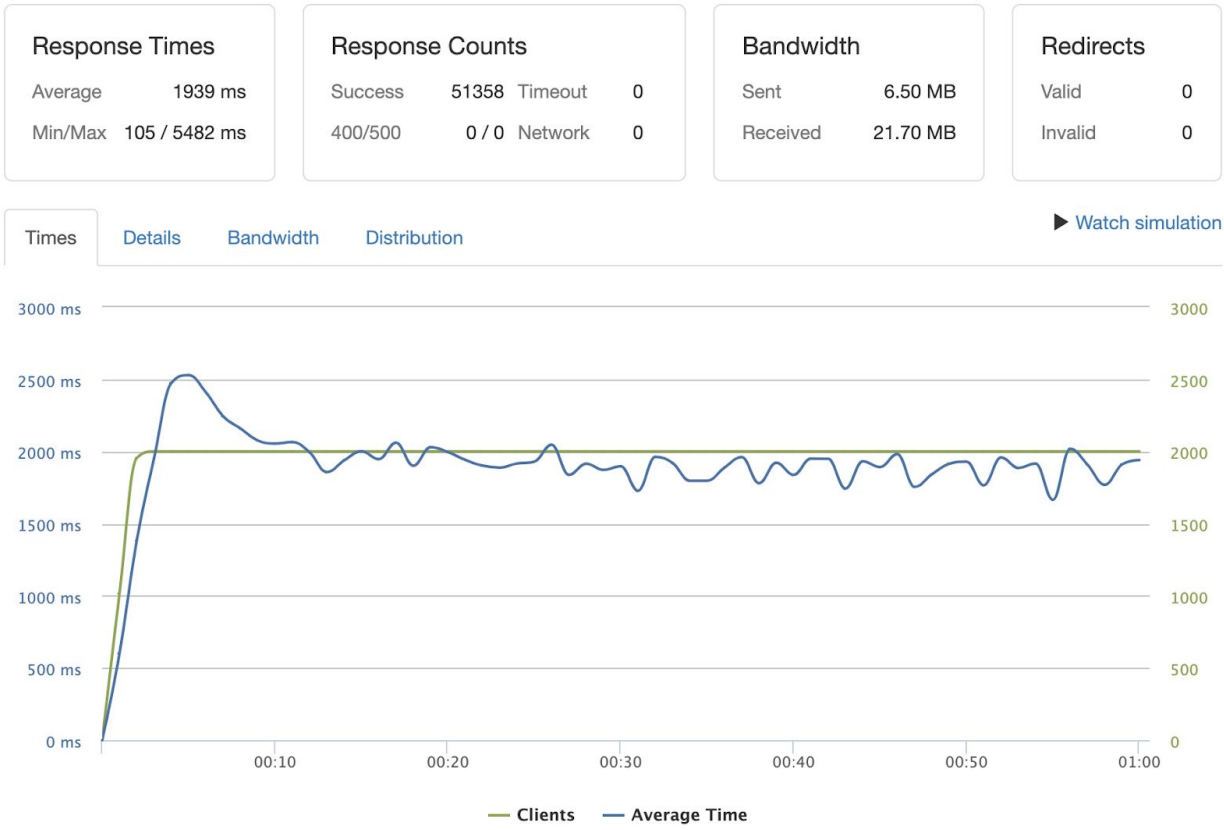
Here are the results: 750 clients per second



Here are the results of the new relic dashboard: 750 clients per second

Then I proceeded to test with 1000 clients per second for 1 minute and saw a HUGE spike in response times. It would take on average 1939ms for node to make a response back and I assumed that this was a bottleneck for my AWS EC2 service running on a t2 micro. Since the little manpower that it currently has it was being capped to a point between 750-1000 clients per second.

