Spencer Lepine

SDC Engineering Journal and Notes

- [x] Source Code: [GitHub Repo](https://github.com/sdc-bareminimum/project-catwalk-related-service)

Contributors

- [Spencer Lepine](https://github.com/spencerlepine)

Tech Stack

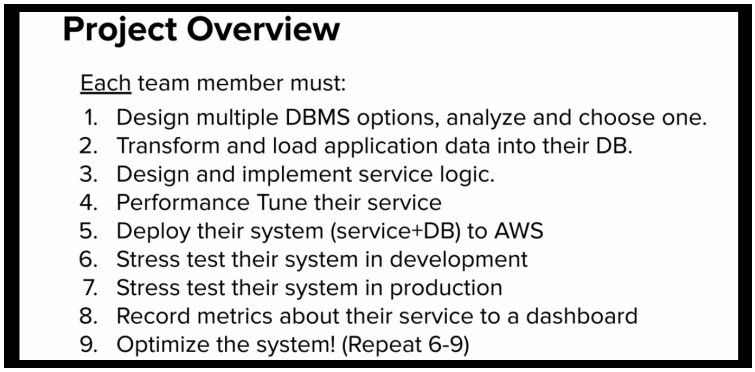
- [Docker](https://www.docker.com/)
- [Postgres](https://www.postgresql.org/)
- [Express](https://expressjs.com/)
- [Node.js](https://nodejs.org/en/)
- [Jest](https://jestjs.io/)
- [Axios](https://www.npmjs.com/package/axios)

API Docs

- [Atelier API](https://gist.github.com/trentgoing/d69849d6c16b82d279ffc4ecd127f49f)

Entries and notes taken during the System Design Capstone (Project Catwalk). This was a project completed at Hack Reactor with a team of 4. These are observations and lessons learned.

This project is the Hack Reactor system design capstone (Project Atelier). This is an API serving retail product info from a SQL database. The goal was to replace the existing API service [Project Catwalk](https://github.com/fec-bareminimum/project-catwalk) and scale to meet the demands of production traffic.



Project Summary:

Day 1:
Read project material/requirements. Create workflow, journal, team group chat, and GitHub repo to store work. Research DBMS systems. Research the SQL and NoSQL approaches for this dataset. Created Schemas for the SQL database, and NoSQL database. Choose PostgreSQL to work with after reasoning the trade-offs. Set up the codebase with Postgres and Node/Express. Worked on creating a GitHub action with a postgres Postgres database to set up continuous integration. Experimented with Docker images in the GitHub action. Struggled a bit configuring the process of migrating (clearing/loading the schemas) the db and seeding the db (populating sample data in the tables).

Day 2:
Set up the local PostgreSQL database. Set up the Sequelize ORM (Object Relational Mapping) library. Created a `init.sql` file with DROP/CREATE TABLE SQL commands that matched the exact schema from day one. I used a tool called `sequelize-auto` to convert the SQL statements into a javascript file that sequelize could use (instead of trying to manually match all the data types for each table name/column). Wrote assertion test's for the Sequelize models to make sure they matched the SQL schema created earlier (slightly an antipattern, but just to be safe). Used Jest testing and the `sequelize-test-helpers` npm package to run queries for the test environment. Modified/Updated my schema. Was going to use an array of IDs for related products, since it just returns [143, 561, 591], but I needed to use foreign keys.

Day 3:
PostgreSQL Database is set up with schemas. Now it is time to use existing .csv files with style, photo, related products, skus, and product information data. Worked on CLEANING the csv files, allowing a line-by-line reader to parse the CSV file with valid column DELIMITERS and new lines. Had trouble with unterminated quotes, causing the CSV parser to read multiple lines as one. Experimented with different csv and file reader packages. (See branch: csv-cleaner for final result). Used smaller 100 lines files to test before cleaning the 2gb+ files. Had various issues trying to even parse the csv data. Had to consider if I should add an entry to the Postgres database WHILE reading each line, OR clean ALL the csv files, and after, separately load them into postgres. Was able to get files cleaned with a couple scripts. Had lots of trouble finding errors between Node file readers and the Postgres db failing to accept data. Was trying to use

the \COPY command in the postgres CLI. Not sure how to get this csv file into a docker container though (since it has an independent file system)

Day 4:
Problem solving the Extract Transform Load process for the CSV data. CVS cleaner was doing a seemingly proper job, BUT it needed to be tweaked for Postgres. Cleaned files, looked for Postgres error, fix CSV file + reclean, and repeat trying to load/copy into Postgres. Finished extracting, transforming and loading the data in Postgres. Had to store the clean csv file in the /tmp folder on my mac, then Postgres could access files in that folder. Completed the Sequelize configuration files in /config folder. Began setting up the test environment and adding `supertest` for easy test http requests. Created express routers and connected them to controllers and models. Most of the boilerplate for the Express server was built at this point. Was doing a good job of separating work between the models, database, server routes, and server controllers. Keeping commits on separate branches and making isolated pull requests. Wrote product endpoint tests to get familiar with jest + sequelize assertions.

Day 5:
Wrote test's for each Express Router endpoint (product info, styles, metadata, related ID's…). Created separate controllers for each, connection to the model and sequelize. The controllers would hold the actual query. Started completing the router files to define the hard-coded REST api scheme, basically translating the API documentation. Installed the k6 tool for load testing the server locally. Wrote a script to test 3 different endpoints and save test output in a markdown file. Was not sure how to perform the load tests, and it was a little early to worry about that. Messed around the database seeding/migrating, since I wasn't seeding any new data (only testing the 4gb+ actual "production" data).

Day 6:
Worked through bugs and issues, but was able to set up all 3 different endpoints. Organized node app files and kept code modular. Wrote a ton of assertions/tests to ensure data returned from endpoint + sequelize query matched exactly to sample data. Transformed some data from Postgres before sending it back to the client. Front-end expected data in a specific format. Explored migrating from the local Postgres db to a Docker image with all the same postgres data. Ran into issues with sequelize configuration and seeding scripts. Ended the day by testing the page/count parameters for the product catalog. There were 1,000,000 products in total, so the max was 999994, or page=199999 with count=5. Getting max 1050ms on the `/products/999999` endpoint, seemed promising.

Day 7:
The API service is almost complete, a little fragile in certain areas though. Worked on putting Node + Postgres into a Docker container. Ran into issues configuring the ports correctly, and also accessing a docker container locally with a random IP address. Docker containers built with my M1 mac ARM chip were not compatible with the AMD64 EC2 instance on AWS. Using a special build command on my local machine with Docker, I could build it for multiple CPU architectures. The next option was to keep the Docker image built from my M1 mac ARM chip,

and emulate this linux architecture on the EC2. Once that was built, I loaded the backup.sql file that was exported from my local Postgres Database. That file contained all the schemas and data for this product (all product entries). The next step was to deploy the docker image onto AWS EC2.

Day 8:
Refactored the codebase and modified the docker-compose files. Had trouble learning Docker and getting containers running on EC2 instances. Was able to push images to DockerHub account. Was trying to run Node server AND postgres containers in the same EC2, which was just a waste of time since I would pull them apart to horizontally scale the Node server anyways. Between Debugging, rebuilding the Docker containers, trying to fix/relaunch the EC2 instances, and tearing down and restarting the Postgres database, it took a LOT of time to work through configuring the postgres passwords, ports, IP addresses, and correctly loading the SQL schemas. Once I had a clearer understanding of separating the containers, it was a milestone to get everything deployed.

Day 9:
Launch an EC2 instance and manually install Postgres. Uploaded a postgres backup.sql file to the EC2, and then ran the \copy command in the ubuntu + psql command line remotely. Everything was deployed, I also had a workflow to update the code and redeploy everything through Docker Hub quickly. I started off by backtracking a bit and running local load testing with the k6 tool. Created a group of prewritten requests in the Postman (a tool for making API requests) app. I exported 8 different URL requests to a file for k6 to use. It could assert that each response had the appropriate status code, and had a latency under 2000ms (target performance). Ran the k6 loading test in the cloud next, which had a web portal dashboard with a chart. The overall performance was pretty bad, so there was some fine-tuning to do. Next, I deployed a NGINX Reverse Proxy and Load balancer docker image to an EC2, which connected to the Node server EC2 and the Postgres Database EC2. Server was performing horribly with load testing. Individual requests were fairly slow. Had a BIG WIN with adding an index to the database. After deploying, the foreign keys were not indexed anymore, and this was taking way too long. Worked through the possible bottlenecks, found the database itself was the slowest (~20 seconds). After optimizing the Sequelize query and adding the SQL index to the database, I can fetch this endpoint normally. From 30000ms to 150ms, that is a 20,000% improvement!

Day 10:
With everything deployed, it was time to stress test my EC2 servers with Loader.io. The k6 (cloud) load test would hit 8 different endpoints at once, for 10 minutes, and report the performance. The loader.io load test now would hit 1 endpoint over a 30 sec or 1min period to max out the server. Tested a light 8.3 request per second (**RPS**), got slow response times. Ran loader.io load test with 8.3 RPS, and the server crashed.. Connected NGINX load balancer to single Node EC2 server. Tested with load balancer and up to 200 RPS over 30 seconds and the server jumped from 170ms average to 1,700ms latency. Ran another k6 (cloud) load test to cover multiple endpoints, and the performance did not improve much.

Day 11:
Worked with AWS AMIs (Amazon Machine Images) for cloning an EC2 instance to spin up another node server quickly for horizontal scaling. No need to install git, node, npm, and whatever else EACH time (or remember which version of node to use). Used DockerHub to upload/pull the spencer lepine/node-server image quickly down. Ran loader.io test with 2 EC2 Noe servers. Added another EC2 instance, now 3 [Loader.io test, 250 RPS, 4779ms latency]. Perhaps a bottleneck. Modified the nginx configuration and redeployed. Connect multiple EC2 instances and use the Round Robin algorithm for load balancing by default.With this, it was able to handle 150 clients/requests per second with an average latency of 2596ms. Much better. Ran the load balancer between FOUR EC2 instances now, scaling horizontally. Got 1549ms latency (still not ideal) and 223 RPS throughput. Switched to Least Connected IP load balancing alg. Average response time of 103ms latency and ALL 250 requests (per second) throughput were successful. HUGE improvement. Added caching, got 64ms latency on 250 clients per second. This shaved off 20ms. Ran loader.io test now for 500, 1000, 2000, 2500, 3000, 3500, to 4000 RPS which had 234ms latency. Ran broad k6 (cloud) load test over 10 minutes, averaged 700 RSP with response time of 70ms. That is amazing. Added more configuration options to nginx load configuration. Was able to load test 5000 RPS on loader.io over 30 seconds.

Day 1:

Phase 0:

- Read through project material on Learn.
- Meet with group, choose services that we will work on -> me: Related Products
- Create a GitHub organization and add team members
- Create slack group
- Create Trello board for ticketing system
- Each team member Creates a Repo for their service
-

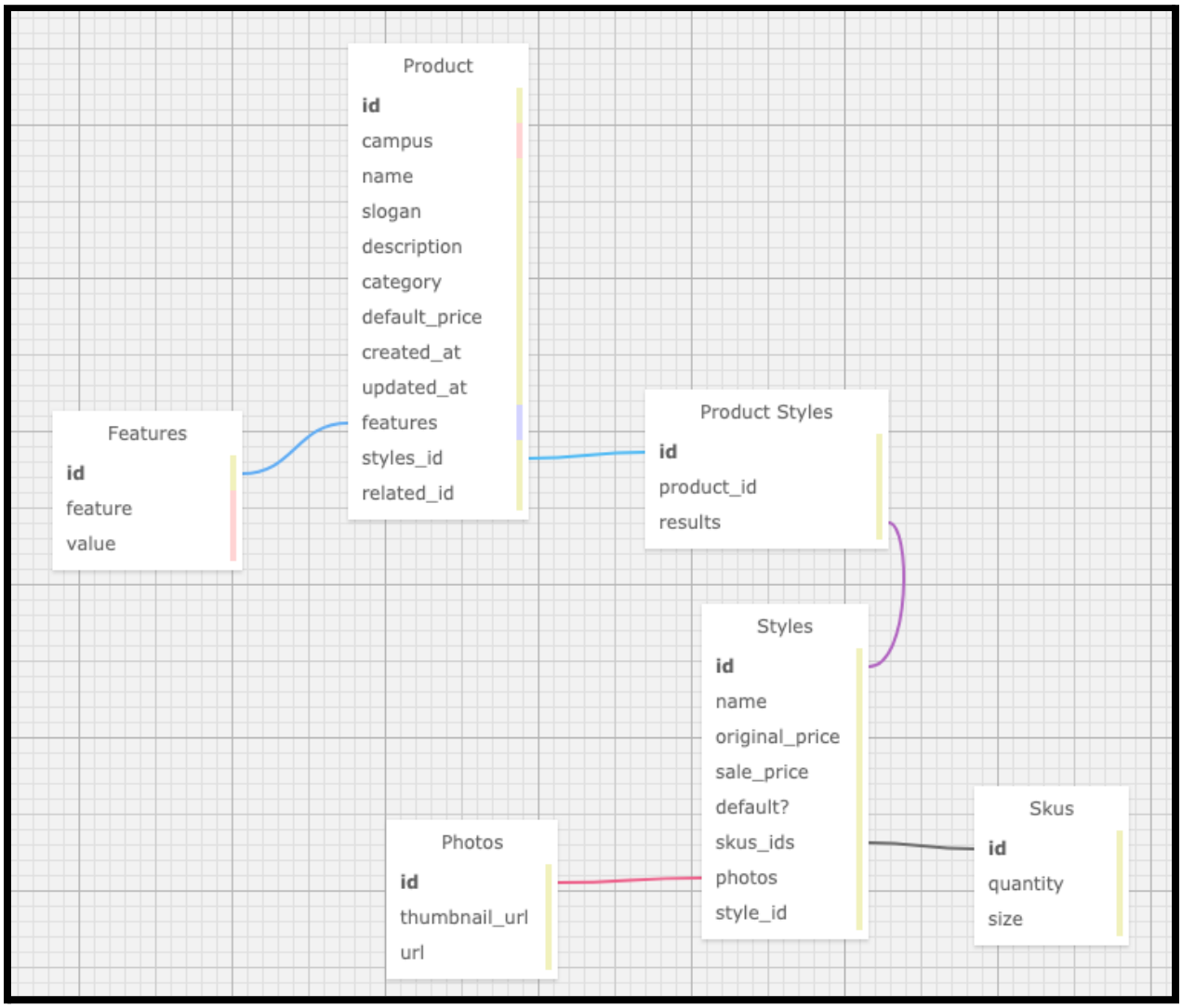
Phase 1:
DBMS Design and Creation
Select two DBMS technologies (one RDBMS and one NoSQL DBMS)

Compare the databases for this project use case:

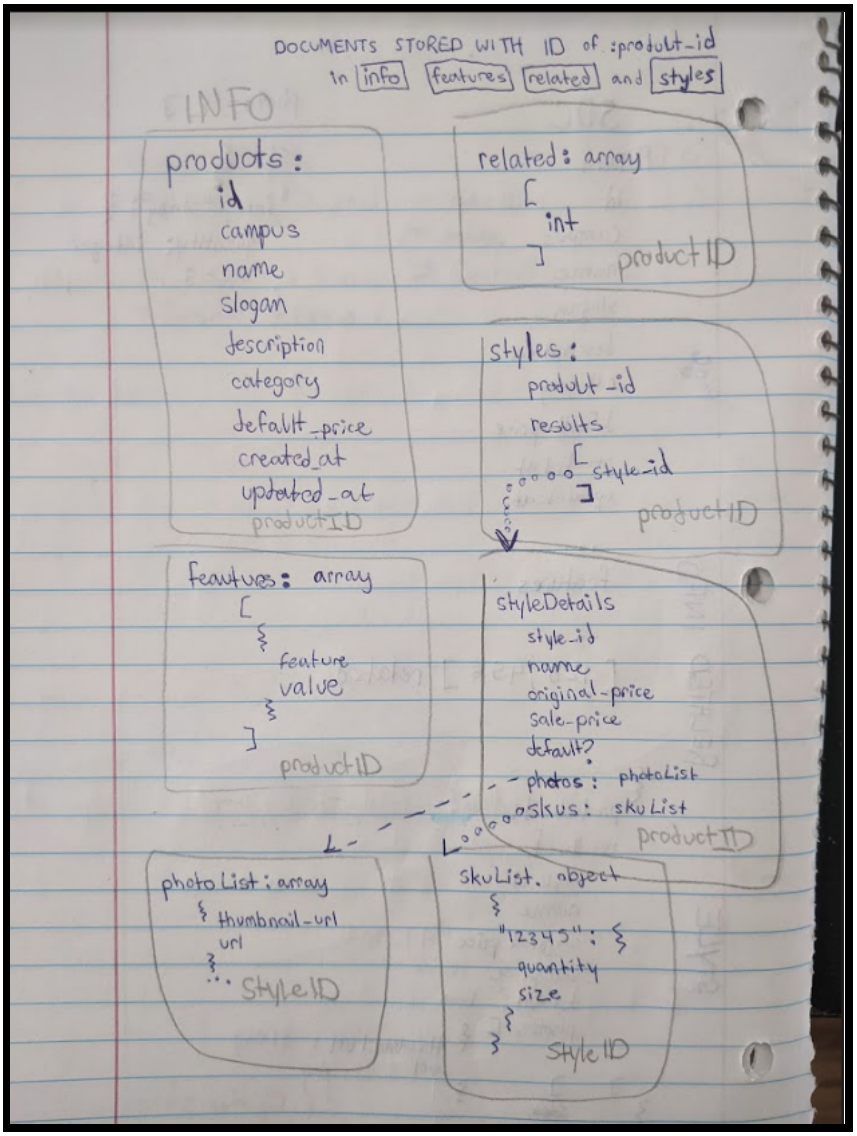
NoSQL	**SQL**
- Reshape data during development for an agile development process - Easy to scale horizontally by just adding more nodes. - Lower Disk I/O bottleneck - Good for unstructured/"schemaless" data - Can store data already how the API will receive it - Better for Agile development (no need to update schema	- Tabular predefined schemas that must be followed from the start - Storing product inventory, not a user feed or - Allows us to run complex SQL queries and work with lots of existing applications based on a tabular, relational data model, PostgreSQL will do the job. - Usually “scale-up” approach for large data

and reload - entire SQL database (migrate)) - Cheaper with scale-out approach for large data sets - Better to scale out for volume of traffic and/or size of data - Less organized -> different forms of data, not always predefined - MongoDB supports ACID (atomicity, consistency, isolation, and durability) - MongoDB also saves documents in a JSON-like format, which means it's very simple to convert queries and results into a format your frontend code understands	volumes - Easily connect product tables (styles, info, features) using relations - Queries for predetermined product IDs will be fast, but we have flexibility to expand

Model the Relational (SQL) Data with [SQL Designer](https://ondras.zarovi.cz/sql/demo/)



Model the NoSQL data:



Deciding between SQL and NoSQL:

After weighing the benefits of relational and non-relational databases, it is a clear decision that SQL will suit the needs for storing products for this service.

