2019

Assignment 2 Report

CAB432 CLOUD COMPUTING – SEMESTER 2 2019

JOHN SANTIAS N9983244

KA LONG LEE N9845097

Contents

Introduction

Tweetery aims to provide users with a visual representation of the analysis performed on searched tweets. Utilising the Twitter API for getting tweets into the webserver, cleaning the tweet body and performing emotional analysis with IBM Watson API. The results of the performed analysis are displayed on a graph for the user.

When the user searches for a topic, the query is sent to the node server for further processing. The server uses the search query to get tweets from Twitter API. The received tweets are then combined and sent to IBM's Watson Natural Language Understanding API to analyse the emotions. These results are stored on a remote global Redis Cache server which can be returned to users who searches for the same topic. Results are also stored on MongoDB for long term storage and can be copied to the cache upon user request. With the assistance of the storage services, Redis cache, and MongoDB, we allow the application to be stateless and provide persistence to the architecture.

Tweetery is hosted on the Amazon Web Services (AWS) cloud provider. Utilising EC2 instances, an auto-scaling group, a launch configuration, and a load balancer. These services help tweetery perform consistently by regularly checking the health of our servers and scaling based on our scaling policy.

There are two features in tweetery, week analytics feature for analysing tweets in the last seven days, stream feature for analysing real-time information. However, for the purpose of this assignment, we focus on the week analytics feature.

Development

Tweetery was developed in stages to have a tight grip on the scope of the application and reduce the number of potential issues towards the end of development.

Stage 1:

Stage 1 involved setting up the skeleton of the project and generating the keys for the Twitter and IBM API. The skeleton for the node application consisted of two folders, one dedicated for the client, the other for the server. Dependencies were also installed for the node application to process API calls. As expected by the end of this stage, the application was able to call the Twitter API for trending topics and tweets based on single/multiple queries for emotional analysis using IBM's API.

Stage 2:

Stage 2 involved creating the front-end UI to display components such as the trending topics list, a navigation bar, a search bar and a graph for the processed emotional analysis. It was ensured that client-side was calling the correct server endpoints for heavy processing.

Stage 3:

Stage 3 involved setting up two storage services for the node application to store and retrieve data. This included setting up a Redis cache server and a MongoDB database. The node application was connected to both services, then stored and retrieved data based on a defined schema. Afterward, testing and bug fixing was conducted followed by dockerising the application.

Stage 4:

Stage 4 involved deploying to the cloud with further testing and bug fixing. Monitoring the application was an additional step to this stage to find the most optimal way of ensuring auto-scaling. This involved setting up CloudWatch alarms and monitoring the application as it was being used.

During this monitoring process, it was discovered that the application was not producing enough CPU load for scaling, so we tried to move the Redis server from being a global cache to a docker container image linked to our node server. We also tried to increase the processing load by adding an NLP library and sorting functions. This issue is further explained in the *Scaling and Performance* section.

Stage 5 (Extra feature):

Stage five involved implementing the streaming feature that would analyse the emotion of real-time tweets and plot new results on the graph every ten seconds. We discovered that scaling the twitter stream was a problem if multiple hit the same endpoint with different search queries. The twitter streaming endpoint only allows one connection at a time. We decided to keep this as an extra feature however only allowing one user to initiate the stream at a time.

API and Packages

Twitter API

https://developer.twitter.com/en/docs.html

Twitter is a social network platform for users to share messages, images, and videos on a feed. Messages can be categorised by hashtags (#) and searches can be conducted on those hashtags or keywords. Tweetery uses this API to retrieve tweets based on a hashtag or keyword.

IBM Watson Natural Language Understanding API

https://www.ibm.com/watson/service/natural-language-understanding

The IBM Watson Natural Language Understanding API is one of IBM Watson's machine learning API. It is used to process advanced test analysis and extract metadata from content such as concepts, entities, keywords, categories, sentiment, emotions, relations, and semantic roles. Some of which can provide a score for the tone or a list of words that have great meaning for the user.

Tweetery uses this API to analyse the emotion of the tweets gathered and return a score between zero and one. Given the highest score is the emotion analysed in the content.

MongoDB

https://www.mongodb.com & https://mlab.com

MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database program and uses JSON-like documents with schema. mLab was used to host MongoDB databases on a cloud provider (AWS, GCP, Azure). Tweetery uses two tables to store emotion analysis.