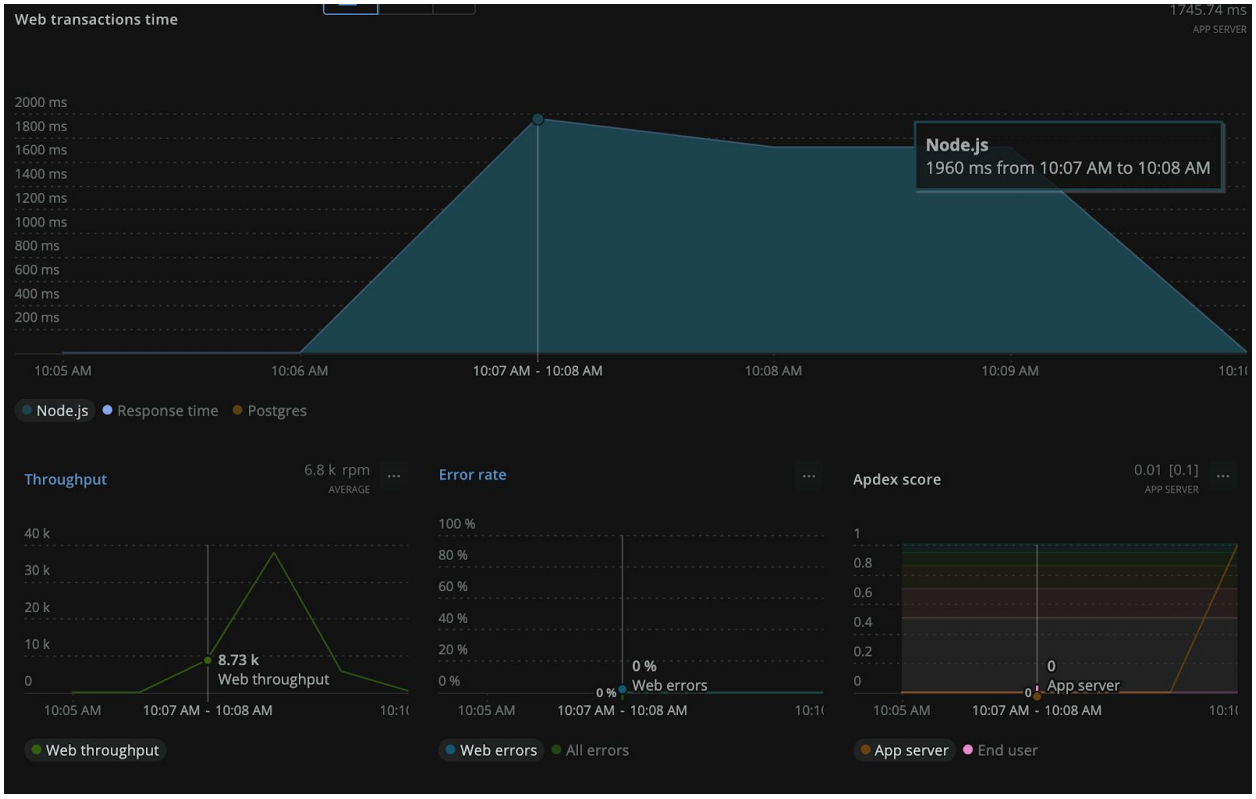
Here are the results: 1000 clients per second



The amount of cpu being used for 1000 clients per second:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 4072 ec2-user  20   0  943664 148924  32820 R  99.9  14.8   4:11.42 node
    1 root      20   0   43556   5268   3940 S   0.0   0.5   0:01.44 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
```

Here are the results of the new relic dashboard: 1000 clients per second



Notes:

I came to realize that when running these loader.io stress tests that there is an initial spike in the beginning of the search and it drops down significantly. Also tracked the cpu usage on my instance and discovered that 1000 clients per second was overloading the power of one t2 micro instance. So that was the bottleneck I encountered and need to scale my service horizontally so that I can spread the requests among multiple service instances.