- Storing Product data in a non-relational database will still need to be broken apart. Different endpoints for products will be queried, perhaps more than others. With a relational database, we are able to create multiple tables that are linked. Without relations, it could be difficult to update records without breaking a link.
- transactional integrity is not needed.
- Product information will be heavily READ operations. Users will not be editing these records. We need a way to update records for photos, styles, or sale prices for admins. We can isolate those tables, since you likely wouldn't update the ENTIRE product (e.g. name, category), just entries for images or styles.

- Complex queries with SQL may be slow and take time to architect/design, but that isn't an issue. The data is queried with predetermined IDs, which will be closer to constant time.
- Database can be migrated, but the schema is already known.
- Not as scalable? Large online retailers such as Walmart, Best Buy, and Home Depot cluster multiple servers around one store.
- Early stage projects may not have requirements 100% defined, so it's best to work with SQL to start
- SQL can enable Data Analytics

Choosing Postgres

Using a SQL database for this project is a great option. Now though, which SQL database technology is the best? My research has led to Postgres.

Postgres is a great tool for serving large databases. It also has many options to scale. It has more tools available, like table inheritance and function overloading.

It supports all needed data types:
- Primitives (int, numeric, boolean, string)
- Structured (date, timestamp, interval, array, range, uuid)
- Documents (JSON, XML, Hstore)
- Geometry (point, line, circle)

Other distinct advantages include:
- Capable of being ACID compliant
- data integrity
- allows basic or advanced indexing
- available security features
- protect against data loss

STRUGGLED: Spent hours trying to set up a GitHub action with Postgres. Trying to set up continuous integration with Postgres. Need to connect a USER + PASSWORD to connect to Postgres. Also need to run a `init.sql` to create a table that the test will use.

Docker image can run postgres easily, but the GitHub action on Ubuntu needs a way to access and run a `init.sql` file. SOLVED: created a Docker folder and manually run scripts during the GitHub actions instead of using the docker `postgres` image.

```yml
    - name: Config Postgress
      uses: ./Docker/
      with:
        postgresql password: example
        postgresql init scripts: ../config
```

Notes:
Researching Postgres:
For production: create pool w/ `pgBouncer` so we don't open client connection EVERY query:
http://www.pgbouncer.org/

Postgres + Node Testing Article Research:
https://medium.com/geoblinktech/postgres-and-integration-testing-in-node-js-apps-2e1b52af7ffc

Node DB Testing Article:
https://www.coreycleary.me/know-what-to-test-using-these-recipes-node-service-that-calls-a-database

Read about Docker + Postgres:
https://www.thisdot.co/blog/connecting-to-postgresql-with-node-js

Note: might use Sequelize

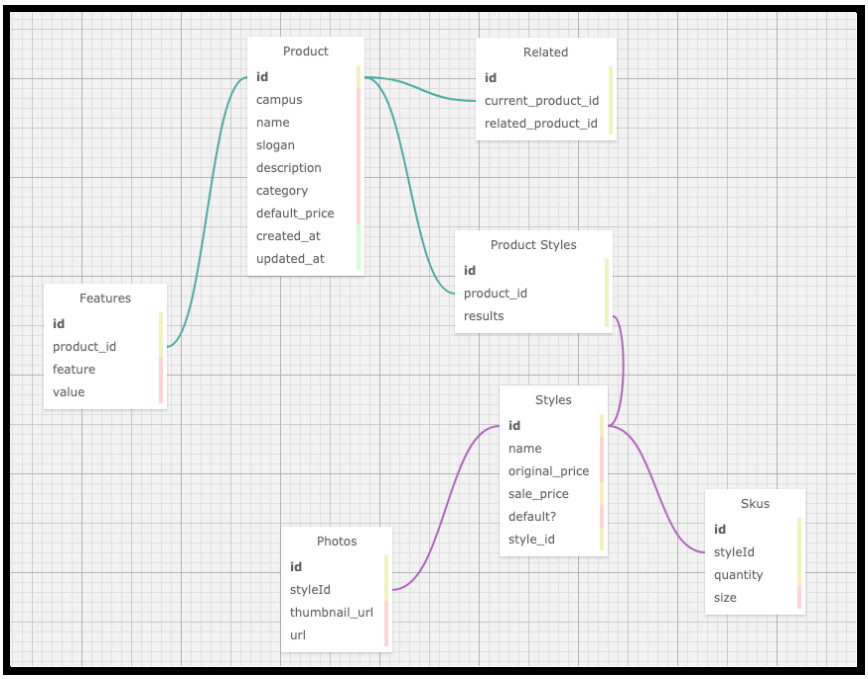
Read About Node, Postgres, and Sequelize:
https://mherman.org/blog/node-postgres-sequelize/

Read about connecting Docker Image + Postgress + GitHub Actions
https://remarkablemark.org/blog/2021/03/14/setup-postgresql-in-github-actions/

Day 2:

Today I set up a Postgres database. This database is connected to the sequelize ORM (Object Relational Mapping) tool. I created a TABLE schema with the correct DATA types for the SQL database. I made a `init.sql` to DROP and CREATE all of the tables, and ran this in the postgres database. Then I used the `sequelize-auto` tool to generate `sequelize` model files, which is a slightly different API than pure SQL. That system will make it easy to migrate a schema (paste a new init.sql file in the postgres cli, then generate the new files from `sequelize-auto`).
With a working ORM, I set up basic tests for the Sequelize Models to check the Model properties (row headers of a table). I used [sequelize-test-helpers](https://www.npmjs.com/package/sequelize-test-helpers) to avoid actually connecting to the postgres every test.

Also updated the SQL Schema. Cannot use ARRAYS or SETS, that would be a NoSQL schema.



NOTES:

CREATE Sequelize MODELS:

Take SQL initialization script and convert it to Sequelize

Use `sequelize-auto` https://github.com/sequelize/sequelize-auto. Take existing SQL CREATE TABLE commands and create sequelize models. This makes it easy to KEEP a init.sql file incase we migrate to another database.

```
// Run Postgres in terminal ($ psql)
// Initialize the database with `config/init.sql`
// Install `sequelize-auto` https://github.com/sequelize/sequelize-auto
// Run the following command (all one line):
//  $ npx sequelize-auto -h 127.0.0.1 -d postgres -u postgres -x example -p
//        5432  --dialect postgres -c ./config/db.config.js -o ./app/models -t
//        style product styles features skus related photos
```

Stress Test: use K6?

Read about Sequelize Testing:

https://itnext.io/unit-testing-sequelize-models-made-easy-108f079f1e38

Use `sequelize-test-helpers` to keep from invoking postgres connection

Connect to PostgresQL: https://node-postgres.com/features/connecting

Connecting to PostgreSQL with Node.js: https://www.thisdot.co/blog/connecting-to-postgresql-with-node-js

Express + Sequelize API and unit testing:

https://levelup.gitconnected.com/building-an-express-api-with-sequelize-cli-and-unit-testing-882c6875ed59

SQL commands for Sequelize

https://fengmk2.github.io/blog/2014/10/sql-to-sequelize-mapping-chart.html

Use existing SQL Schema of TABLES and relations in Postgres to create `sequelize` compatible files:
https://github.com/sequelize/sequelize-auto
```
npx sequelize-auto -h 127.0.0.1 -d postgres -u postgres -x example -p 5432  --dialect postgres -c ./config/db.config.js -o ./app/models -t products
```

Day 3:

Now that I have chosen a database, it is time to Perform an ETL Process. I will transfer the full application data set from a CSV file into Postgres.

Challenge:
Clean/transpile the CSV file into a new one, and then import them into the Database all at once afterwards? OR Read the CSV file, clean the data, and input into the database AS we go.

Action:
Look through CSV files, see how data may be “incomplete”. Set up a CSV reader and read sample files. Transform the SQL schema to use foreign keys without a SET of IDs. Found mismatching keys in CSV data. “styleID” camel case gives errors in Postgres. Need to convert this to “style_id” during the ETL process. Fix syntax errors and unterminated string quotes.

Result:
Read the CSV file line-by-line. Generate a clean csv file. Then we can use that file with a Postgres COPY command.

Honestly, I'm really struggling. Running Postgres on my local MacOS. Also tried running the PostgreSQL in a docker image. Getting many errors trying to use the \COPY command.

Day 4:

Failing to load CSV files into my Postgres database. Instead of assuming the data was correct, I had to break the problem down.

- First, I had to verify that the Postgres command line COPY command was using the correct values.
- I also needed to verify my SQL schema matched the CSV headers exactly
- The next idea was to load a sample CSV, which was known good. If the CSV file is correct, then we can focus on narrowing down on the SQL schema or Postgres.
- Next, increase the SQL Schema restrictions. Some column values (e.g. description) have a VAR CHAR(255) length limit, which can throw an error.

That seemed to work! After modifying the SQL schema to extend the rules, and ensuring the

Finished Modeling the data. Finished extracting, transforming and loading the data in Postgres. Add indexes to prepare the database (adjustments/optimizations).

Many things to think about when processing the CSV files.

- - Should I combine csv files into one for a single large upload?
- - Should I CLEAN the csv data, then copy into Postgres in place? (as we go, line by line)
- - Should I generate new CLEAN the csv files, and copy in Postgres separately?
- - Should the process stop/fail if a csv file line throws an error?
- - Should the cleaning process skip over lines?
- - Should I manually query Postgres after reading a csv file line? Or use \COPY
- - What to do if column values don't match up?
- - What order should I upload the tables that have foreign key references?
- - How much should I transform/filter the data from a query when sending data back to a client from the api endpoint?

SUCCESS! I managed to CLEAN the products.csv file, copy it into `/tmp`, open the PostgreSQL CLI for the database, and execute the COPY command. Now I can run `SELECT * FROM product` and see all entries.

Next: Create + Test First Enpoint

Set up Sequelize

- [x] Organize Database files (`connection.js` + `/models`)
- [x] Create `/seeders`, and `/migrations` folder with `npx sequalize-cli`
- [x] Create `config/config.json` for `sequalize` settings
- [x] Create `config/options.js` to route database folders to `sequalize`
- [x] Added scripts to package.json:

Test Enviroment

- [x] Set up test environment to run `sequelize` test postgres database
- [x] Set up `supertest` library in API endpoint test

Products Controller

- [x] Created `/routes/products/index.js`
- [x] Created `/controllers/products/index.js`

Connect Server Files

- [x] Build express server routes
- [x] Connect model + controllers + routes + server

Test New Endpoint

- [x] Create `routes/product/product.test.js`
- [x] Write test assertions:

```js
const request = require('supertest');
const { app } = require('../../index');
...
describe('/products', () => {
    it('should return array', async () => {
      const res = await request(app).get(endpoint);
      expect(res.statusCode).toEqual(200);
      expect(res.body instanceof Array).toBeTruthy();
    });
});
```

NOTES:
How to get the data in AWS? The Postgres data can be backed up. Restore the db on AWS. You can use the psql command line options to interact with a server remotely!

Useful Commands:
```sh
npx sequelize-cli seed:generate --name projects --options-path=config/options.js
npx sequelize-cli db:seed:all --options-path=config/options.js
```
Relevant Links:
-

[Article](https://levelup.gitconnected.com/building-an-express-api-with-sequelize-cli-and-unit-testing-882c6875ed59) - Express, Sequalize, Unit Tests
- [Question](https://github.com/sequelize/cli/issues/28) - Sequelize Migrations

New Dependencies 🧰
`npm i sequelize && npm i sequelize-cli sequelize-test-helpers cross-env supertest -D`

Read this great article for setting up Express and Sequelize:
https://levelup.gitconnected.com/build-an-express-api-with-sequelize-cli-and-express-router-963b6e274561

Read this great article for Unit Testing Sequelize:
https://levelup.gitconnected.com/building-an-express-api-with-sequelize-cli-and-unit-testing-882c6875ed59

Technologies to use:
Use k6 for stress testing locally.

Use the cloud service stress tester Loader.io (deployed testing)
- Throughput is (successful responses / (Clients per second * Seconds Run))

Use New Relic (a monitoring platform) for local testing (use config file)

Metrics :
Metrics that describe system performance:

Response time: (aka latency): how fast does your API respond? Goal: < 2000 ms under load
Throughput: how many requests can you process per second (Request per seconds)? Goal: 100RPS on EC2 (use v Request per minutes though).

Error rate: how often does a response generate an error? Goal: <1% error rate under load

A successful stress test should be under 2000 ms, and <1% error rate. Get 1000 RPS in development, then just test production

Day 5:

Wrote tests for API routes. Translate the pre-existing API documentation into Express router endpoints. Write assertions tests for expected data at each endpoint. After creating the router, I worked on the controller file that would connect to the database for the actual query.

Created branches and pull requests for each endpoint. Need to implement the controller to fetch data, and write integration tests for expected response for EACH.

Set up the [k6](https://k6.io/) for stress testing.

Created an npm script to generate a (time-stamped) folder with endpoint tests. `npm run k6:load`. All endpoints return 100% checks. See `resources/k6-tests/test_NNNN/*` for results in markdown files.

Ran a load test with k6 for these three endpoints:

```
- products
- products/42370/styles
- products/42370/related
```

Notes:
Read this [article](https://medium.com/codeinsights/how-to-load-test-your-node-js-app-using-k6-74d7339bc787) to set up k6.

Useful commands:

`npm run sequelize db:seed:undo:all`
`npm run sequelize db:seed:all`

Finish writing tests for Styles object, Add ALL endpoints to route index.js, Create styles controller, Bundle router + controller + test into one PR

```sh
# Output to folders:
ENDPOINT=products k6 run ./k6.js > "test_${NOW}/products.md"
ENDPOINT=products/42370/styles k6 run ./k6.js > "test_${NOW}/styles.md"
ENDPOINT=products/42370/related k6 run ./k6.js > "test_${NOW}/related.md"
```

Day 6:

After many bugs and rabbit holes, I finally made progress to implement ALL three API endpoints. The router and controller files were well organized and easy to isolate. Keeping the code modular helped with development.

I tested each endpoint to verify all expected data was being returned. I had to update the controllers to fetch the proper data, as well as return formatted data. The SQL database did not store data exactly how the API returns it. Once the controllers returned the proper data, the integration test for each endpoint was passed again.

Currently, I have an existing PostgreSQL database running on my local machine from the desktop app. This database contains all of the sample CSV data loaded from files on my machine. I pasted the `config/init.sql` file to load the SQL schema. I cleaned the CSV with a Node file reader line by line to fix unterminated commas and validate column values. I then pasted the clean CSV file into the /TMP folder on my mac. The desktop Postgres application has access to this folder. I then ran ` \copy product FROM '/private/tmp/product.csv' DELIMITER '|' CSV HEADER;` for each csv file.
I wanted to be able to load this data into a postgres docker container. I found this great [method](https://stackoverflow.com/questions/35679995/how-to-use-a-postgresql-container-with-existing-data) to include existing data. Since there were over 20 million data entities, I needed this PostgreSQL database to be portable somehow.

Error: Unable to get sequelize db:seed:all to run. Used this command: `npx sequelize db:seed:all --options-path=config/options.js --debug`. My SEEDER file was trying to overwrite the actual database with the same ID. Since I didn't have a test and development database, I didn't erase the entries each round. Once it seeded, it would throw an error after trying to seed the same document a second time

BUG: updated the package.json scripts running sequelize to drop, create, migrate, and seed the database. It would clear the 'test_db' in postgres, upload the model schemas, and then add some sample entries (created by me in /seeders). The GitHub actions kept failing, but I managed to tweak the environment variable to configure the postgresql database. Github is creating a new postgres database every time the workflow runs.

Worked on creating Docker images for Postgres. Worked on Creating a Docker image for Node

Running some test on endpoints:
- http://localhost:3000/products (there were 1000000 products total)
  - (page 1, count 5) Average 6-15ms
  - (page 200000, count 5) Average: 205ms

- http://localhost:3000/products/999991
 - Average: 150ms

- http://localhost:3000/products/999991/styles
 - Average: 1050ms

- Where I left of:
- Put Node + Express + Postgres app into docker container/image
- Deploy the app onto Dockerhub?
- Deploy the app on EC2
- Run new relic stress tests
- FIXING THE RELATED ENDPOINT

Notes:
https://dev.to/studio_hungry/how-to-seed-a-postgres-database-with-node-384i

Create a docker image: `docker-compose up`
Load the Postgres backup: `docker exec -i $(docker-compose ps -q postgresContainer ) psql -Upostgres < csv_data/backup.sql`
Add the test database: `docker exec -it $(docker-compose ps -q postgresContainer ) psql -Upostgres -c 'CREATE DATABASE test_db;'`

Dockerizing a Node.js / PostgreSQL App Backend
https://medium.com/@zbbergma/dockerizing-a-node-js-postgresql-app-backend-ac81750cf6df

Work on Containerizing Docker?
https://docs.docker.com/docker-hub/repos/

Day 7:

Working on putting Node + Postgres into a Docker container. I followed the documentation for writing a Docker-compose.yml file that will maintain the set up for the Docker Image. The Docker container will create a service for the Express API on port 3000, as well as the Postgres Server at port 5432.

Problem: Docker containers built with my M1 mac ARM chip were not compatible with the AMD64 EC2 instance on AWS.

Action: There were two ways to solve this. Build an image locally for multiple linux systems, OR emulate the specific x86 architecture on the container host machine (the EC2 machine).