Compromise

http://compromise.cool/

Compromise is a JavaScript library for processing natural language. Tweetery uses utilises compromise to clean tweets, removing invalid or unusual characters. This increases the accuracy and the likelihood of running a proper analysis from IBM's API.

ChartJS

https://www.chartjs.org

ChartJS is a community-maintained project for developers to visualise data in eight different ways. Each of them animated and customisable. Tweetery uses this library to visualise the emotional scores of the given topic.

Socket.io

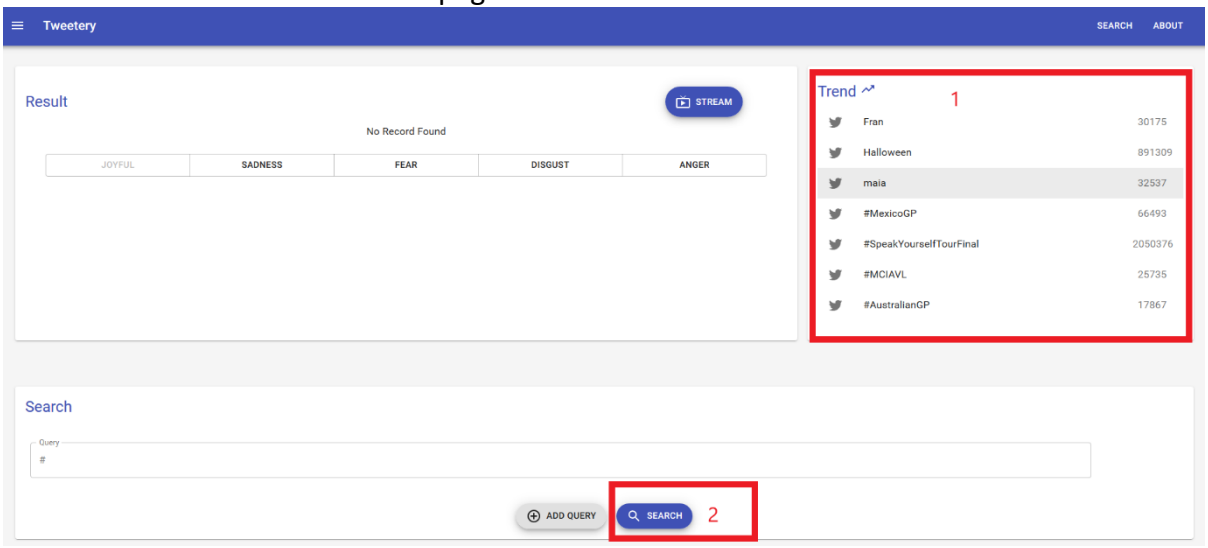
https://socket.io/

Socket.io is a JavaScript library for building real time web applications. It enables real time, bi-directional communication between the client and server. Tweetery utilises socket.io for the streaming feature and sending newly analysed data.