10/28/2020

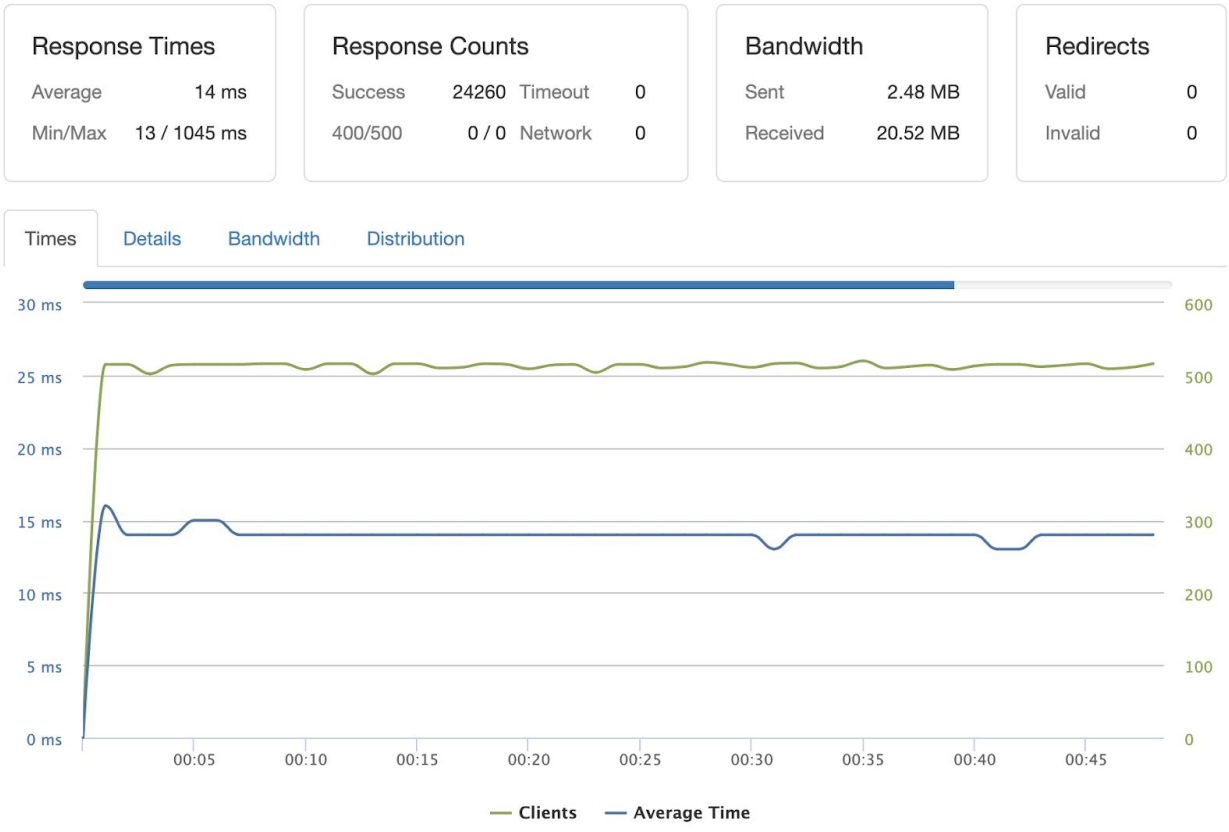
Notes:

Today I finally set up the proxy server / load balancer to serve up my other service instances. I then decided to stress test with loader.io to see how adding another service instance will help distribute the requests.

Testing for 500 clients per second:

Before using a load balancer and just stress testing the actual individual service themselves I was getting around **23ms** response time and after spreading the requests to two services I was able to reach **14ms** which is a **9ms** difference!

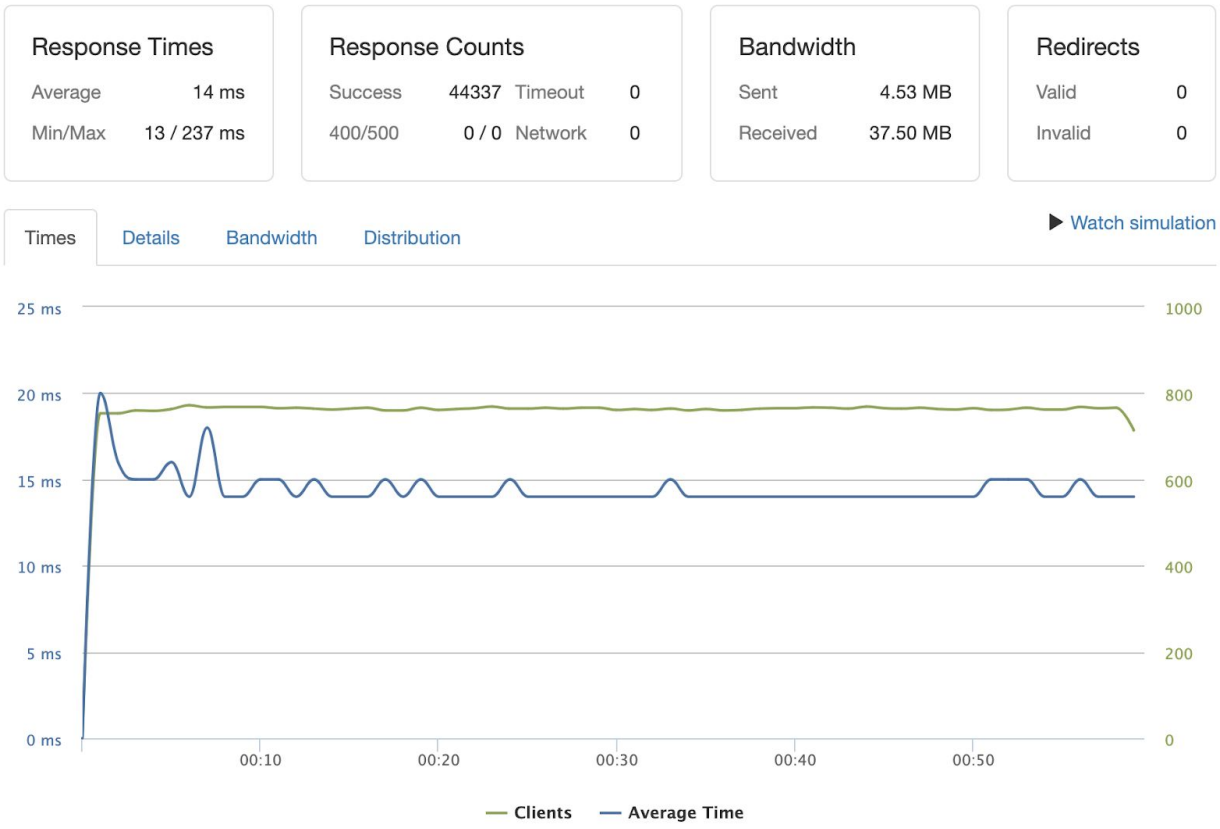
Here are the results: 500 clients per second



Testing for 750 clients per second:

Then testing for 750 clients before, I was able to still provide a decent response time of **100ms**. But after load balancing with NGINX I was able to achieve response times of **14ms.**

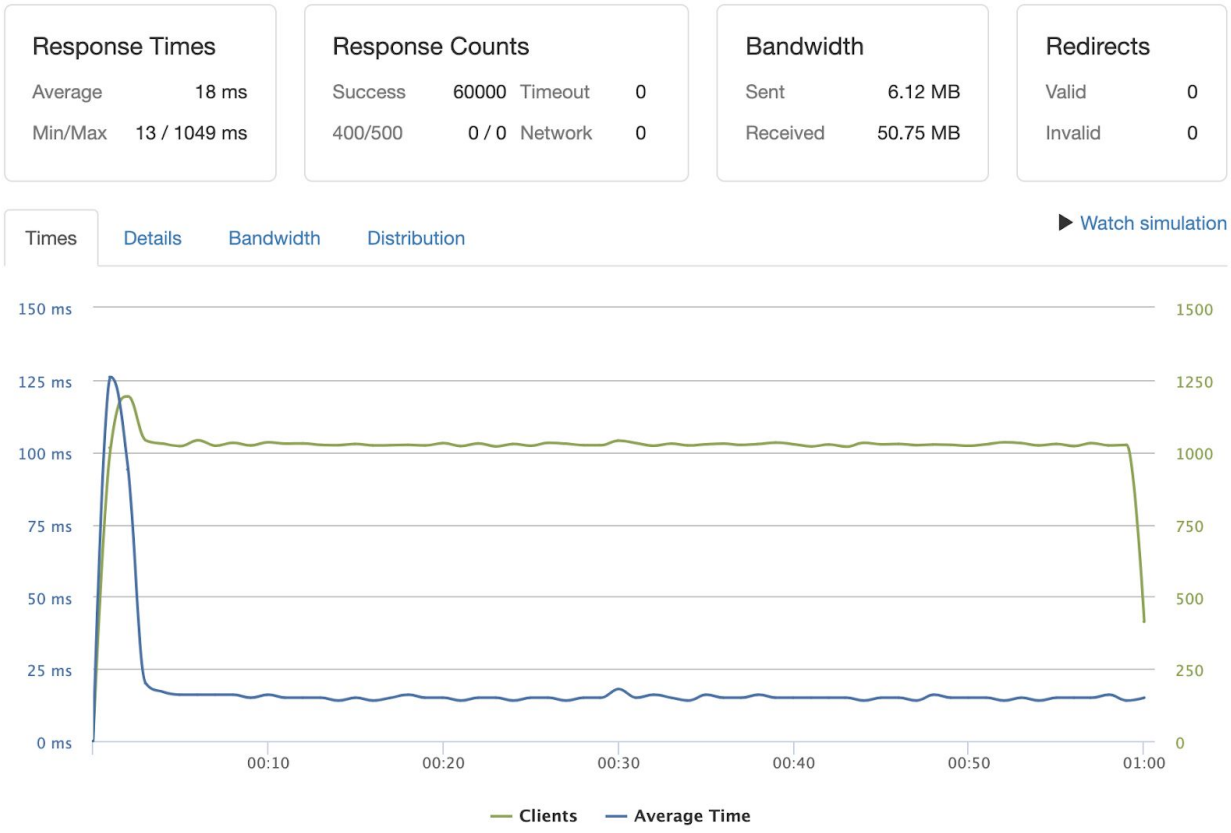
Here are the results: 750 clients per second



Testing for 1000 clients per second:

Then testing for 1000 clients before, I was able to still provide a response time of **1939ms**. But after load balancing with NGINX I was able to achieve response times of **18ms.** Testing for 1000 clients was too much for one instance to handle and this was the maximum it was able to reach. But with the help of a reverse proxy such as NGINX, it helped reduce that time by almost **1921ms.**

Here are the results: 1000 clients per second



Extra Notes:

So in the end I noticed that it helped with the response times by a huge scale and I am going to continue to test and see how far I can horizontally scale until I reach a point of diminishing returns.

10/29/2020

Notes:

Recording bottlenecks and when to horizontally scale out by response times. Going to see how to keep a consistent 30ms or less response time as the clients per seconds go up.