Result: Using a special build command on my local machine with Docker, I could build it for multiple CPU architectures.

```sh
#
https://github.com/docker/docker.github.io/blob/2d8b420d3c49712ec4a7bcec1464278fa4c419
36/docker-for-mac/multi-arch.md
# BUILD A MULTI ARCH IMAGE
docker buildx ls
docker buildx create --name mybuilder
docker buildx use mybuilder
docker buildx inspect --bootstrap
# cd path/to/file/with-Dockerfile
docker buildx build --platform linux/amd64,linux/arm64,linux/arm/v7 -t
spencerlepine/demo:latest --push .
```

The next option was to keep the Docker image built from my M1 mac ARM chip, and emulate this linux architecture on the EC2. This involved downloading an external tool through docker called qemu, and it allowed docker containers to run through the emulator.

```sh
sudo su
sudo apt-get install qemu binfmt-support qemu-user-static # Install the qemu packages
docker run --rm --privileged multiarch/qemu-user-static --reset -p yes # This step
will execute the registering scripts
docker run --rm -t arm64v8/ubuntu uname -m # Testing the emulation environment
docker pull spencerlepine/node-server
docker run -p 3000:3000 -d spencerlepine/node-server
```

Once that was built, I loaded the backup.sql file that was exported from my local Postgres Database. That file contained all the schemas and data for this product (all product entries). The next step was to deploy the docker image onto AWS EC2. Follow this: https://github.com/soumilshah1995/Deploy-Docker-Container-on-AWS

Notes:
https://www.youtube.com/watch?v=Dm0CmZz-QyI
https://github.com/hidjou/classsed-docker-tutorial/tree/done

https://stackabuse.com/deploying-node-js-apps-to-aws-ec2-with-docker/

Docker Can't connect:
`sudo chmod 666 /var/run/docker.sock`

My M1 Chip is
https://stackoverflow.com/questions/66350893/why-macosx86-can-run-docker-arm-container-arm64v8-alpine

Docker Bridge

```
## Building a Docker Container
Follow these steps to build and run a docker container.
The compose file will create the Node server, and Postgres database.
1. Export/Backup an existing Postgres database:
   `$ pg_dump postgres > csv_data/backup.sql`
2. Run the following `docker-compose` commands:
```sh
# PostgreSQL Connection String
#
postgres://<POSTGRES_USER>:<POSTGRES_PASSWORD>@<DATABASE_HOST>:<DATABASE_PORT>/<POSTGRES_DB>
# postgres://example:1234@db:5432/postgres_db
# Configure the Postgres Database Name + User
$ cp docker-compose.yml.sample docker-compose.yml
# Build the container
$ docker-compose up --build -d
# After updating code, rebuild the container
$ docker-compose down
$ docker-compose up --build -d

# Test the database connection
$ docker exec container_name_server_1 npm run test:db:connection
# Enter the psql CLI
$ docker exec -it container_name_db_1 psql -U example -W postgres_db
#                                 #
#  ***** POSTGRES SET UP *****  #
#                                 #
# Upload Postgres backup into docker container:
$ cat ./csv_data/backup.sql | docker exec -i container_name_db_1 psql -U example -d postgres_db
# Print the Postgres data INSIDE the docker container:
```
```

```
$ docker exec -it $(docker-compose ps -q db ) psql -U example -d postgres_db -c '\z'
```

Day 8:

Today was spent learning about Docker containers. I worked on refactoring my Docker container setup and cleaning up the code base. I am having a lot of trouble understanding how to correctly deploy my code to an EC2 instance.

SOLVED: Successfully deployed the Postgres Database on EC2. Successfully connected the Node Express and to the EC2. Modified the configuration several times to make sure everything was connected. Tested the Docker Containers locally and made sure everything was running. I ended up SPLITTING the postgres and node containers into two images, with different docker-compose files. THEN I uploaded them to Docker HUB, and pulled them down to AWS EC2.

For Postgres, I installed the postgres ubuntu command line software. I set up the ubuntu user and password. I tested the connection in pgAdmin on my local machine. I started the postgresql service on the Ubuntu EC2 machine and connected to it via the public IP.

The problem was a tangled mess with the configuration, not enough separation of concerns. It was very difficult to tell what my error was, whether something had to wrong password, postgres didn't have the user created yet, the SQL schema hadn't been loaded, the EC2 ports weren't configured, maybe the Docker wasn't running on localhost anymore to connect, maybe the Docker configuration/compose was wrong. Between Debugging, rebuilding the Docker containers, trying to fix/relaunch the EC2 instances, and tearing down and restarting the Postgres database, it took a LOT of time to work through each issue.

For the Node, I built the image with a basic docker-compose file. I sent this image to DockerHub as well.

Once I had a clearer understanding of separating the containers, it was a milestone to get everything deployed.

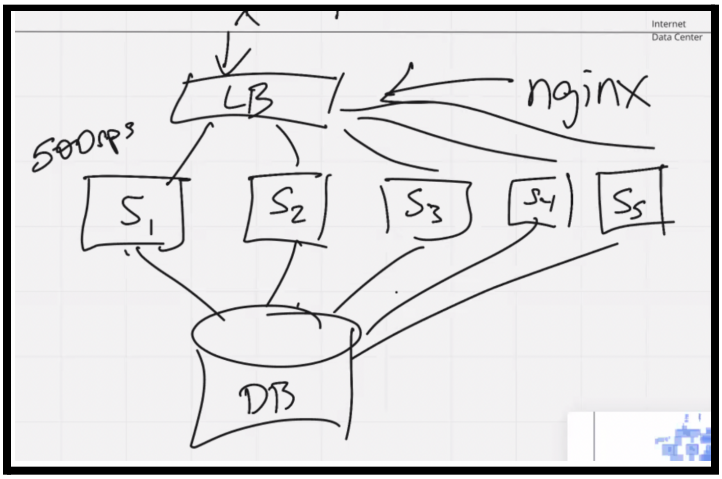
Read this article to improve the Docker Compose File:
Speed Goal: 100rps throughput, 2000ms latency, >1% error rate

https://codewithhugo.com/node-postgres-express-docker-compose/

https://betterprogramming.pub/containerize-node-react-postgres-with-docker-on-aws-ca548595f01e

Read about a load balancer:
https://levelup.gitconnected.com/nginx-load-balancing-and-using-with-docker-7e16c49f5d9

Set up a Server, and Database
Create a Load Balancer with NGINX
HORIZONTAL SCALE: Create more services until creating another one does not improve the performance
THEN tweak/optimize, and horizontal scale again



Day 9:

Today I was able to successfully deploy a Node Docker container on an EC2 instance. This could connect to the second EC2 instance. The second EC2 instance was running a Postgres database and exposing the PORT 5400. By using a postgres connection string with a matching username/password, I could query the postgres database from any node application.

Now that everything was deployed, I also had a workflow to update the code and redeploy everything through Docker Hub quickly. Simply rebuild the Docker image, push to Docker Hub, SSH into the EC2, stop running the current docker container, pull the new image from Docker Hub, and run the container again.

It was time to optimize my database now. I started off by running local load testing with the k6 tool. I read through this [article](https://medium.com/@gururajhm/load-testing-api-with-postman-node-js-k6-8fa04725e3b4) for some tips. Using Postman, a tool for making API requests, we can write simple assertions for a given URL request. For this project, we will expect a 200 response code for a database READ at each endpoint. We also need to make sure each endpoint responds in under 2000ms. I created a collection of API endpoint calls each with tests in Postman. I exported this collection to a json file. Then I transpiled the postman export to a k6 usable script.

`postman-to-k6 Capsules_Testpostman_collection.json –iterations 25 -o k6-script.js`

`k6 cloud k6-script.js`

With the k6 script, I could run and test the performance of each endpoint. I then connected the k6 script to the cloud service with a simple addition to the configuration object and a TOKEN. That allowed me to view the test result in the web portal.

Let's run a k6 load test on our API. This first test below is hitting `http://59.10.35.290/products/999999/styles` which will fetch from another IP address (eg. `98.10.76.490)` where the database is stored at.

URL	METH...	STATUS	COUNT	MIN	AVG	STDDEV	P95	P99	MAX
✓ http://52.35.89.230/products/1	GET	200	1	306ms	306ms	0ms	306ms	306ms	306ms
✓ http://52.35.89.230/products/	GET	200	1	118ms	118ms	0ms	118ms	118ms	118ms
✓ .../products/700000/styles	GET	200	1	34 513ms	34 513ms	0ms	34 513ms	34 513ms	34 513ms
✓ .../products/999999/styles	GET	200	1	33 880ms	33 880ms	0ms	33 880ms	33 880ms	33 880ms
✓ .../products/700000/related	GET	200	1	1 354ms	1 354ms	0ms	1 354ms	1 354ms	1 354ms
✓ 52.35.89.230/products/454755	GET	200	1	309ms	309ms	0ms	309ms	309ms	309ms
✓ .../products/999999/related	GET	200	1	502ms	502ms	0ms	502ms	502ms	502ms
✓ 52.35.89.230/products/999999	GET	200	1	890ms	890ms	0ms	890ms	890ms	890ms

NAME	SUCCESS RATE	SUCCESS COUNT	FAIL COUNT
✓ Status code is 200	100%	8	0
✓ Response time is less than 2000ms	75%	6	2

Here we test 8 different URL requests. One for each category of this service (styles, related, product info). I also added a request for a product in the last 10% of the data set. I may take longer for the database to query data at the end of the table.

We can also see that only 6/8 endpoints pass the test. This is because the `/products/999999/styles` and `/products/700000/styles` are taking over 30000ms. This chart helps show EXACTLY what endpoints we should focus on optimizing.

Next, it was time to work on implementing the NGINX Reverse Proxy and Load balancer. We will keep the EC2 Postgres Database deployed. We will have Node servers connecting to this database to serve the RESTful API endpoints. If many clients are connecting to the API, we can scale our servers horizontally, deploying MORE Node Servers available to process client requests. We do not create more instances of the database yet. NGINX will handle the expansion of Node servers to optimize the API routes. NGINX is a manager at the supermarket, who opens NEW checkout lanes to meet customer demand along with directing customers to open lanes. The manager also closes and removes/re-assigns cashiers when customers are

absent.
https://ashwin9798.medium.com/nginx-with-docker-and-node-js-a-beginners-guide-434fe1216b6b

Next, I ran a load test on my Deployed application. With a nginx docker image, and a node docker image running on my EC2 instance, I could run more tests with https://loader.io/.

Postgres Database Improvement Result:

Situation:

I built a Node Express API with several endpoints. I have retail product data stored in a PostgreSQL database. I am connecting the Node server to the Postgres database and querying data. Most of the endpoints in the database are responding with under 2000ms latency. However, the `/products/999999/styles` endpoint returns data, which will query a list of style data from the styles table, and a list of sku data from the skus table AND list of photos data from the photos table for EACH style. This is a huge problem.

```
I am making a request for product ID 999999 because it is in the last 10% of the dataset. This could rule out one problem, if the Postgres Database is having to read through ALL of the rows until it finds product ID 999999. This is a HUGE problem because scaling the database or adding millions of new entries would cause this query time to increase exponentially.
```

When I run 'localhost:/products/999999/styles` with Postman locally, I am getting 4000ms response times. This is well over the 2000ms project cutoff. This query is taking 30 seconds in production.

More details: even after making a request to the `localhost:/products/1/styles` in the first 10% of the database, it did not respond any faster. This is because the photos for example, could be anywhere within the last or first 10% of the data set, so too variable overall.

After running the tests again with k6 with more (50 max Virtual Users) VUs, I was able to get 6 requests per second. This ran across all endpoints at once.

Action:

a. The Node Server Sequelize ORM was using Promise.all to query multiple tables to aggregate a list. I modified the sequelize.query to use an `include` array and properly generate a join command. I also worked with excluding fields, and changing query object options keys (like "separate: true", and "raw: true") to improve the Sequelize query.

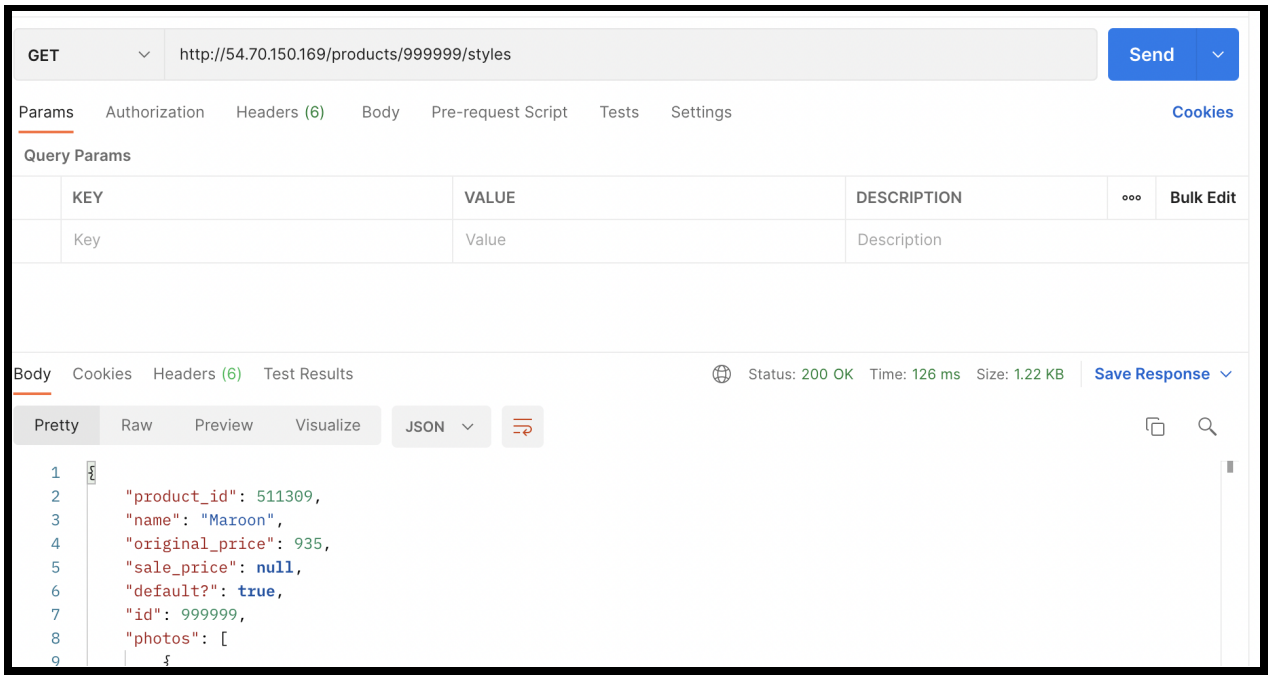
b. Before the nested query, I was passing ALL nested photos and sku objects through a formatter function. The database did not return the data in the format of the API documentation. This could adhere to performance slightly, but it probably wasn't the issue. I was able to modify the Sequelize nested query to only include valid fields, and I was able to return ONLY the exact date expected. Hell yeah
c. Now Node created ONE nested SQL query string, basically returning a JOIN table from the API. No more separate queries with promises, since that would be x20 times the number of requests to the database.
d. Next, I ran the ANALYZE command in the postgres database. [DOC](https://www.postgresql.org/docs/9.2/populate.html)
e. Next, I exported the SQL query that Sequelize generated when requesting the `/products/999999/styles` endpoint. I ran this query IN the PostgreSQL CLI myself, to see the performance. Without even timing the query, it was VERY CLEAR that it was taking over 20 seconds. This was the issue. The API could only respond as fast as the database, and I found the problem.
f. Added INDEXes to my PostgreSQL database. Every relevant table had the style_id already saved in the SQL database. However, the PostgreSQL database did not know how to query this optimally. That was easily solved with an index. Since the style_id is known, it should be a constant time query searching in the photos or skus table. WITHOUT the index in the SQL database, it was searching through EVERY row of the table EACH time we queried the data. This was making it slow.

Result:

Success! After optimizing the Sequelize query and adding the SQL index to the database, I can fetch this endpoint normally. From 30000ms to 150ms, that is a 20,000% improvement!

None of the other queries were taking long to respond. That is because they request a pre-defined product id, which is a constant time operation for the SQL query. The styles query was the only one fetching from multiple tables. That meant, it would be searching through ALL photos table entries, and ALL sku table entries.

The first step was to optimize the Sequelize query. Once I refactored the nested query, I could rule out that the Node ORM request was NOT the bottleneck/problem.



INDEX Commands inside the psql PostgreSQL command line interface

```
CREATE INDEX products_id_index ON product (id);
CREATE INDEX skus_style_id_index ON skus (style_id);
CREATE INDEX skus_style_id_index ON skus (style_id);
CREATE INDEX photos_id_index ON photos (id);
CREATE INDEX photos_style_id_index ON photos (style_id);
CREATE INDEX styles_style_id_index ON style (id);
CREATE INDEX styles_product_id_index ON style (product_id);
```

Day 10:

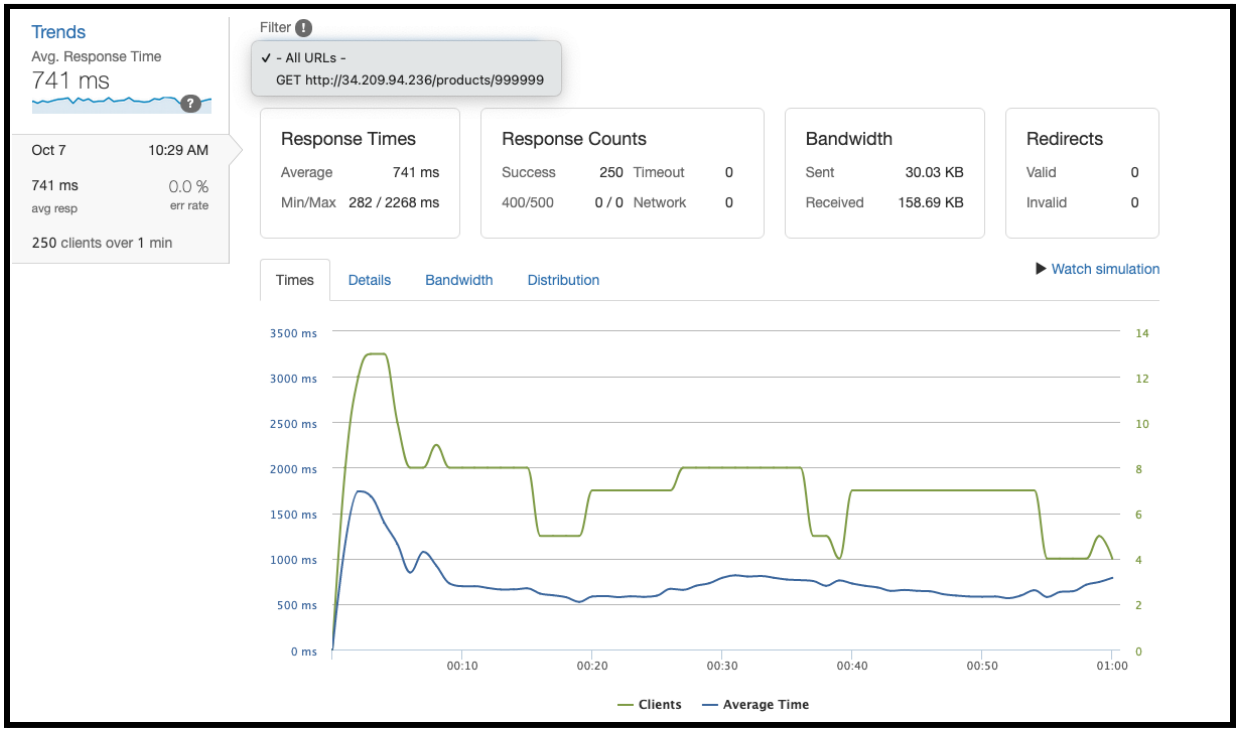
With everything deployed, it was time to stress test my EC2 servers with Loader.io. This will send a TON of requests to the server based on the given endpoint URL, and we can increment the request load to see how it performs.