Use Cases

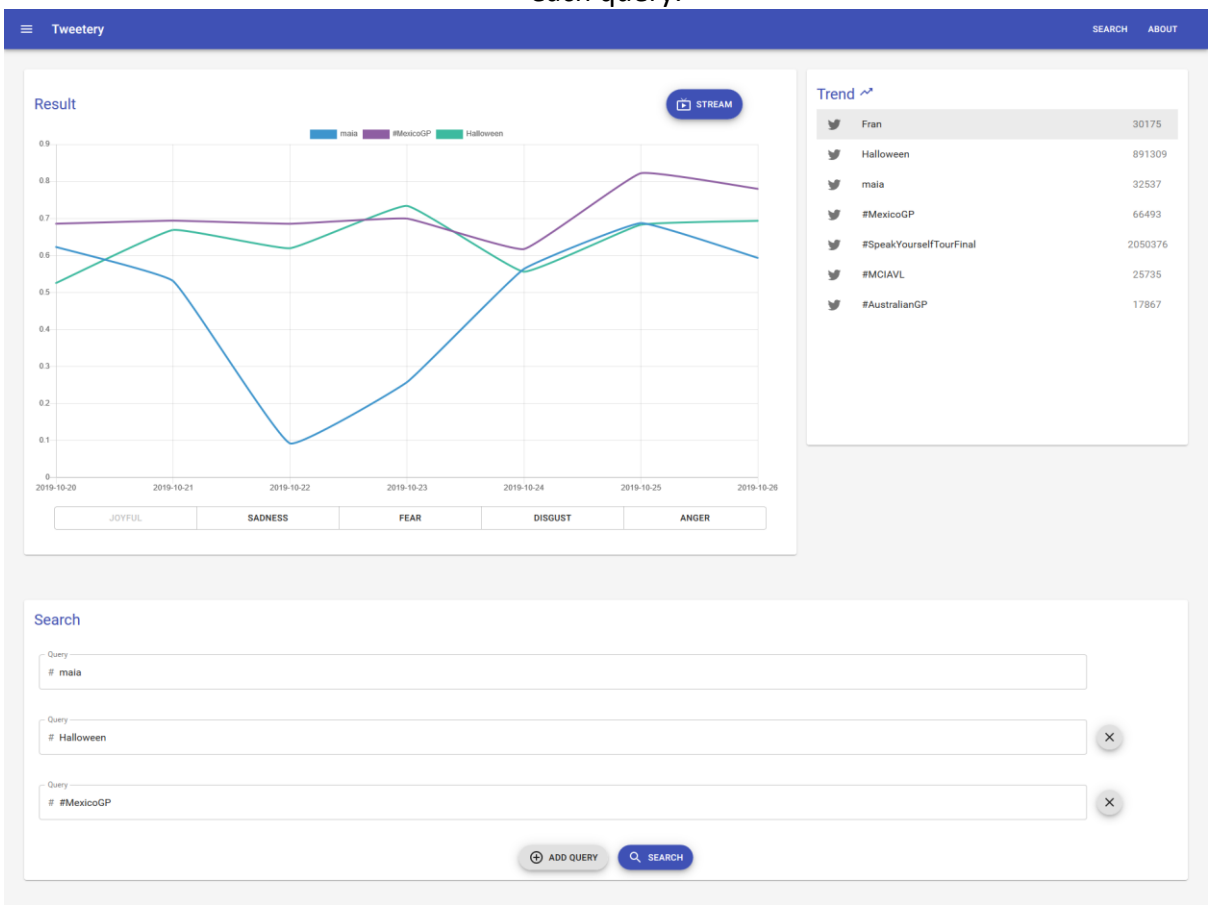
As a user, I want to see a list of trending topics and select one to see the average emotional stats.

The user lands on the index page of Tweetery. The user selects a trending topic from the list on the right-hand side and initiates the search with the button. Results are displayed on the page after a few seconds.



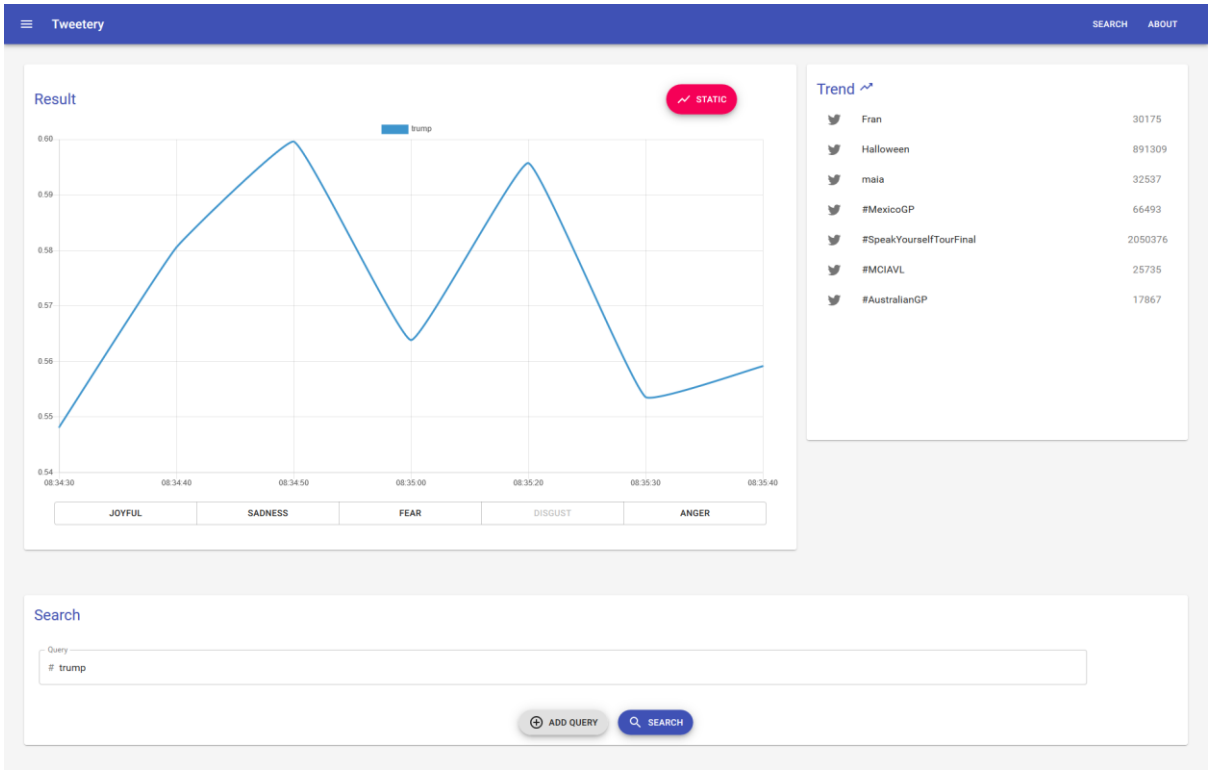
As a user, I want to select or search multiple trending topics to compare the emotions of those topics.