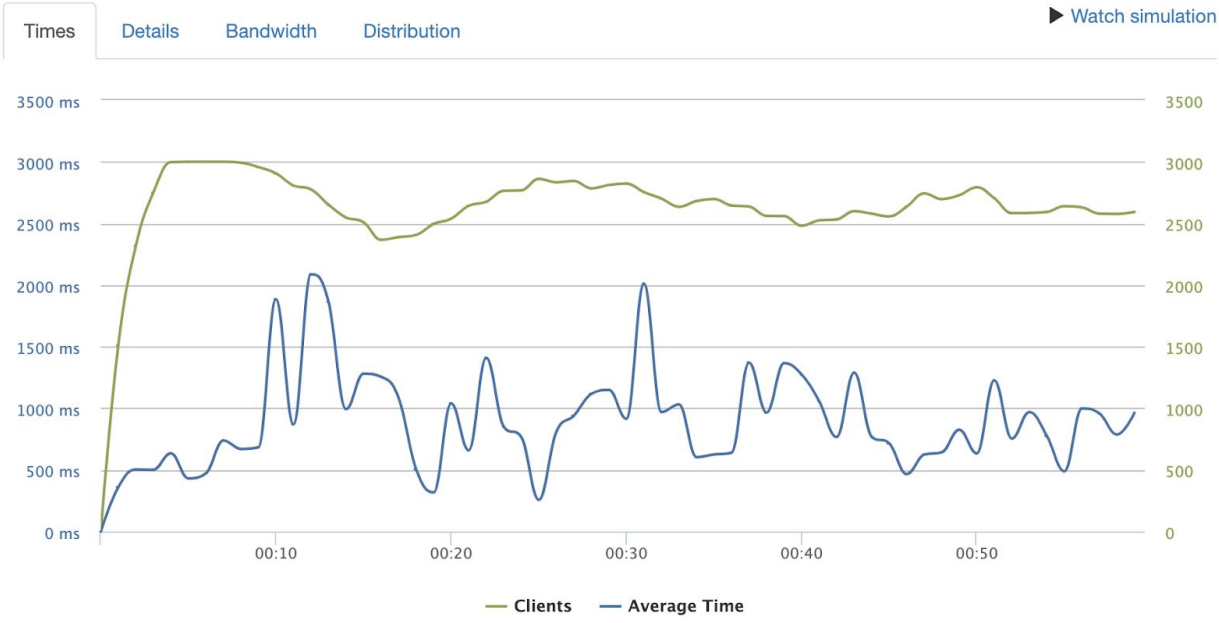
BOTTLENECK 1: (2 services only)

Testing for 1500 clients per second:

Realized that 1500 clients per second was a little too much for my 2 services to handle themselves so 1500 was the bottleneck I ran into. Also **15.4% error rate.**

Here are the results:

Response Times		Response Counts			Bandwidth		Redirects	
Average	920 ms	Success	73002	Timeout 4674	Sent	8.42 MB	Valid	0
Min/Max	12 / 11060 ms	400/500	0 / 8578	Network 0	Received	64.53 MB	Invalid	0

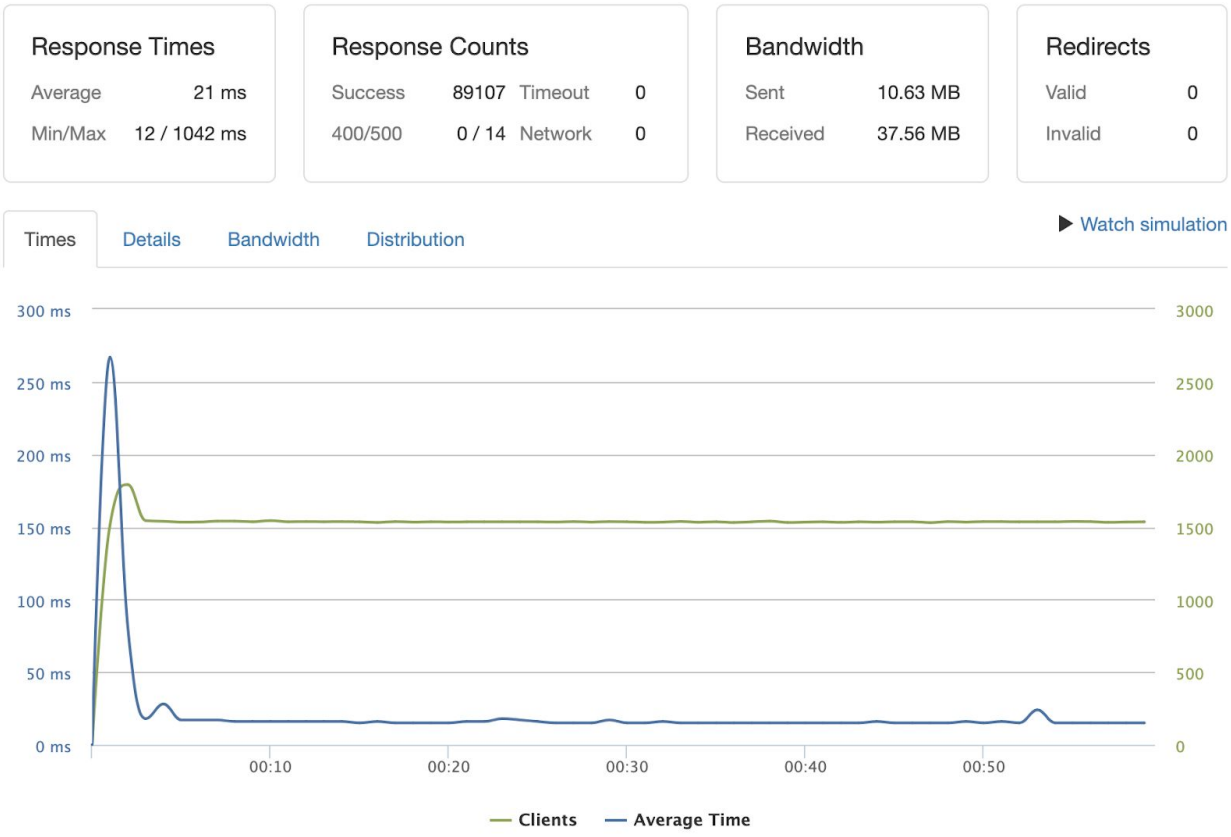


RESOLUTION OF BOTTLENECK 1: (3 services only)

Testing for 1500 clients per second:

The results of adding one more service to divide the requests per second really helped pass 1500 clients per second. Previous response time with 2 services were around **920ms** and with 3 services it's around **21ms.** Also **0% error rate.**

Here are the results:



BOTTLENECK 2: (3 services only)
Testing for 2500 clients per second:
Realized that 2500 clients per second was a little too much for my 3 services to handle themselves so 2500 was the bottleneck I ran into. Also **11.4?% error rate.**
Here are the results:

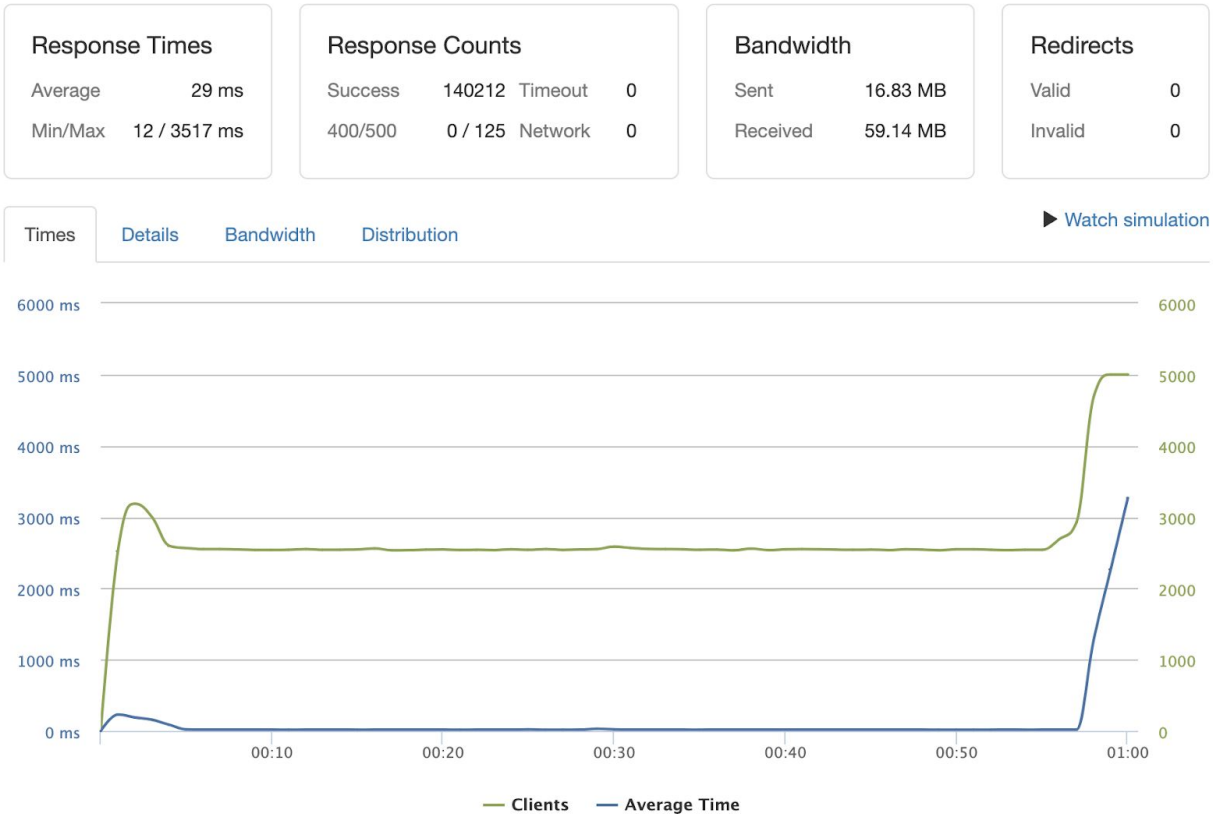
Response Times		Response Counts			Bandwidth		Redirects	
Average	769 ms	Success	126517	Timeout 7437	Sent	16.32 MB	Valid	0