NGINX Load Testing Tests:

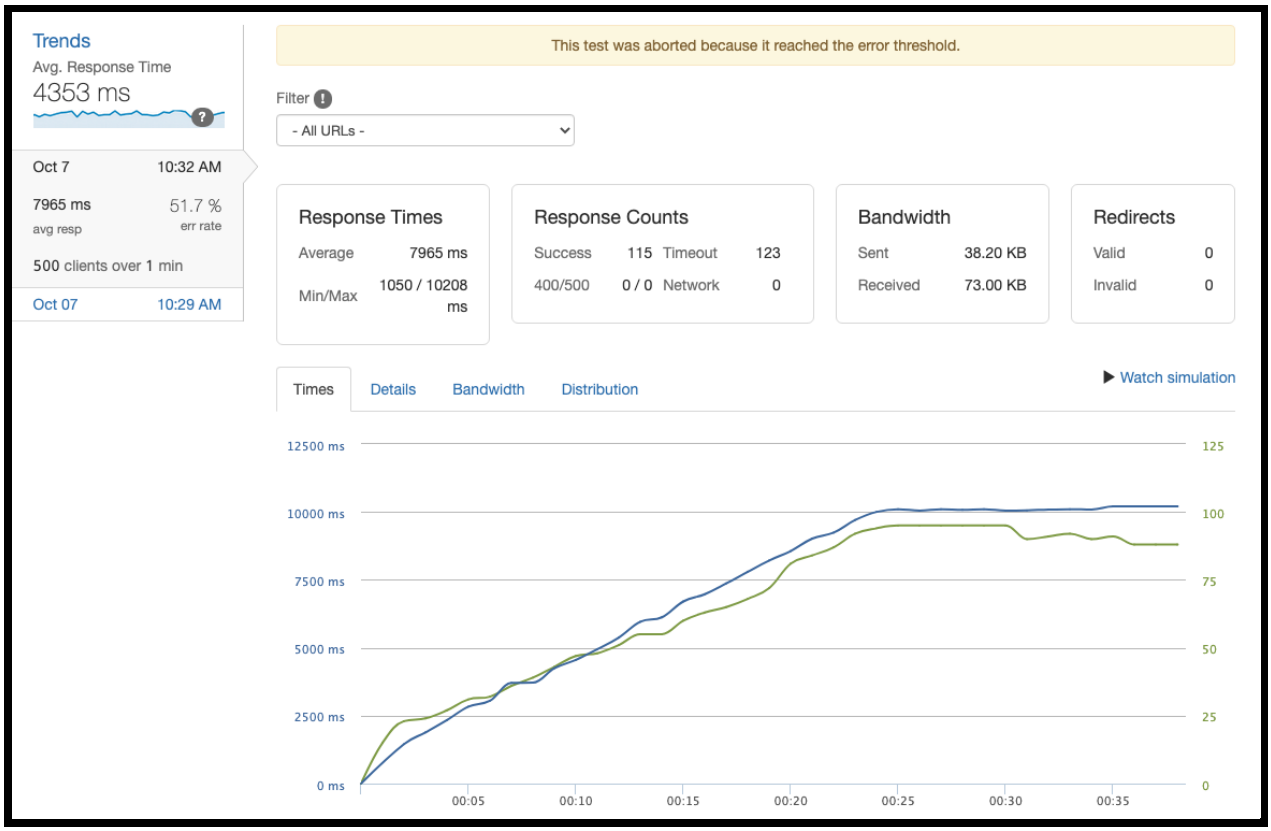
NO LOAD BALANCER:

First, I ran a load test on the /products endpoint on the EC2 Node RESTFUL Express app, which connects to the EC2 Postgres instance. No load balancer.

I tested the /products endpoint with 250 clients over 1 minute (4.3 Request Per Second). It returned 741ms latency.



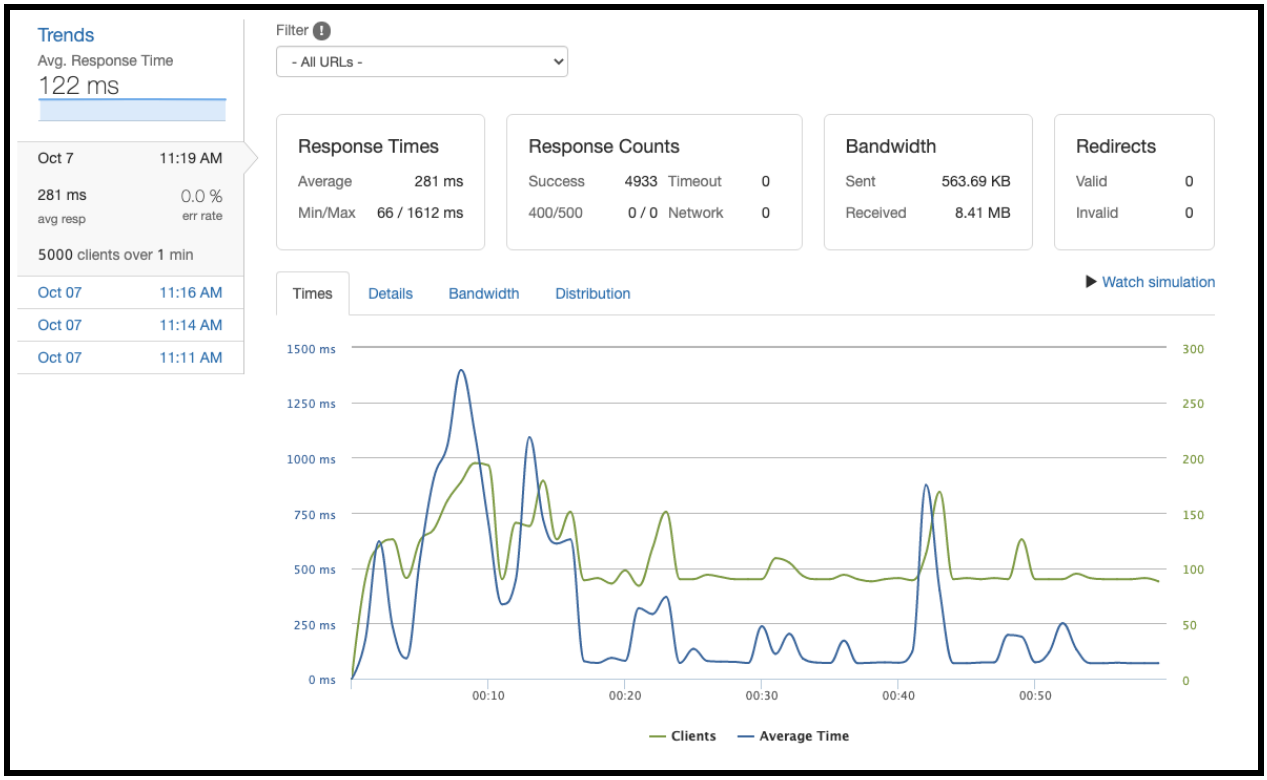
Next, I tested the same setup with 500 clients over 1 minute (8.6 Request Per Second). It returned 4353ms latency. This test failed because the server could not handle the traffic.



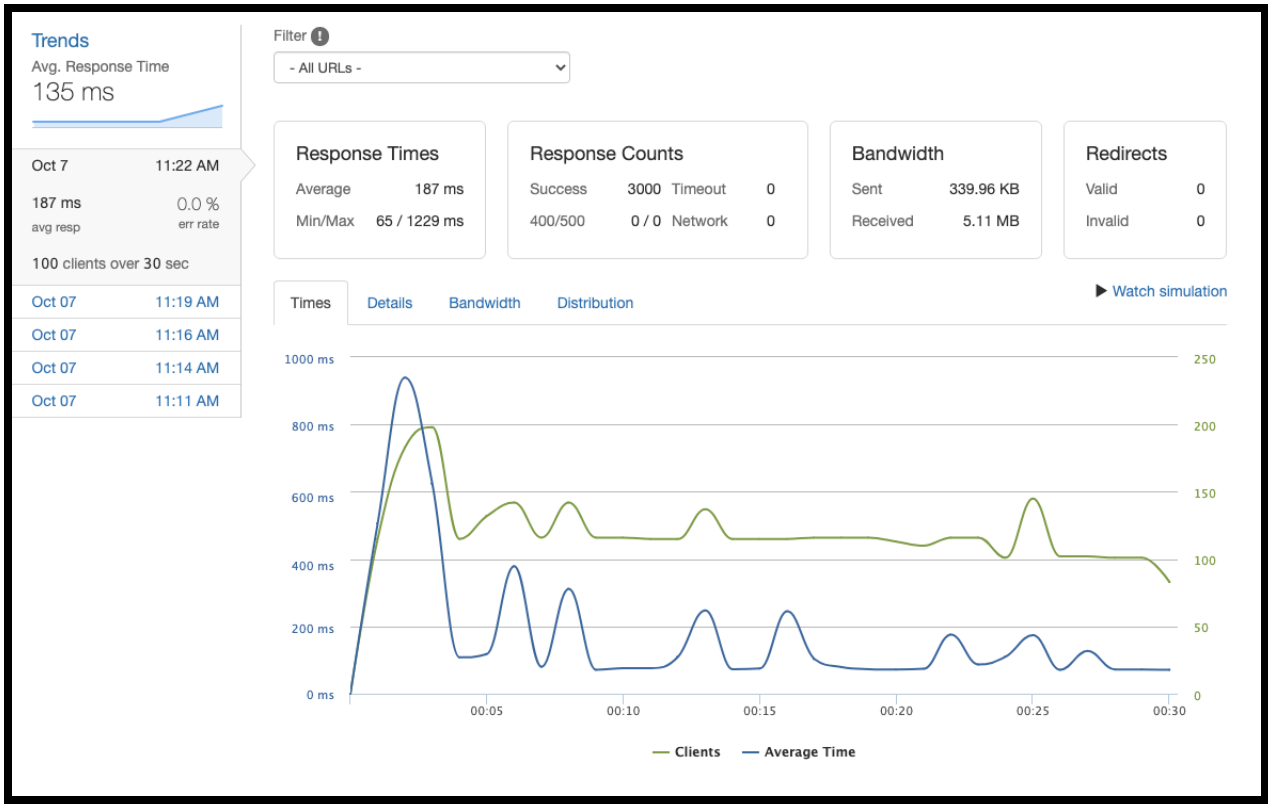
WITH LOAD BALANCER:

Second, I ran a load test on the /products endpoint on the NGINX Load Balancer EC2 instance, with fetches from the EC2 Node RESTFUL Express app, which connects to the EC2 Postgres instance.

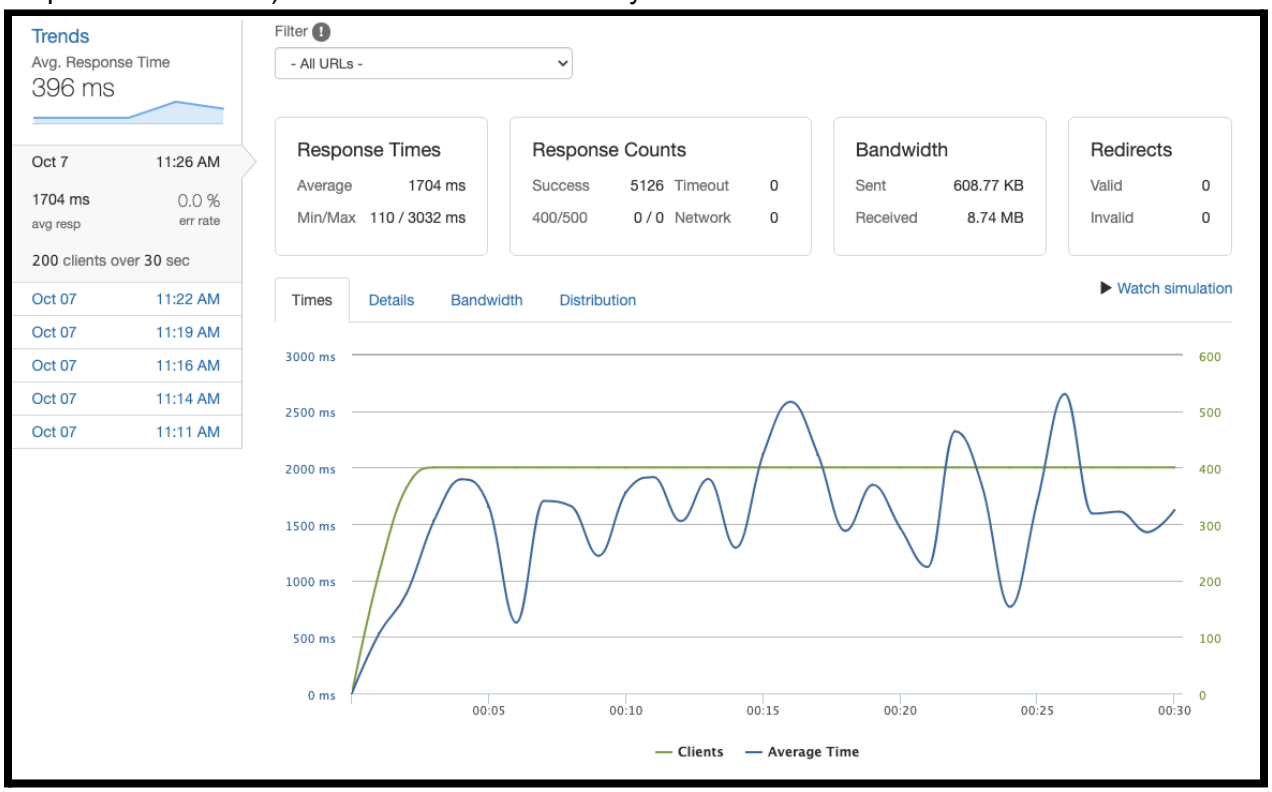
I tested the load balancer setup with 5000 clients over 1 minute (83 Request Per Second). It returned 122ms latency. This was a substantial performance improvement over the previous endpoint.



I tested the load balancer setup with 3000 clients over 30 seconds (100 Request Per Second). It returned 135ms latency.



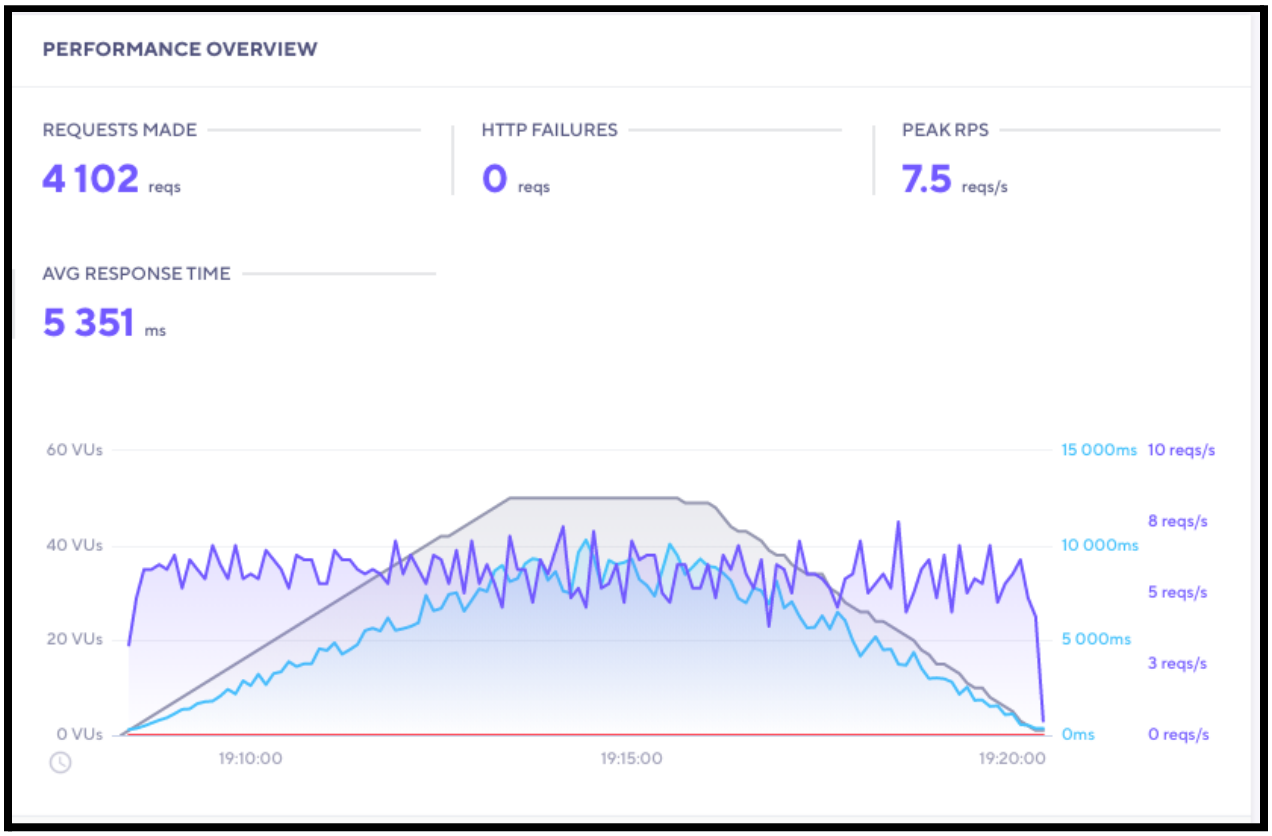
Looking good so far, let's test the load balancer setup with 6000 clients over 30 seconds (200 Request Per Second). It returned 1704ms latency.



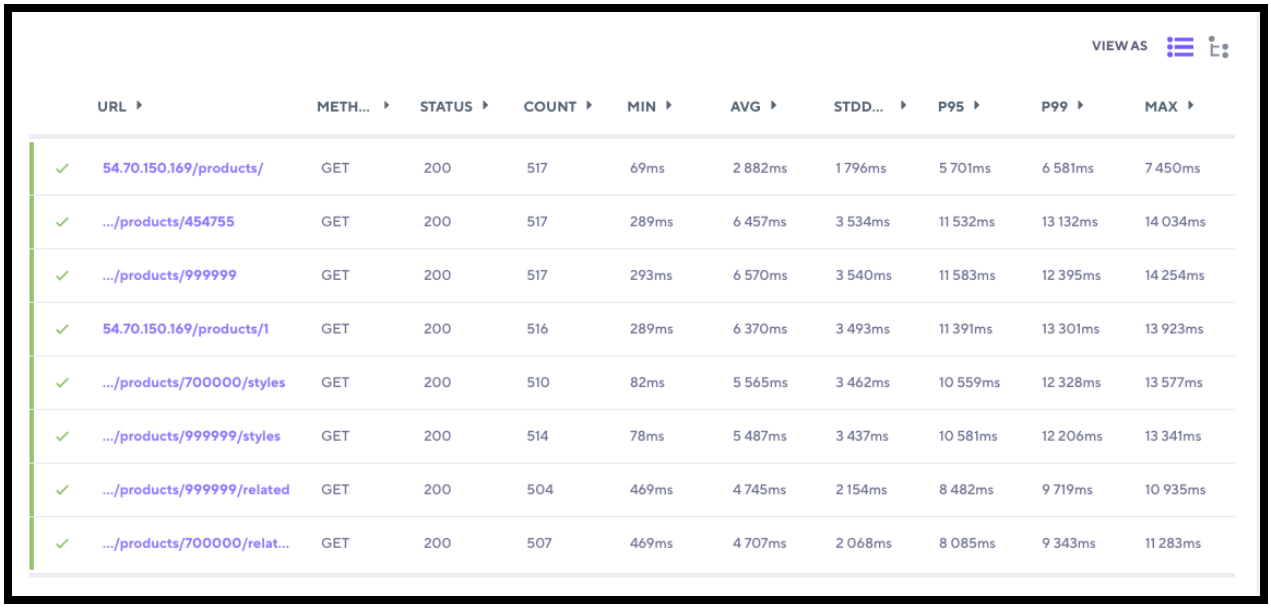
LOAD BALANCER Result:

After adding a deployed Load balancer, this drastically improved the throughput AND latency for the products endpoint. Going from a MAX 8.3 Requests per second with 4000ms+ latency, I was able to improve it to 200 requests per second throughput and 396ms latency. Note, all of these endpoints are able to maintain a <1% error rate. They all perform under 2000ms latency, and have well over 100 RPS.