The user selects multiple topics and initiates the search. The results will show one line for each query.



As a user, I want to see real-time analytics of my chosen topic so I can get informed on how people feel about the topic

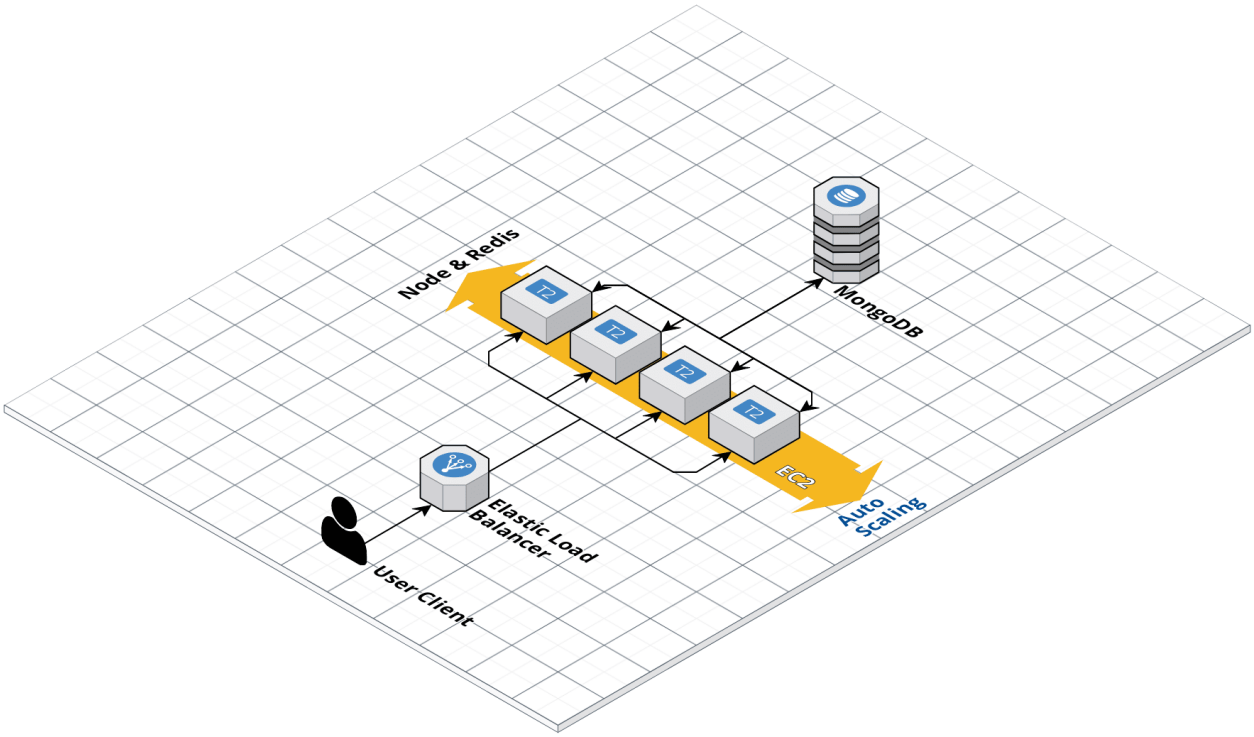
The user selects the stream button on the top right of the result box. Enters a query and initiates the search. Results will show only one line for the query and update every 10 seconds.