Min/Max	12 / 17318 ms	400/500	0 / 8770	Network 37	Received	56.16 MB	Invalid	0

RESOLUTION OF BOTTLENECK 2: (4 services only)

Testing for 2500 clients per second:

The results of adding one more service to divide the requests per second really helped pass 2500 clients per second. Previous response time with 3 services were around **769ms** and with 4 services it's around **29ms.** Also **0.1% error rate.**

Here are the results:



Extra Notes:

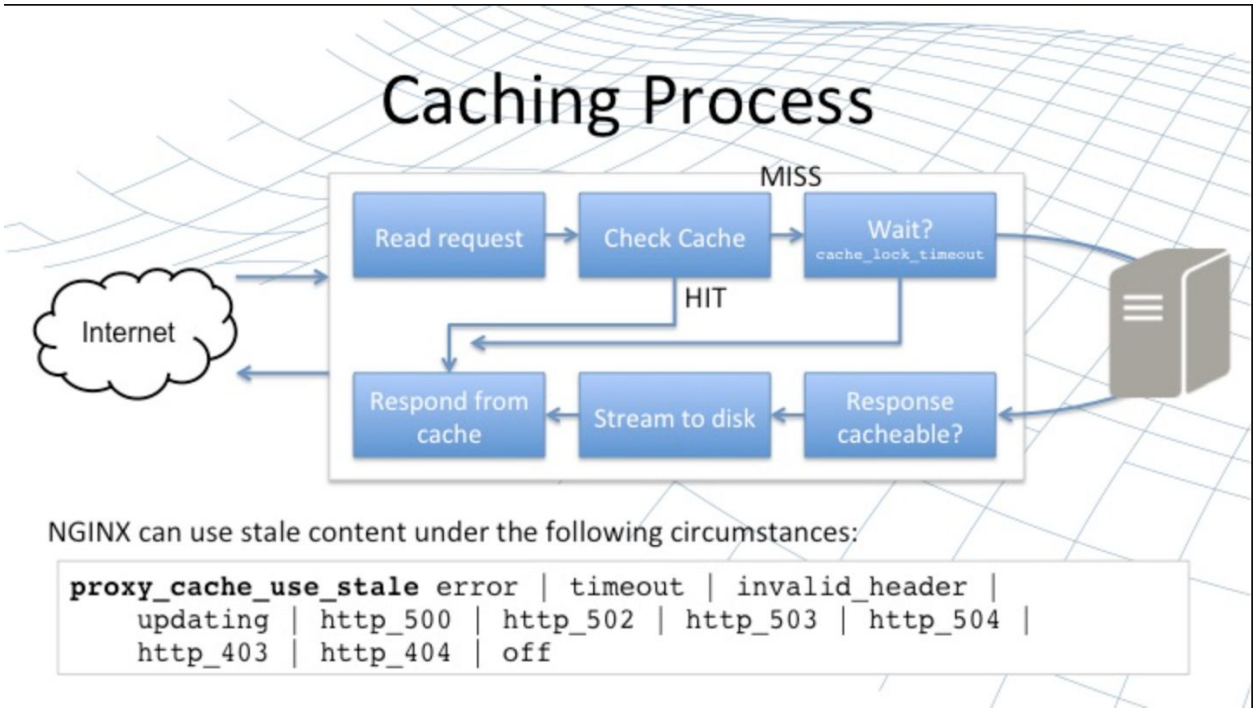
One thing I realized also was that when the response times are way above 100ms, the error rate will increase as well. So response times and error rate definitely have a correlation to one another. Noticed that when testing with loader.io for the proxy/loadbalancer, the clients per seconds would skyrocket up near the end of the stress test. But during its correct amount of clients per second, the response times are stable and below **20ms.**

10/30/2020

Notes:

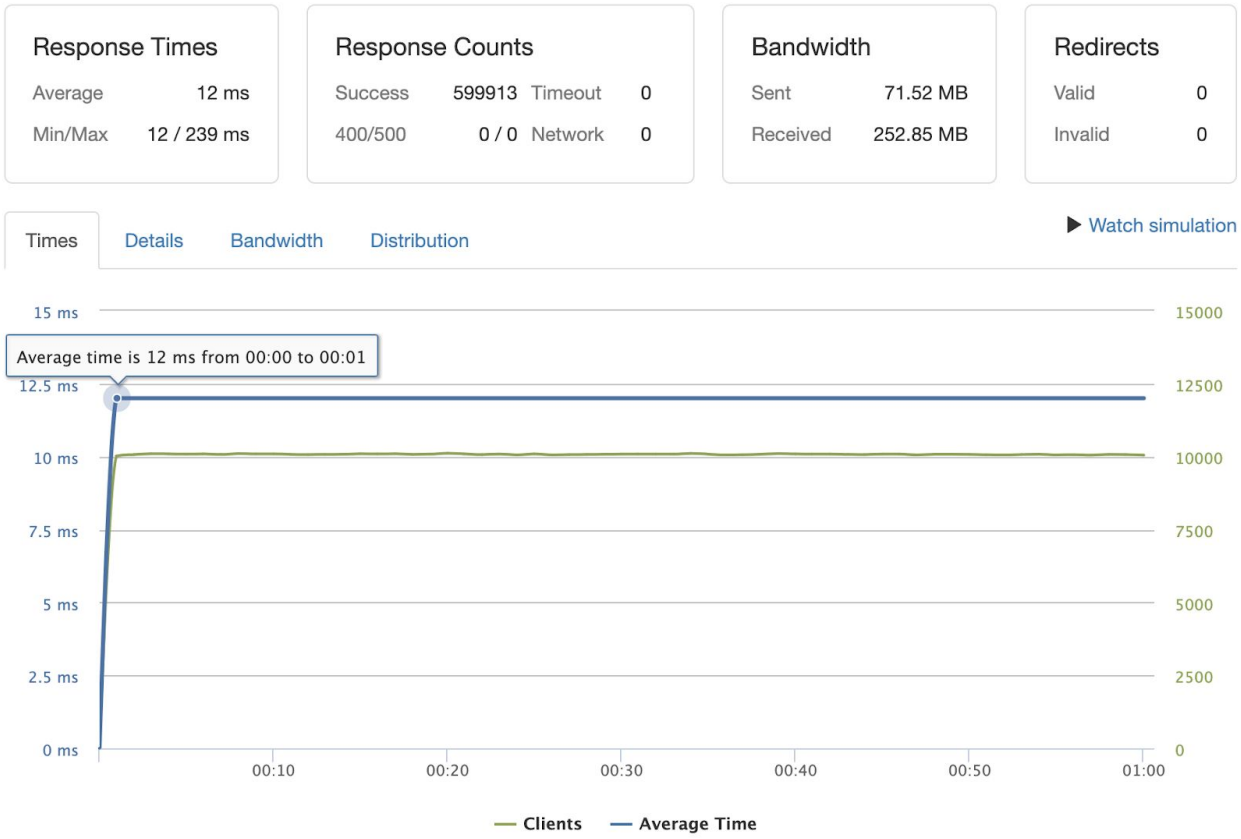
While working on caching today, I realized that caching helps extremely with the response times. I noticed that while stress testing the proxy, caching will store the endpoint and make sure if that same endpoint is hit then it will already have it stored in memory so no search to the service instance is needed.

Here is a small diagram:



1. The **internet** makes a read request (GET)
2. Checks the cache to see if the same endpoint was previously hit
3. **If hit**
    a. Respond with the cache
4. **If miss**
    a. Cache the response
5. Send it back to the client.

Here are the results of 10000 clients per seconds after caching:
The results were at a complete straight line because once it stored the endpoint searched for once, it will always return back the cached response without having to go to the service instances. I also checked to see if my service instances were being used and I could tell that their CPU levels were almost at 0% since they weren't being reached anymore since we already stored the endpoint in the cache's memory.



FINAL CONVERSATION NOTES:

1. **First started off by creating one service instance on aws ec2 t2.micro which is superbly slow. Hence, I stress tested that one service with loader.io in stages of 500, 750 and lastly 1000 clients per second. The conclusion of the 3 different stages was that around 750 clients per second for a 102ms response time was the maximum one instance can handle before the latency would sky rocket to around 1900ms for 1000 clients per second. So I figured that anything above 750 clients per second would be too overbearing for one service instance. Therefore, I was headed towards scaling my architecture out horizontally by adding more service instances.**
2. **Second with the help of NGINX as the proxy/loadbalancer and one additional service instance (so in total two service instances now) after testing that structure out, I saw amazing results. For example a stress test previously with 750 clients per second was 102ms latency and with the requests being shared amongst two services it brought the latency down to 14ms. As for the unachievable 1000 clients per second for one service instance, it went down from 1900ms latency to around 18ms. Furthermore, I decided to continue to scale out and find any bottlenecks while I continue to increase the clients per second rate. I considered a bottleneck to be when my error rate is above 0.1% and response times are in the thousands.**
3. **Third I chose the load balancing technique of least connections because it provided the best response times between all the other techniques. For example with round robin as the load balancing technique there would be a huge spike during the initial stress test for about 15 seconds and then finally drop down, but when using least connections the initial spike would be decreased to around 5 seconds. The other load balancing techniques had negative impacts on my latency and would increase the response times by either 100% or more.**