Another k6 (cloud) stress test:
I ran the pre-written test for 8 different URLs on the server. This test ran 4102 requests over 10 minutes (avg 6 req per sec). It appears there were NO HTTP failures, which is great. The only problem is an average 5,351ms response time. Let's look at the breakdown chart for each endpoint.



Here we can see the average response time for EACH endpoint. This can help narrow down individual queries that are slow. It looks like most requests are beyond 2000ms.



VIEW AS

	URL	METH...	STATUS	COUNT	MIN	AVG	STDD...	P95	P99	MAX
✓	54.70.150.169/products/	GET	200	517	69ms	2 882ms	1 796ms	5 701ms	6 581ms	7 450ms
✓	.../products/454755	GET	200	517	289ms	6 457ms	3 534ms	11 532ms	13 132ms	14 034ms
✓	.../products/999999	GET	200	517	293ms	6 570ms	3 540ms	11 583ms	12 395ms	14 254ms
✓	54.70.150.169/products/1	GET	200	516	289ms	6 370ms	3 493ms	11 391ms	13 301ms	13 923ms
✓	.../products/700000/styles	GET	200	510	82ms	5 565ms	3 462ms	10 559ms	12 328ms	13 577ms
✓	.../products/999999/styles	GET	200	514	78ms	5 487ms	3 437ms	10 581ms	12 206ms	13 341ms
✓	.../products/999999/related	GET	200	504	469ms	4 745ms	2 154ms	8 482ms	9 719ms	10 935ms
✓	.../products/700000/relat...	GET	200	507	469ms	4 707ms	2 068ms	8 085ms	9 343ms	11 283ms

Check out one more data point for this load test. Each request will check for a 200 status code, and also a response time UNDER 2000ms. Only 17% of the requests were successful.

THRESHOLDS (1/1) | CHECKS (4.8K/8.2K) | HTTP (4.1K/4.1K) | ANALYSIS Compare metrics | SCRIPT View executed script | LOGS Execution logs

VIEW AS

	NAME	SUCCESS RATE	SUCCESS COUNT	FAIL COUNT
✗	Response time is less than 2000ms	17.84%	732	3.4K
✓	Status code is 200	100%	4.1K	0

Summary of what I deployed:

I installed postgres on one instance manually, uploaded a large backup.sql (a pg_dump export) to the EC2, and then loaded the data into THAT remote postgres. Then I always have ONE ip address to reference for the database

and then I created a Docker image for the Node/Express server, and uploaded it to docker hub. It pretty much just does ‘npm install’ and copies files over.

So on my second EC2, I only had to install docker.io, and then I ran docker pull spencerlepine/node-server, and docker run -p 3000:3000 spencerlepine/node-server

And then for nginx on the third, I also used docker, because I found an article to follow. So the public IP addresses of the Node and Database EC2 always stay the same, and then I am load-testing the third nginx EC2 instance.

without docker, I would need to install node, npm, git, etc, but I am a bit too invested to back out

of using docker at this point

Problem with not using the load-balancer, not an Elastic IP. Whenever I change the IP of an EC2 instance, I have to update the nginx configuration with that new IP each time. This can be a problem using hard-coded addresses, because if a developer made a mistake some service might break.

I was able to [duplicate](https://docs.bitnami.com/aws/faq/administration/clone-server/) the EC2 Image.

Notes:
Load balancer: https://dev.to/ehdtlaos/testing-loader-io-after-installing-nginx-7ob

http://api/products/9999999/styles

Here is a test for the Deployed api/products/9999999/styles. This will be in the last 10% of our data set and presumably take longer. We get an average response of 73ms with 4 RPS. With NGINX + Node fetching from the Deployed EC2 Postgres database. It can easily handle 4 requests per second.

This next test will send 200 Requests per second to the styles endpoint. We get 6000 successes. With NGINX + Node fetching from the Deployed EC2 Postgres database.

This next test will send 14,242 Requests to the styles endpoint over 30 seconds. We get 6000 successes. That reports a 474 request per second throughput, with 1783 latency.

Next, I ran 10,000 clients over 30 seconds, giving us a 65ms response time for 333 RPS throughput.

NEXT, I booted up another EC2 running JUST the node server without nginx. Now I can compare the load test. When I tried running 1000 RPS, it crashed the server.

Ideas: use Round Robin IP address for laid balancing, or least used algo

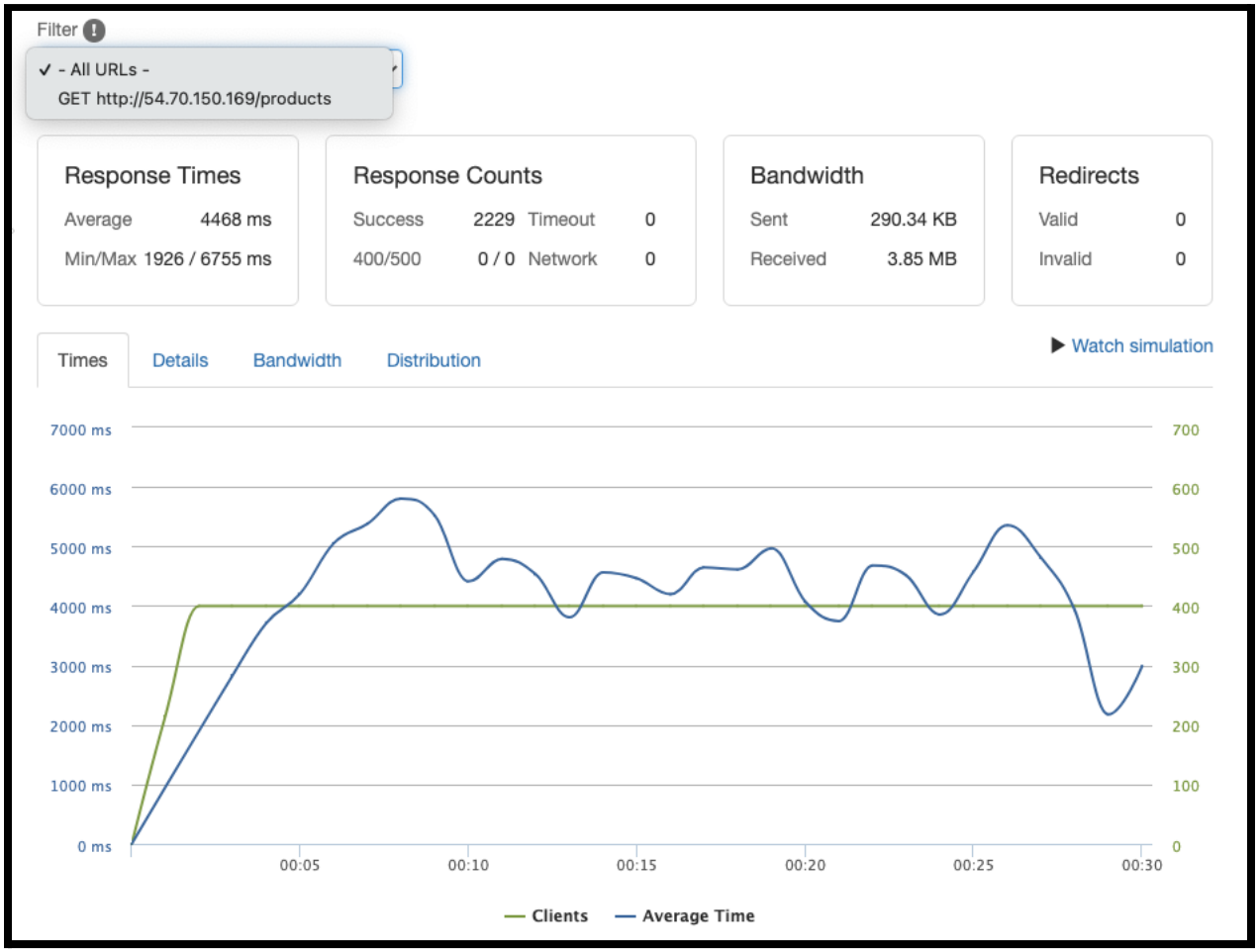
Day 11:

Today I was able to use AWS AMIs (Amazon Machine Images) for development. When scaling horizontally, my app needed another EC2 instance for the node server. Keeping one same EC2 load balancer nginx image and one Postgres Database EC2 instance, I was creating duplicate EC2 instances for Node. This meant I would need to install git, node, npm, and dependencies on EACH machine and hope everything worked. Also, I needed to keep track of the version of each software to make sure everything ran smoothly.

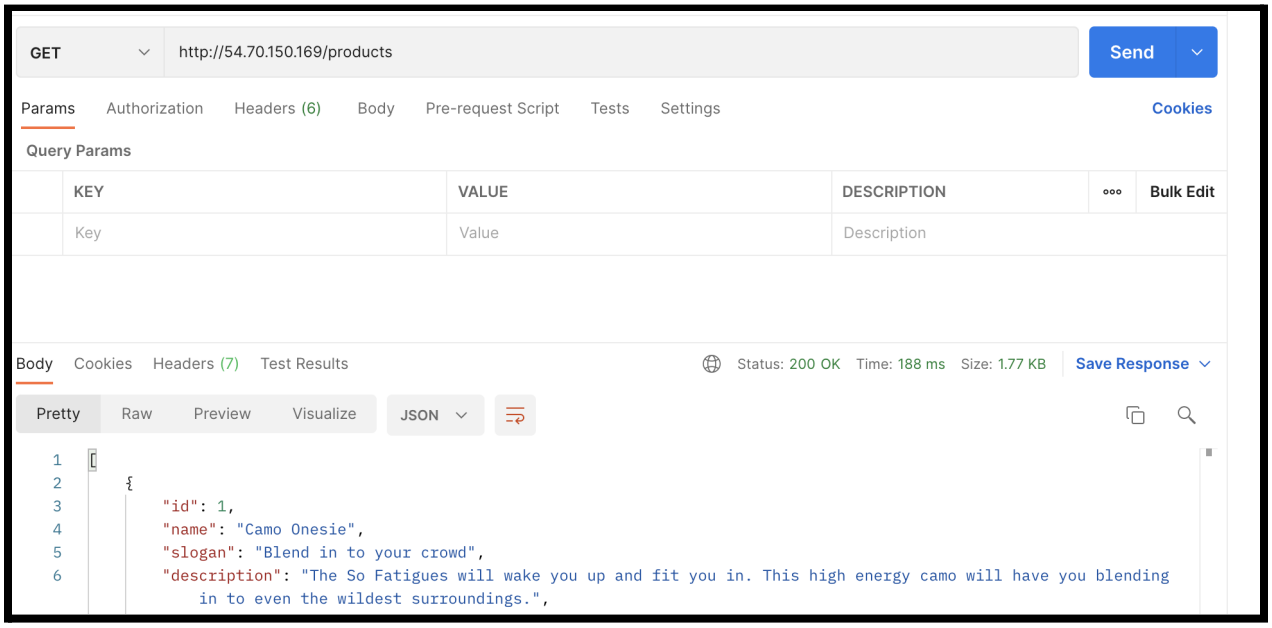
First, I used Docker containers to automate many repetitive tasks for starting up a node server. Next, I created the AMIs to spin up more EC2 instances for horizontal scaling. This was a reusable configuration that saved a lot of time.

Next, I ran the load tests with the load balancer and TWO EC2 server instances (horizontal scaling). It appears to have really slow response rates when running 200 clients per second. It appeared to have 0 failures though.

The load test FAILS when running 300 clients per second.



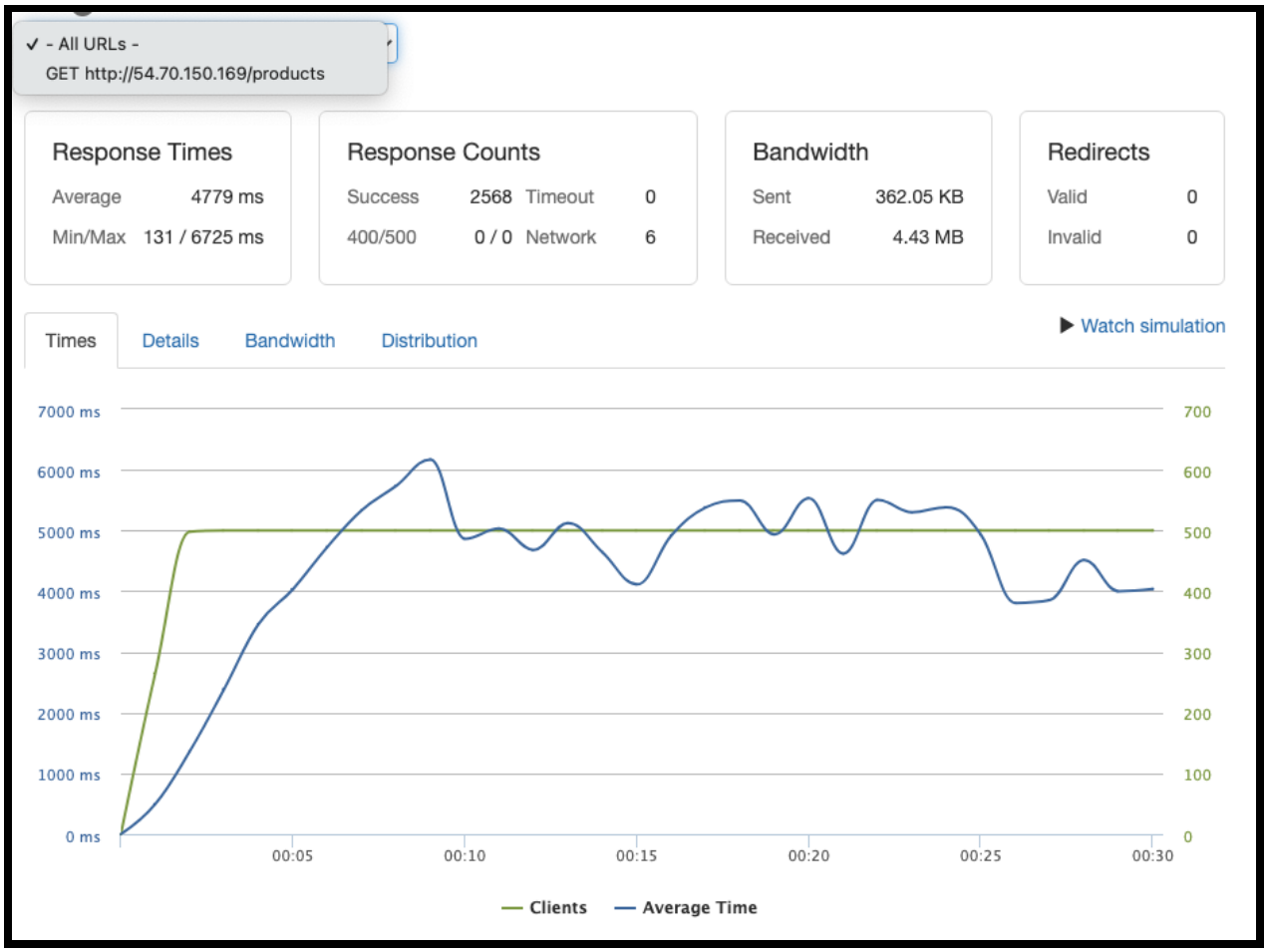
It appears to return <200ms response times for single requests in Postman. This means that the server cannot handle LARGE numbers of requests over 200 RPS.



This test below is 250 clients per second. The average response is 4601 ms. The two problems to fix are getting this under 2000ms, AND/OR extending the capacity.

Let's try running another Node EC2 instance, with 3 servers to balance between. We will test 250 clients per second, and work to get the latency time down.

[Loader.io test, 250 RPS, 4779ms latency]



Scaling horizontally does not seem to be making a difference. Let's try to identify the bottleneck in our system. Let's modify the nginx config to the .conf below:

```

```
upstream myapp1 {
   server 18.236.153.95;
   server 35.85.153.118;
   server 34.209.94.236;
}

server {
   listen 80;

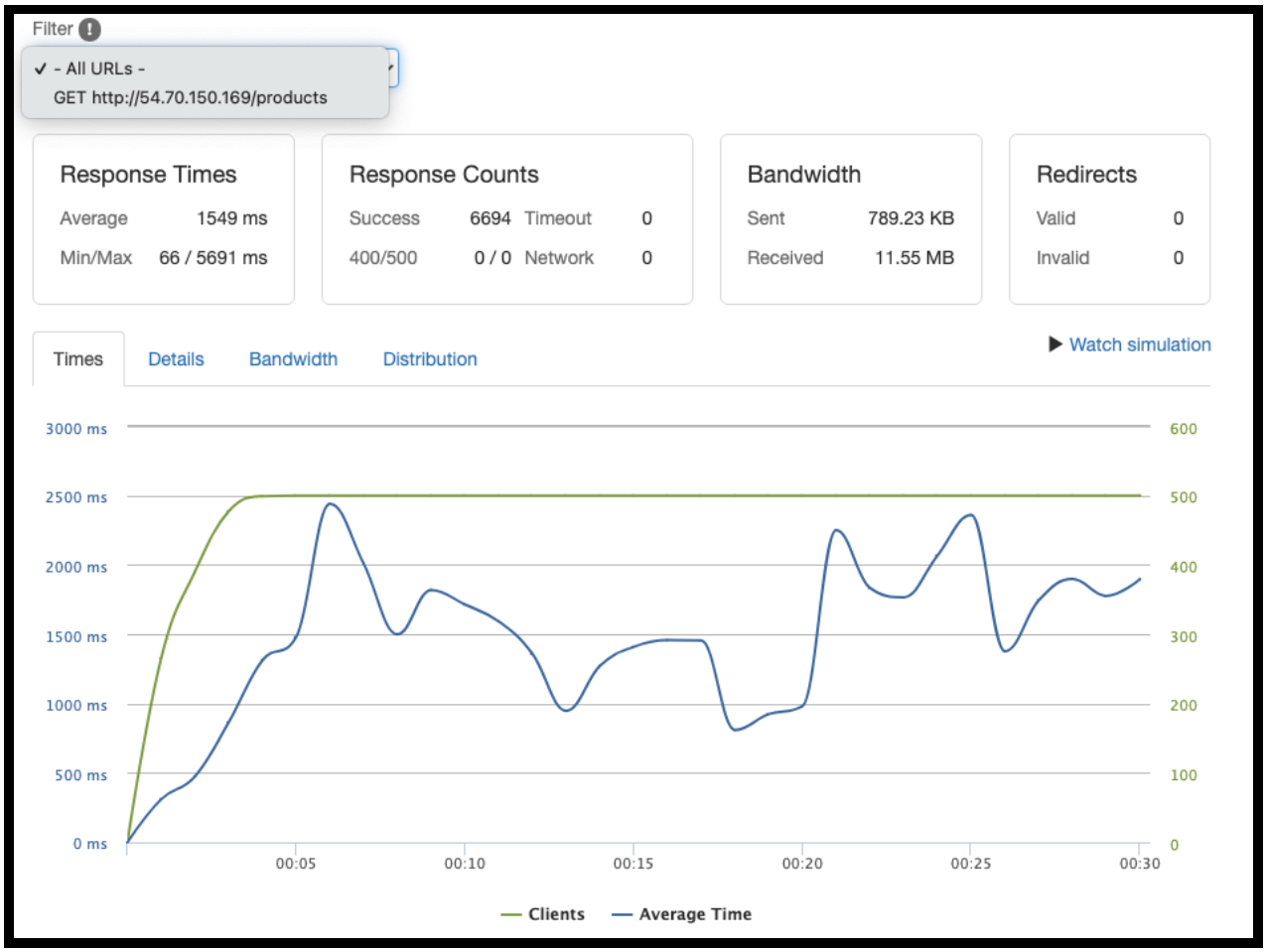
   location / {
       proxy_pass http://myapp1;
   }

   location /loaderio-e83d5e3095f6a92e5dace346247bd424 {
       return 200 'loaderio-e83d5e3095f6a92e5dace346247bd424';
   }
```
```

```
}
```

This setup will properly connect multiple instances and use the Round Robin algorithm for load balancing by default. After running the load test, it was able to handle 150 clients/requests per second with an average latency of 2596ms. Much better.