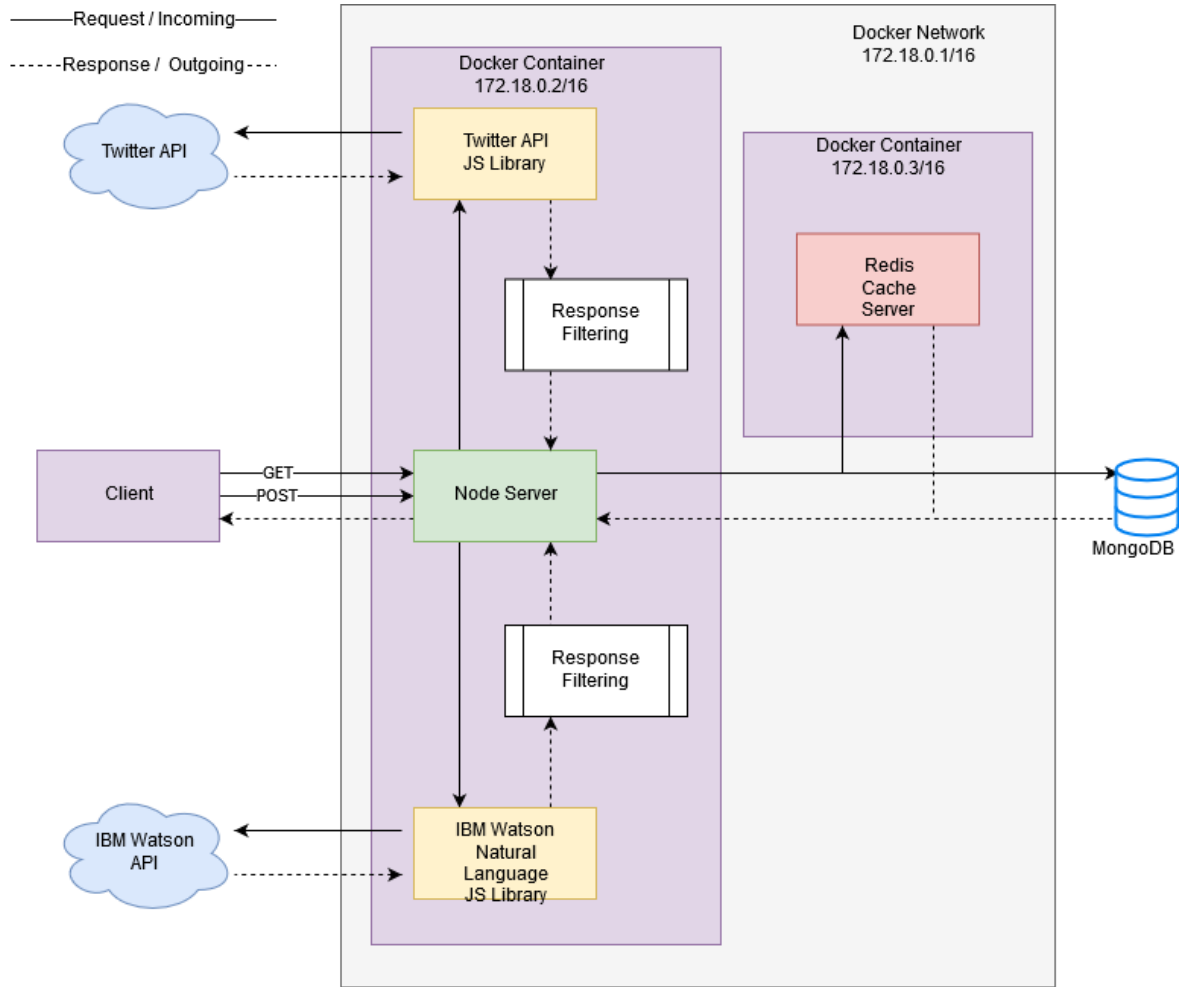
Technical Description

Tweetery utilises one EC2 t2.micro server to achieve its tasks in a stateless manner. Inside this server runs two docker containers inside of a bridged network. One running NodeJS application to process user requests, and the other running a Redis server to cache data. In addition to this, MongoDB is utilised to provide persistence.