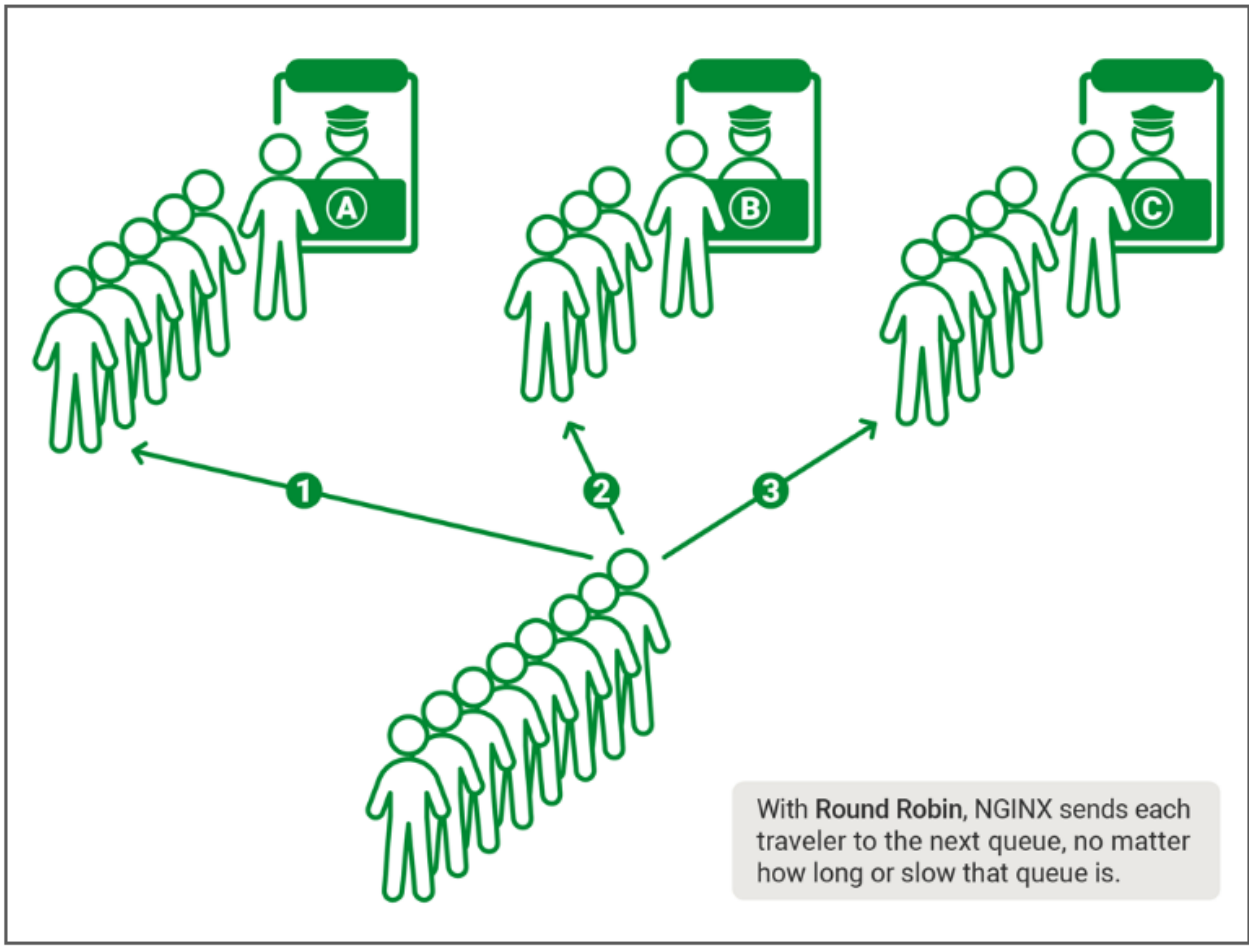
Let's try running the load balancer between FOUR EC2 instances now, scaling horizontally. Hell yeah, now we've got 1549ms latency (still not ideal) and 223 RPS throughput.



We could keep scaling horizontally, but let's work on some other optimizations.
Let's try running the load balancer with a Least Used IP algorithm between the four instances. YOO. Below we see an average response time of 103ms latency and ALL 250 requests (per second) throughput were successful. This is a HUGE improvement.

Before moving on, I should explain WHY the least used IP address algorithm was such an improvement. First, I will go over what the original/default Round Robin load-balancing alg did. Checkout this [post](https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm/) for more details.

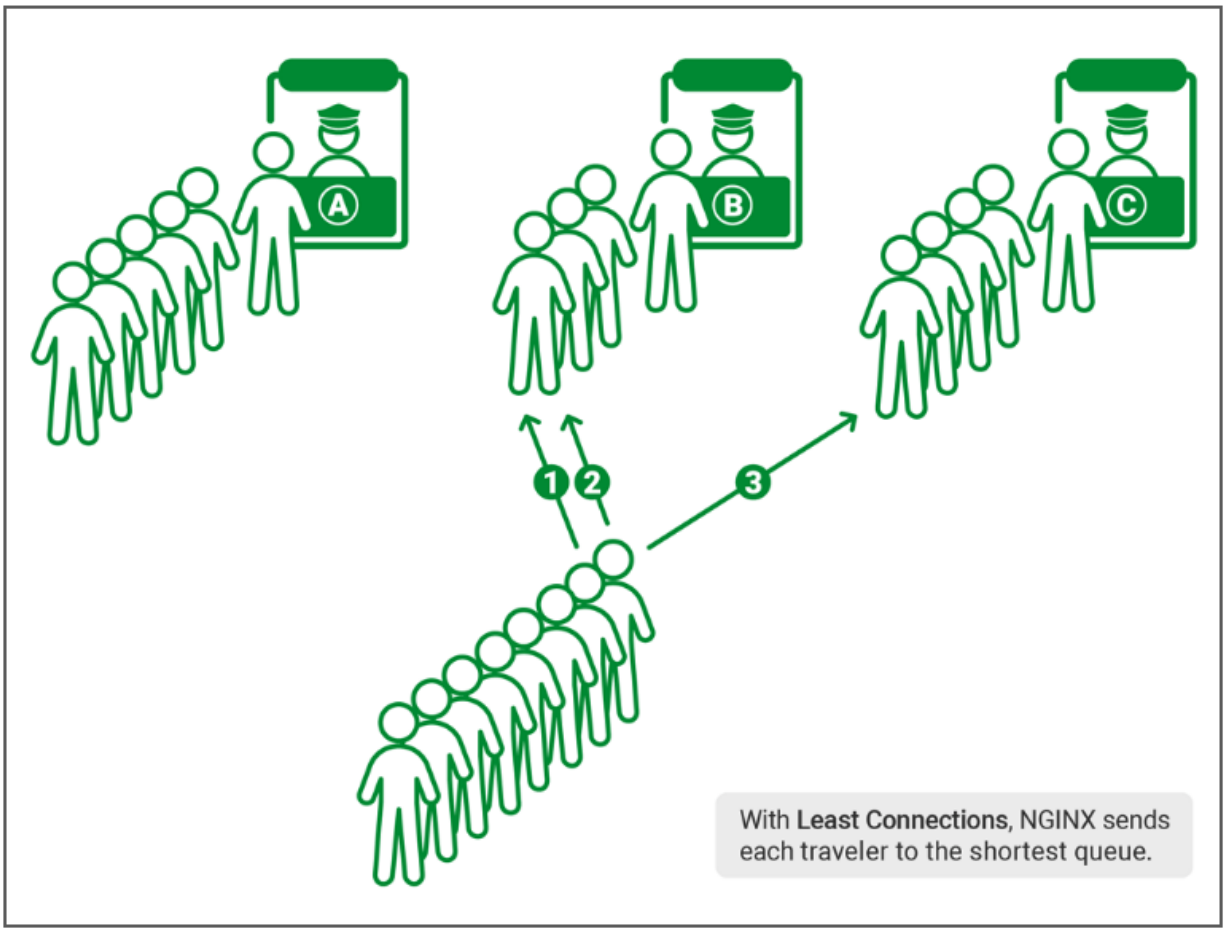
Round robin is a naive approach to load balancing. In this approach, the guide selects each queue in rotation – the first traveler is directed to queue A, next traveler to queue B, and so on. Once a traveler is directed to the last queue, the process repeats from queue A. Round Robin is the default load-balancing algorithm used by NGINX:



Those naive algorithms use a queue. If ONE station/Node API instance happens to be slow, the backlog gets longer and longer in that queue. Since this product's API can vary in times depending on how LARGE the response data is (e.g. if it has many images, or only two), we want to adapt to a slow response. This needs to be approached to deal with ANY endpoint, and we cannot tweak/hard code just a certain request. That brings us to load balancing with the Least Connection algorithm:

Least Connections Load Balancing

There's a much better approach. The guide watches each queue, and each time a traveler arrives, he sends that traveler to the shortest queue. This method is analogous to the Least Connections load-balancing method in NGINX, which assigns each new request to the server with the fewest outstanding (queued) requests:



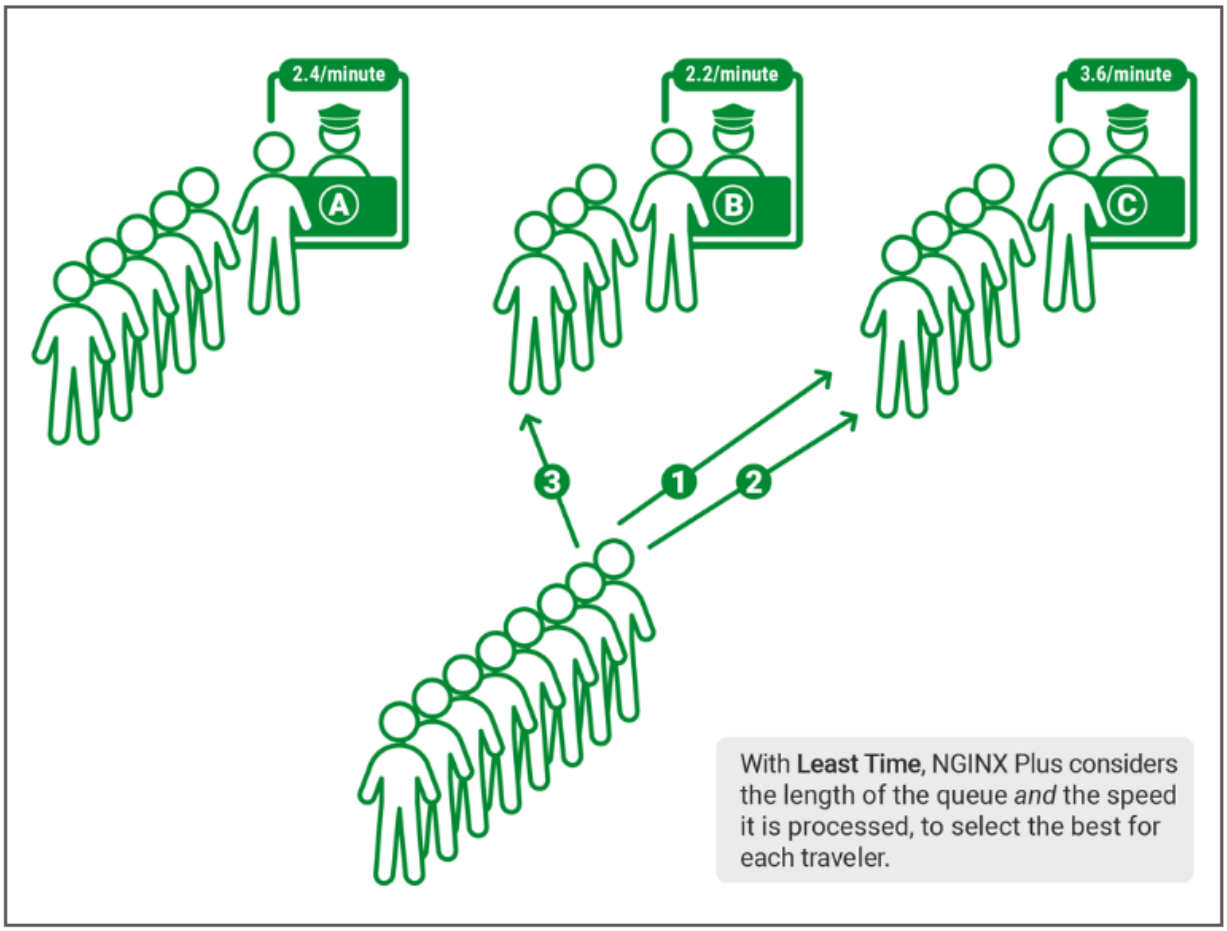
“Least Connections load balancing deals quite effectively with travelers who take different amounts of time to process. It seeks to balance the lengths of the queues, and avoids adding more requests to a queue that has stalled.”

ALSO it may help to mention the idea of Least time load balancing. Only NGINX Plus uses this, so we will stick to least used IP.

Think of the analogy of Airport customs. Multiple stations with workers on a computer, processing a line/queue of travellers. A traveller may be slow/fast if they have paperwork ready. ALSO, the station itself may be slow or fast if they have computer issues. Maybe the Node API Server EC2 is performing slow in general. If the load balancing takes this into account, we can adapt to slow performance of the Hardware itself. Some request’s may take more memory retrieving or serializing data.

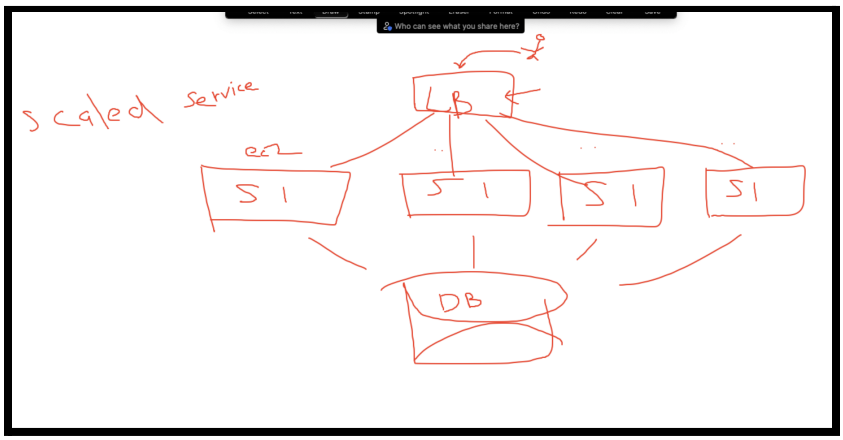
What if each immigration booth has a counter above it, indicating how many travelers have been processed in, for example, the last 10 minutes? Then the guide can direct travelers to a queue based on its length and how quickly it is being processed. That’s a more effective way to distribute load, and it’s what the Least Time

load-balancing algorithm in NGINX Plus does:



This algorithm is specific to NGINX Plus because it relies on additional data collected with NGINX Plus's

Nginx mentions that “Least Connections” is ideal for a single load balancer setup like we have here. “Classic load-balancing methods such as Least Connections work very well when you operate a single active load balancer which maintains a complete view of the state of the load-balanced nodes.”



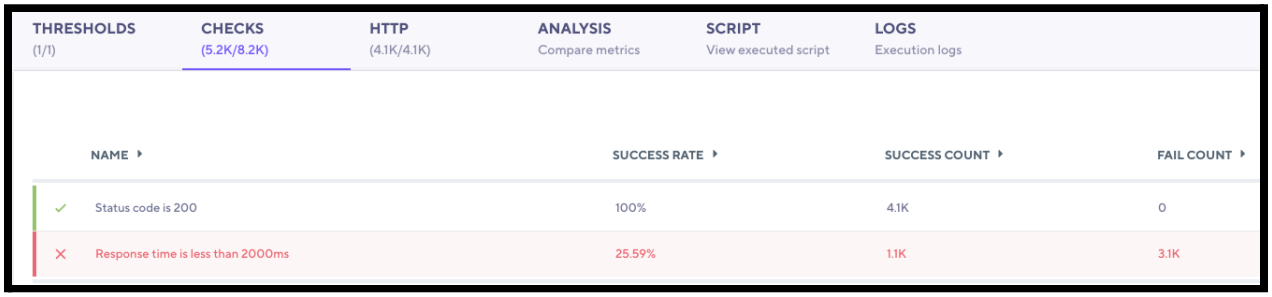
For comparing Round Robin and Least Connected, imagine a Grocery store checkout where customers exit the exact lane specified by a manager. If we sent customers to each checkout lane sequentially, 1,2,3, until 10, and back to 1 again, this system could work. Maybe it distributes customers just enough. There are problems with this however. Imagine one customer bought 400 different items. Or imagine the store gets super busy, and some lines get backed up due to cashiers being slow. The checkout lanes could be uneven and customers would complain about not getting through as fast as another lane.

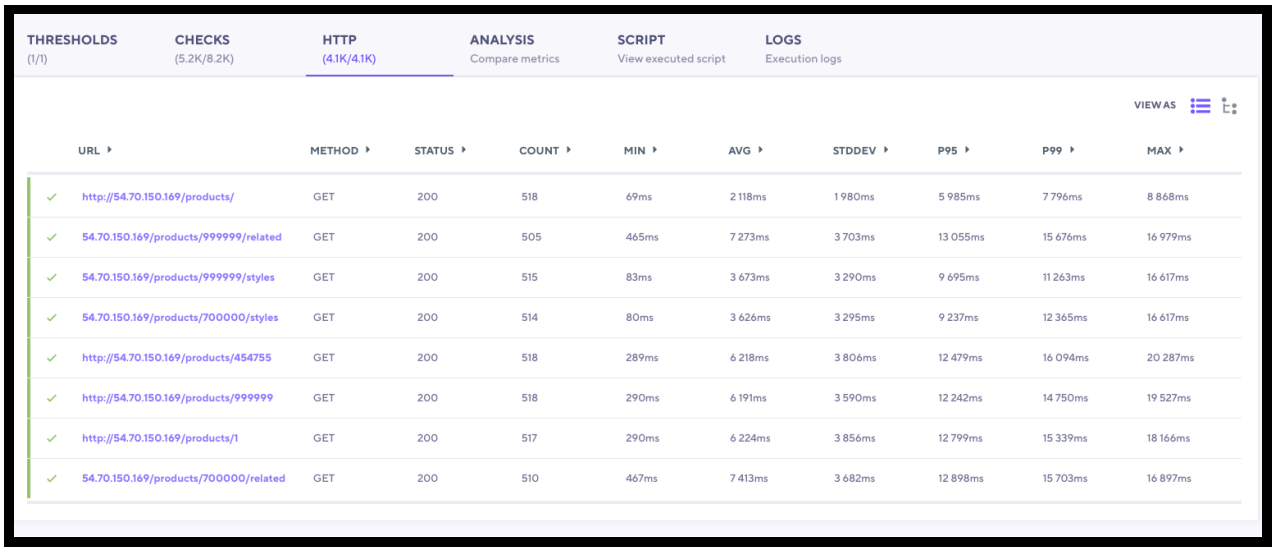
In comes the LEAST CONNECTED load balancing algorithm. This would be similar to a checkout system where we direct and distribute customers dynamically across all checkout lanes. Whenever we see the QUEUE of one line getting long, we look for the currently shortest line instead.

Now let's connect this analogy to our application. We have a single load balancer (the store manager), which distributes customers (http requests) to optimal checkout lanes (the Node servers). We could cycle through each IP address with a naive Round Robin load balancing algorithm. If any machines are slower than others, that QUEUE could be much longer (increasing latency), but we ignore this. We could also dynamically redirect requests to available IP addresses with the Least Connected load balancer algorithm. Now, we look at the queue for each machine, and send the request to the shortest one. This optimizes the request queue across ALL of the servers we are using. This allows an efficient USE of the servers. Without this, horizontal scaling would not improve the API performance much, and it could have varying results with a naive alg.

Let's run a k6 load test across multiple endpoints to see some overall performance stats:
Here we can see a huge improvement from the earlier k6 cloud test. Each request will assert a <2000ms response time, and the test returned **5.2K/8.2K** successes or **25.59%**. This is slightly better than the 17% success rate earlier. (some endpoints are under 2000, which is good, we can see we should worry about THESE n endpoints that are pretty slow).

The third chart below shows an average throughput of 6 Requests per second when hitting all 8 endpoints at once. This could be for many reasons (any number of bottlenecks), but the takeaway is that the API is not quite production ready. HEAVY loads over a longer test period show the system slowing down.



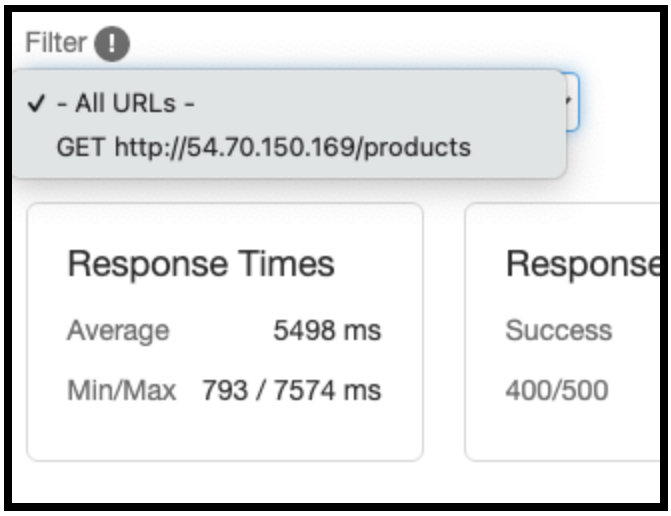


THRESHOLDS (1/1) | CHECKS (5.2K/8.2K) | HTTP (4.1K/4.1K) | ANALYSIS Compare metrics | SCRIPT View executed script | LOGS Execution logs

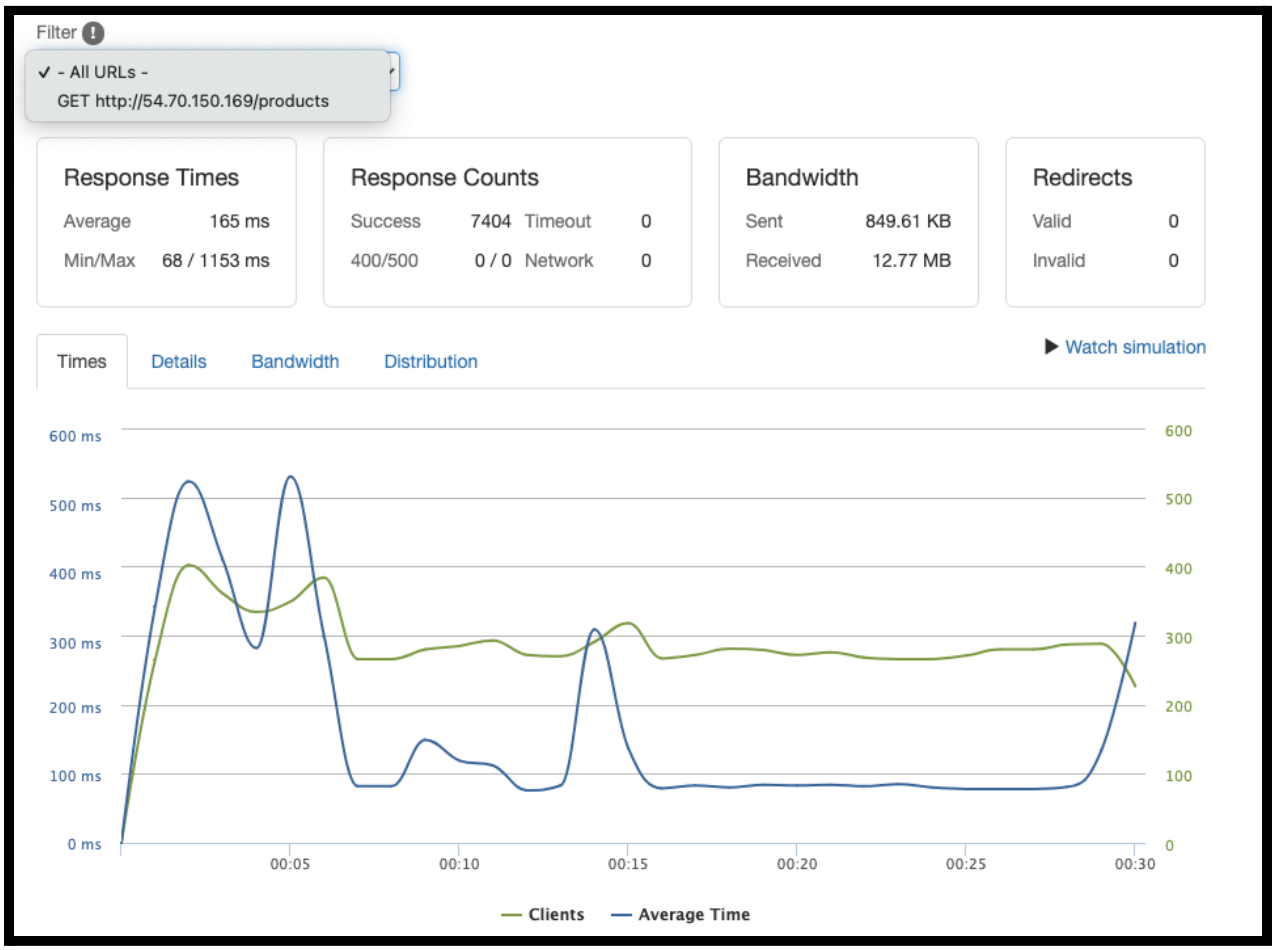
VIEW AS

URL	METHOD	STATUS	COUNT	MIN	AVG	STDDEV	P95	P99	MAX
http://54.70.150.169/products/	GET	200	518	69ms	2 118ms	1 980ms	5 985ms	7 796ms	8 868ms
54.70.150.169/products/999999/related	GET	200	505	465ms	7 273ms	3 703ms	13 055ms	15 676ms	16 979ms
54.70.150.169/products/999999/styles	GET	200	515	83ms	3 673ms	3 290ms	9 695ms	11 263ms	16 617ms
54.70.150.169/products/700000/styles	GET	200	514	80ms	3 626ms	3 295ms	9 237ms	12 365ms	16 617ms
http://54.70.150.169/products/454755	GET	200	518	289ms	6 218ms	3 806ms	12 479ms	16 094ms	20 287ms
http://54.70.150.169/products/999999	GET	200	518	290ms	6 191ms	3 590ms	12 242ms	14 750ms	19 527ms
http://54.70.150.169/products/1	GET	200	517	290ms	6 224ms	3 856ms	12 799ms	15 339ms	18 166ms
54.70.150.169/products/700000/related	GET	200	510	467ms	7 413ms	3 682ms	12 898ms	15 703ms	16 897ms

Let’s try running the load balancer with a nginx caching between the four instances. Here is the loader.io result: Hm, looks were getting 5000ms+ with caching. What



I did another load test after removing the cache, and this request was getting much better speeds.



I REMOVED the caching because it was not set up correctly. I kept scaling horizontally, and ended with 5 EC2 instances. At this point, we could consider scaling each server vertically, but

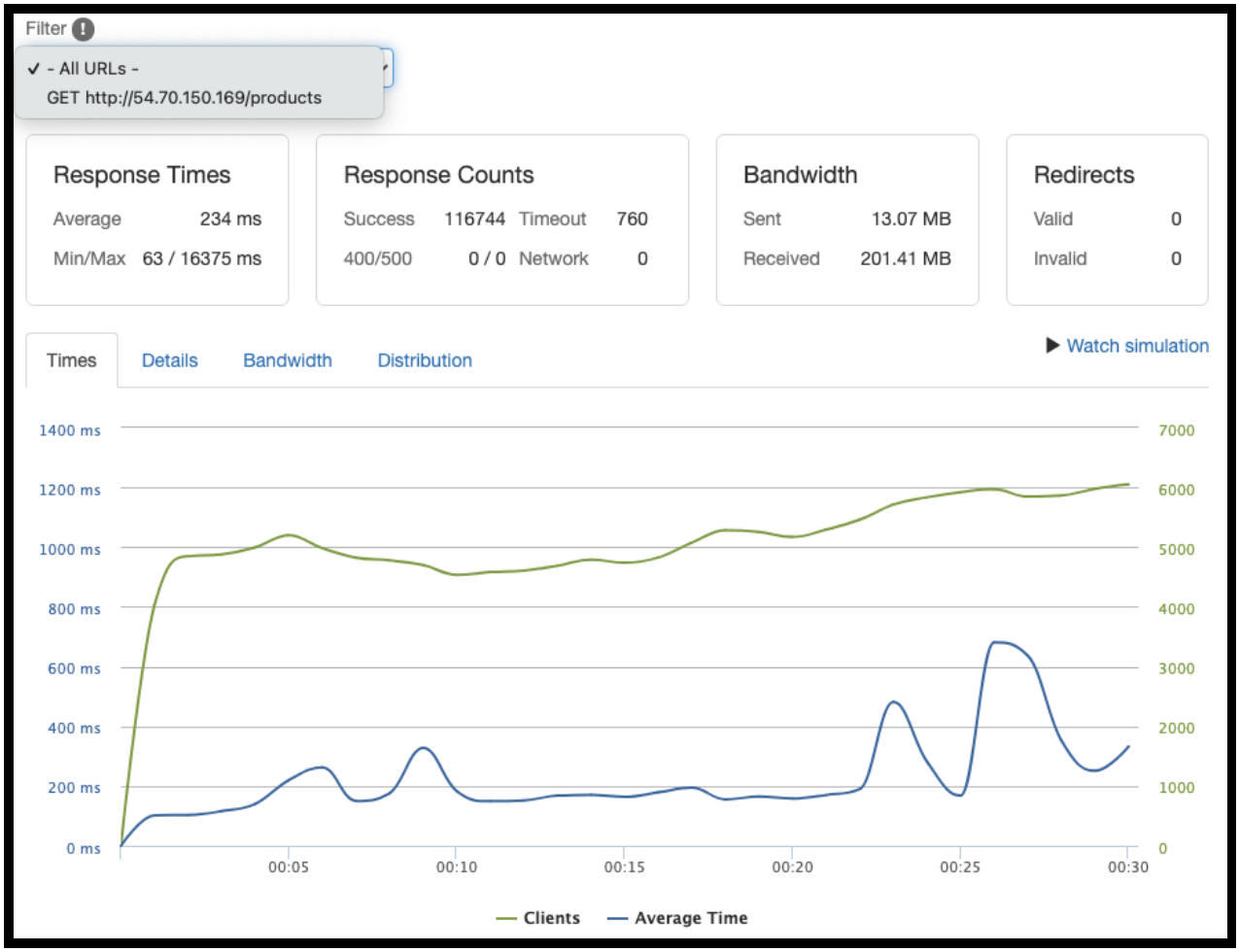
some bottleneck perhaps, or the limitations of horizontal scaling have been hit. Here below is the result of running 5 EC2 instances easily handling 250 requests per second on one endpoint.

Below is a test with CACHING, giving 64ms latency on 250 clients per second. This speed is MUCH better. I modified the nginx config cache properly this time. This shaved off 20ms.

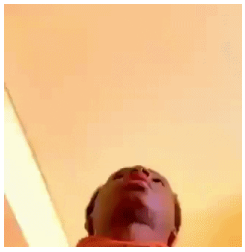
Next, let's run another loader.io load test with 500 RPS. This was a breeze with 65ms latency (on this endpoint) and 15,000 successful responses over 30 seconds. This is with NGINX load balancing 5 different APIs, caching, and using the least connected load balancing algorithm.

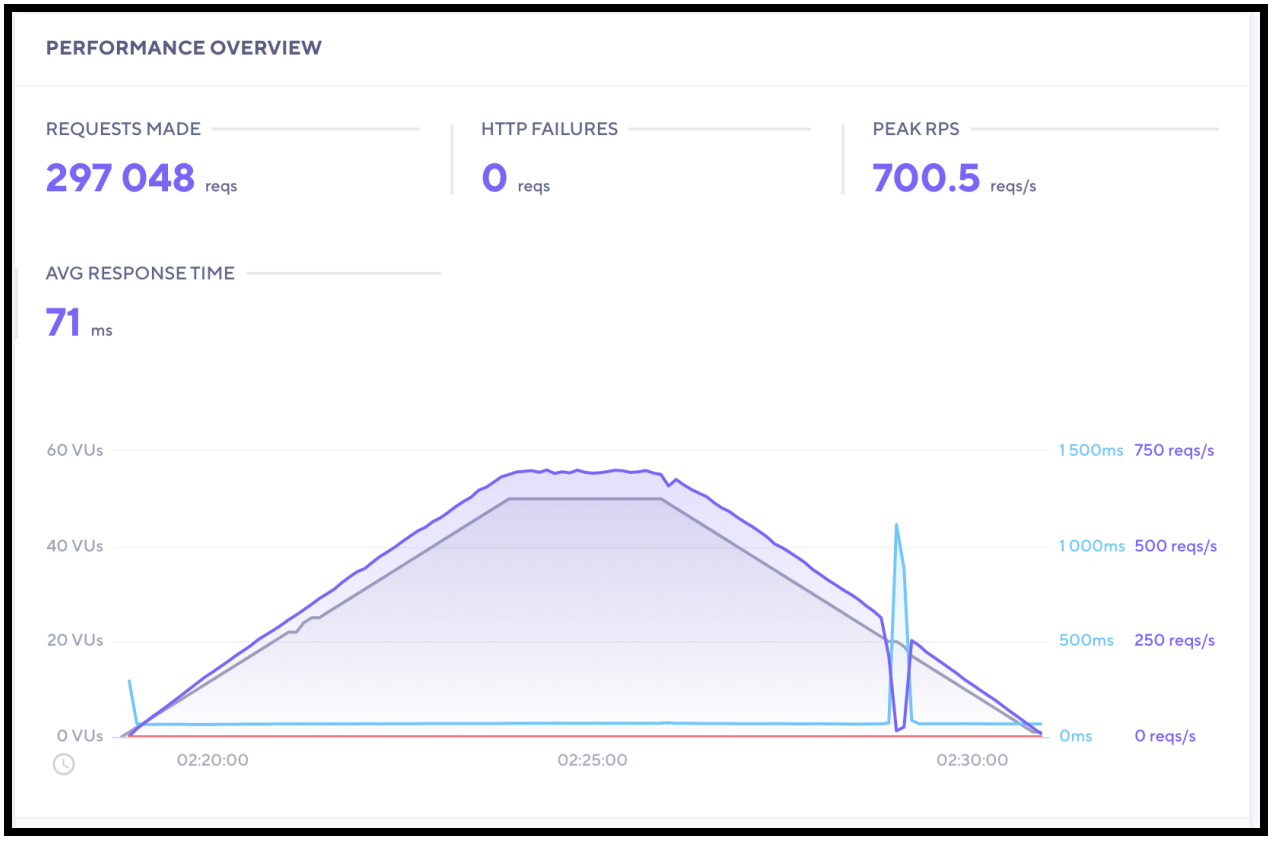
Let’s run 1000 RPS over 30 seconds. 29228 / 30 = 974 RPS throughput \(^o^)/

I incremented the RPS to 2000, 2500, 3000, 3500. The latency was under 75ms for all tests with <1% error rate. Now, it just matters to make sure the API can handle LOADS of traffic. This last test chart below is 4000 RPS with an average latency of 234ms. It seems the throughput is shaky with this load test. We will benchmark 4000 RPS and try to get this performance down.



Let's run k6, with this setup, to see how the API performs across multiple endpoints. This show WAY better performance than the previous broad k6 (cloud) load test.





URL	METH...	STATUS	COUNT	MIN	AVG	STDD...	P95	P99	MAX
.../products/700000/rela...	GET	200	37.1K	60ms	74ms	180ms	81ms	94ms	8 780ms
.../products/999999	GET	200	37.1K	60ms	72ms	98ms	81ms	94ms	5 129ms
.../products/999999/relat...	GET	200	37.1K	60ms	70ms	37ms	81ms	94ms	3 610ms
.../products/454755	GET	200	37.1K	60ms	70ms	34ms	81ms	94ms	5 373ms
54.70.150.169/products/1	GET	200	37.1K	60ms	70ms	22ms	81ms	94ms	2 422ms
54.70.150.169/products/	GET	200	37.1K	60ms	70ms	5.94ms	81ms	94ms	502ms
.../products/999999/styles	GET	200	37.1K	60ms	70ms	5.68ms	81ms	94ms	480ms
.../products/700000/styles	GET	200	37.1K	60ms	70ms	5.52ms	81ms	94ms	458ms

This k6 test above is AMAZING!! We have 297,048 requests ALL under 2000ms. The average latency for ALL requests are ~70ms. With caching, this is showing a very low latency, basically only however long it takes to return data without hitting the database.

ALSO, we have a SOLID 700 RPS for 10 minutes. This is pretty good. While we are hitting 8 different endpoints with varying data responses, the API can perform WAYYYYY better than before.

NEXT: Let's add `keepalive_requests` to the nginx configuration. Let's run another load test with loader.io and 4000RPS with this new setup. Here we see 3,800 requests per second. There were some errors, but this is a rate less than 1%.

```
# Keepalive connections
# keepalive_requests - The number of requests a client can make over a single keepalive connection
# The default is 100, but a much higher value can be especially useful for testing with a load-generation tool, which generally sends a large number of requests from a single client
```

Let's add more features to the load balancer nginx configuration. Here I added these settings to the nginx load balancer:

```
# to boost I/O on HDD we can disable access logs
access_log off;

# send headers in one piece, it is better than sending them one by one
tcp_nopush on;

# don't buffer data sent, good for small data bursts in real time
tcp_nodelay on;

# reduce the data that needs to be sent over network -- for testing environment
gzip on;
# gzip_static on;
gzip_min_length 10240;
gzip_comp_level 1;
gzip_vary on;
gzip_disable msie6;
gzip_proxied expired no-cache no-store private auth;
gzip_types
   # text/html is always compressed by HttpGzipModule
   text/css
   text/javascript
   text/xml
   text/plain
   text/x-component
   application/javascript
   application/x-javascript
   application/json
   application/xml
   application/rss+xml
   application/atom+xml
   font/truetype
   font/opentype
   application/vnd.ms-fontobject
   image/svg+xml;

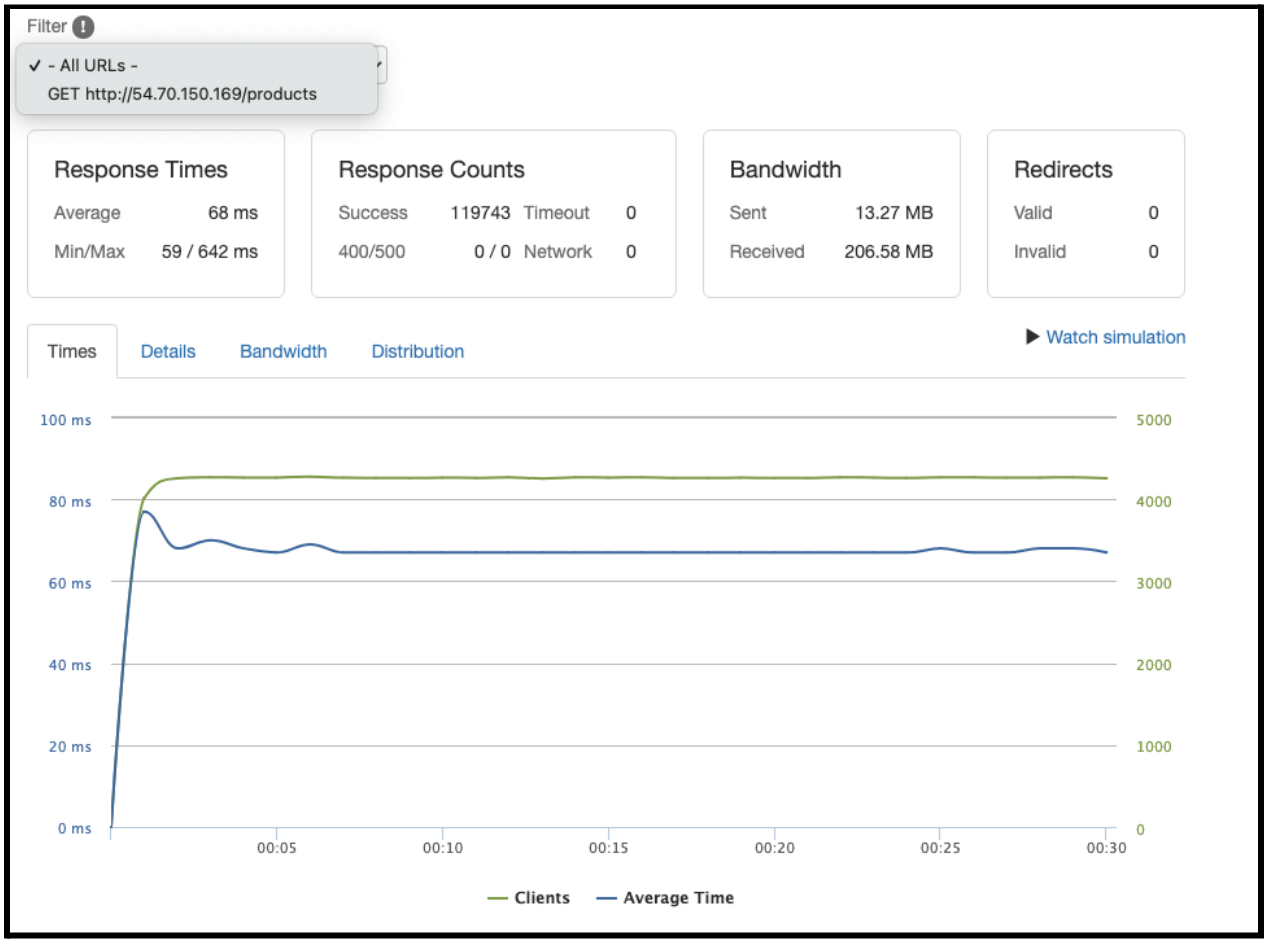
# allow the server to close connection on non responding client, this will free up
memory
reset_timedout_connection on;

# request timed out -- default 60
client_body_timeout 10;
```

```
# if client stop responding, free up memory -- default 60
send_timeout 2;
```

```

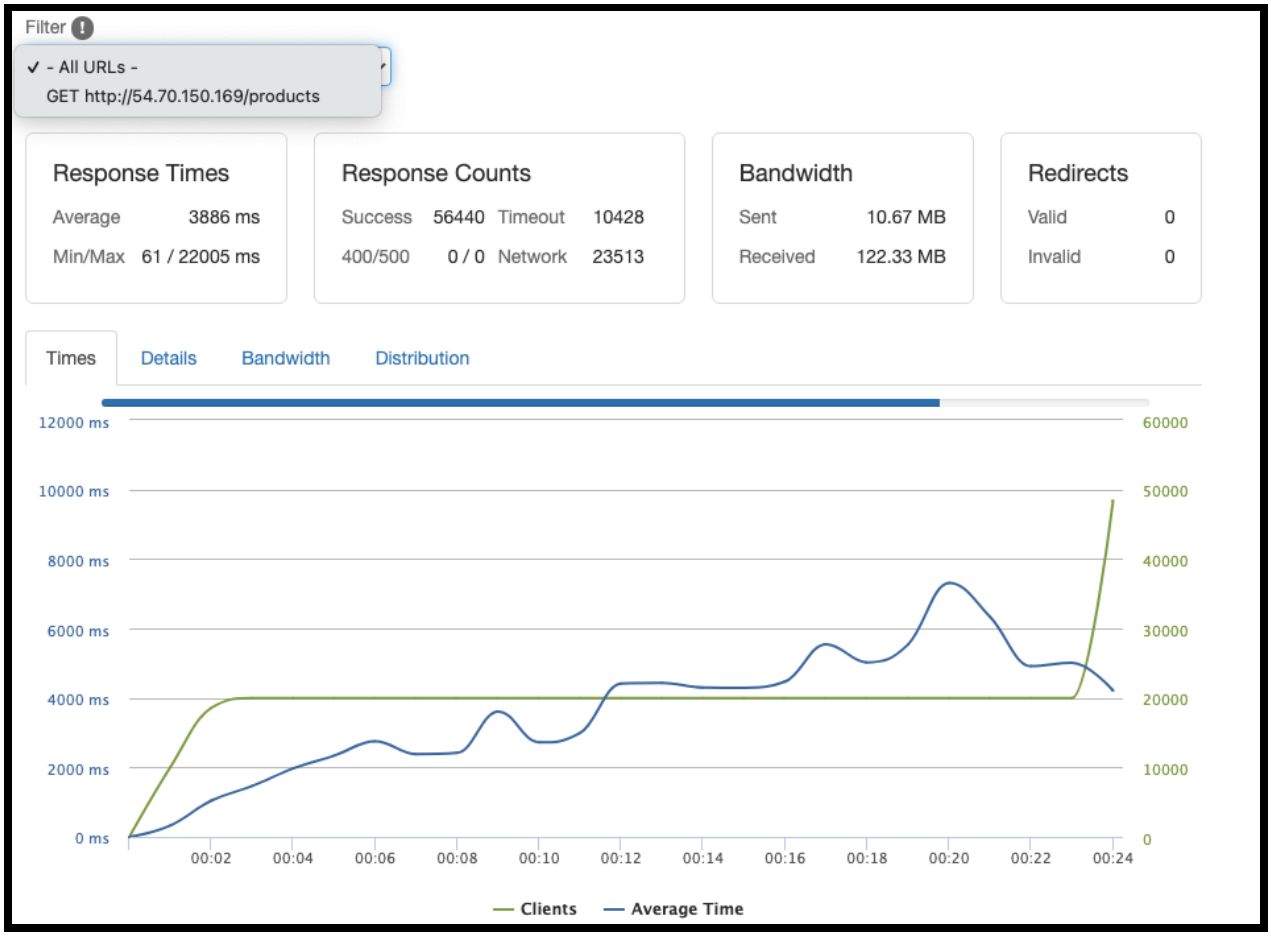
This brought down the latency on 4000RPS down to 68ms. This was excellent. Also, ZERO timeouts occurred in the Response Counts Results.



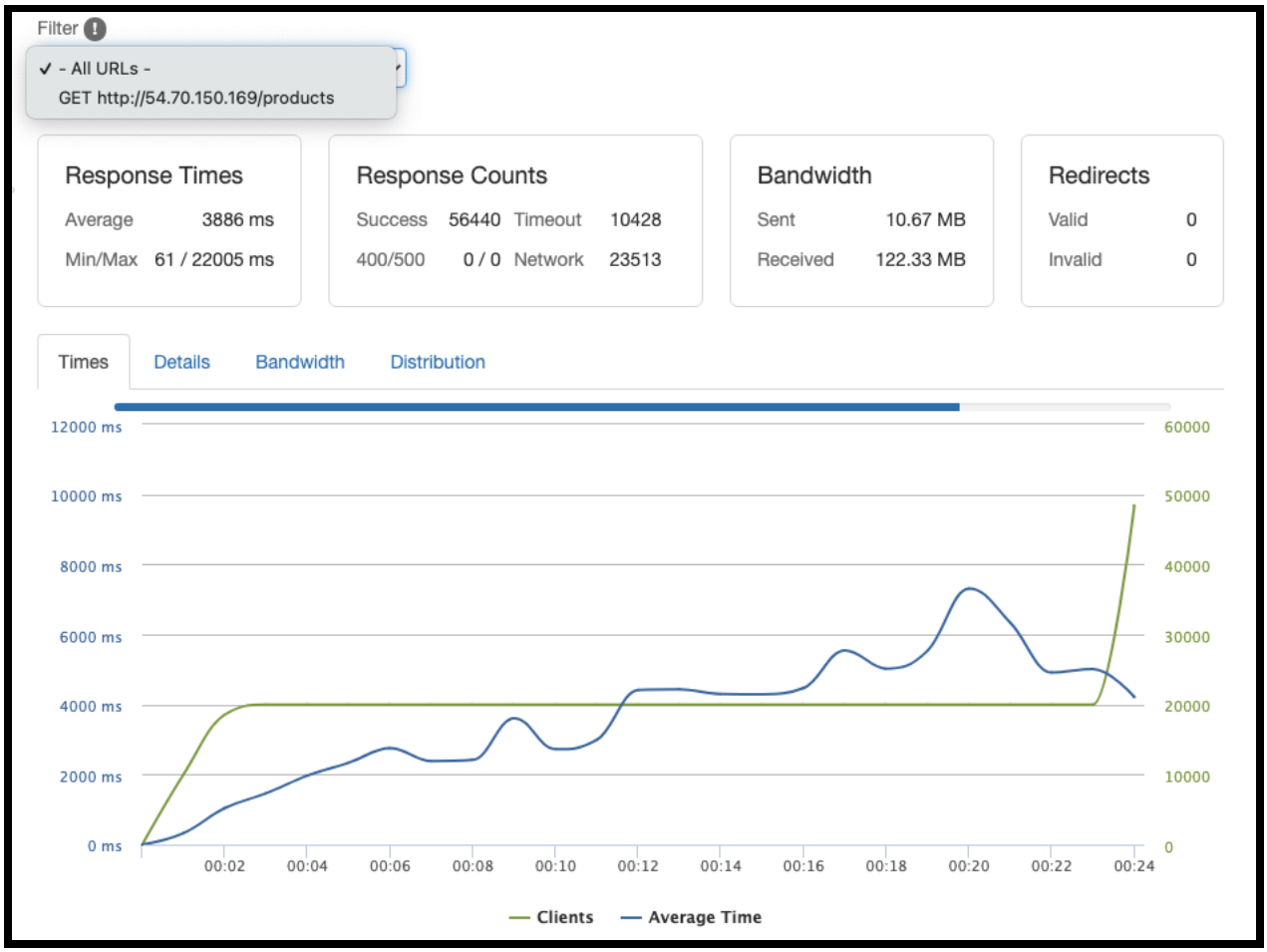

Has the API throughput ceiling been lifted? Next, I tested 4,500 RPS. Everything seemed find. I decided to load test 5000RPS over 30 seconds, here is the result:
```

Next, I tested 10,000 RPS. Yikes, this had a 40%+ error rate. BUT, before the EC2 straight crashed (could not even ssh into it) when I stress testing around a 100 RPS.

This test below is 8000 RPS for 30 seconds. The server starts to buckle up under this load.



Well, it looks like the performance is tapering off around 5000-6000+ RPS.



Notes:
Why we need load balancing:
https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm/

Day 12:

Cleaned up the codebase. Journaled everything. Created powerpoint to showcase lessons learned during the project.

Notes:
// PostgresSQL Connecting:
// https://node-postgres.com/features/connecting
// Connecting to PostgreSQL with Node.js
// https://www.thisdot.co/blog/connecting-to-postgresql-with-node-js
// Postgress Commands:
// https://www.postgresqltutorial.com/psql-commands/

Express + Sequalize API and unit testing:
https://levelup.gitconnected.com/building-an-express-api-with-sequelize-cli-and-unit-testing-882c6875ed59

SQL commands for Sequalize
https://fengmk2.github.io/blog/2014/10/sql-to-sequelize-mapping-chart.html

// https://github.com/sequelize/sequelize-auto
npx sequelize-auto -h 127.0.0.1 -d postgres -u postgres -x example -p 5432 --dialect postgres -c ./config/db.config.js -o ./app/models -t products

Match the PostgreSQL User

Make sure Postgress has a user matching .env credentials
Create a USER matching configuration credentials

```sh
psql=# CREATE USER postgres WITH PASSWORD 'example';
OR
psql=# ALTER USER postgres WITH PASSWORD 'example';
```

Resources:

Dealing with Data
- [Copy command in Postgres](https://www.postgresql.org/docs/12/sql-copy.html)
- [Mongo Import](https://docs.mongodb.com/manual/reference/program/mongoimport/#csv-import)
- [Bulk loading in Postgres](https://www.mydatahack.com/bulk-loading-postgres-with-node-js/)
- [Node Streams to Clean csv](https://dev.to/zluther89/using-node-streams-to-make-a-csv-cleaner-148m)
- [Backpressuring](https://nodejs.org/es/docs/guides/backpressuring-in-streams/)

Benchmarking (Optimization) and Scaling
- [6 Rules of Thumb for MongoDB Schema Design: Part 1](https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1)
- [4 Architecture Issues When Scaling Web Applications: Bottlenecks, Database, CPU, IO](http://highscalability.com/blog/2014/5/12/4-architecture-issues-when-scaling-web-applications-bottlene.html)
- [Efficient Use of PostgreSQL Indexes](https://devcenter.heroku.com/articles/postgresql-indexes)

- [How To Install and Use PostgreSQL on Ubuntu 18.04](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-postgresql-on-ubuntu-18-04)
- [Simple Tips for PostgreSQL Query Optimization](https://statsbot.co/blog/postgresql-query-optimization/)
- [High Performance Postgres](https://stackify.com/postgresql-performance-tutorial/)
- [Caching with Node](https://scotch.io/tutorials/how-to-optimize-node-requests-with-simple-caching-strategies)
- [Manifesto for Agile Software Development](https://agilemanifesto.org/)

System Design Primer
- [GitHub - donnemartin/system-design-primer: Learn how to design large-scale systems]()

Node/Express
- [Performance Best Practices Using Express in Production](https://expressjs.com/en/advanced/best-practice-performance.html)
- [Good practices for high-performance and scalable Node.js applications Part 1/3](https://medium.com/iquii/good-practices-for-high-performance-and-scalable-node-js-applications-part-1-3-bb06b6204197)
- [Good practices for high-performance and scalable Node.js applications Part 2/3](https://medium.com/iquii/good-practices-for-high-performance-and-scalable-node-js-applications-part-2-3-2a68f875ce79)
- [Good practices for high-performance and scalable Node.js applications Part 3/3](https://medium.com/iquii/good-practices-for-high-performance-and-scalable-node-js-applications-part-3-3-c1a3381e1382)

AWS Scaling
- [Scale Your Web Application — One Step at a Time | AWS Architecture Blog](https://aws.amazon.com/blogs/architecture/scale-your-web-application-one-step-at-a-time/)
- [How to find an optimal EC2 configuration in 5 steps (with actual performance tests and results) - Concurrency Labs](https://github.com/donnemartin/system-design-primer#cache)

Caching
- [GitHub System Design Primer - Caching](https://github.com/donnemartin/system-design-primer#cache)
- [Caching guidance - Best practices for cloud applications | Microsoft Docs](https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching)

NGINX
- [A Guide to Caching with NGINX and NGINX Plus - NGINX](https://www.nginx.com/blog/nginx-caching-guide/)