Cloud Architecture



Server Architecture



Client Side

The client can initiate two types of requests, a GET, and a POST request. When a user requests to access the application, a GET request is sent to the server. The server returns the index page with a navbar, a list of trending topics and a search bar. When the user initiates a search, the form body sends a POST request to the server for processing. The server handles the query sent from the form, processes the API calls with the query and displays the results in a graph. The client uses the ReactJS framework to render the information and send to the user.

Server Side

The node server manages the incoming client request and outgoing responses. When it receives a GET request, the server renders and returns the home page with a navbar, search bar, and trending topics. There is no heavy processing in this request because the client hasn't searched for anything. The client is only requesting the home page.

The heavy processing is initiated when the server receives a POST request containing the search queries. The server processes one query at a time and searches the tweets for each of the last seven days. This endpoint returns a JSON object with tweet data stored in an array. Each element containing the tweet message, user information, time, date. At this

point, the server extracts the tweet messages, combines them and sends it to IBM's API for emotional analysis. The result of this analysis is pushed to a JSON object with the date and query, then stored in the Redis cache for short-term storage and MongoDB for long-term storage. This information is also rendered and sent to the client for display. This process is repeated for each search query.

The server runs in a stateless manner, relying on storage services and APIs for storing and searching information. In every search initiated, the server always checks if the data exists on Redis and MongoDB before running a new search on the APIs. The process is similar for the streaming feature.

AWS Elastic Load Balance

An AWS Elastic Load Balancer automatically distributes incoming application traffic across multiple targets. Tweetery utilises a load balancer to balance the traffic between all the instances in the Auto-Scaling group.

AWS Auto-Scaling Group

An AWS Auto-Scaling group creates EC2 instances dynamically based on the defined conditions. It maintains the number of instances by performing periodic health checks on the instances in the group. Tweetery utilises this to ensure consistency by scaling when the server's resources are overloaded.

AWS EC2 Server

All of the servers are launched as a t2.micro server in the Asia Pacific (Sydney) region. Each of the servers runs a custom bash script on boot to check for the latest version of the docker image. If there's a new version the server will pull and run it. This was used throughout development, allowing me to make changes locally on my machine, create and push the docker image and reboot the server.

Docker

The Node and Redis servers are separated into docker containers and linked to each other for storage. These containers are run inside of an independent network (172.18.0. 0/16) connected to the default network bridge for ease of access and minimising multiple port forwarding rules. By default, public users can connect to the node application through port 5000 and the node application can access the cache through port 6379.

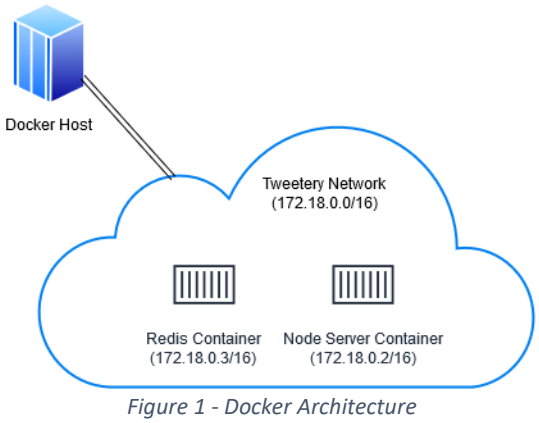


Figure 1 - Docker Architecture

MongoDB

MongoDB was remotely hosted through mLab. Tweetery utilises a NoSQL database to write and retrieve records. Collections 'Trends' to store trending data, and 'Emotions' to store emotional data. MongoDB was chosen due to the ease of use and documented-oriented storage suitable for our JSON data.

Scaling and Performance

The auto-scaling group has been configured to scale based on the size of network traffic coming in from the load balancer. This was chosen over CPU utilisation because server processing was very low and wasn't ideal to scale when less than 10% of the CPU was used. This feature is intended for the week analytics feature.

Thorough investigation went into the metrics used to scale the application. CPU utilisation never hit peaks that warranted extra server capacity as it managed to handle client requests quickly. At the launch of a new instance, the server would utilise around 40% of the CPU. During this time the server would start the docker containers and establish new connections to the storage services. However, usage would then drop to 3%.

We found that initiating a search with new queries would slightly increase the processing to 6%. This was due to the information not available on our storage services and the server would have to process API calls, structure the data, store it and return to the client. New information takes more time to process. In the case where the information was available on either of the storage services, the server would usage less processing power and respond quicker.

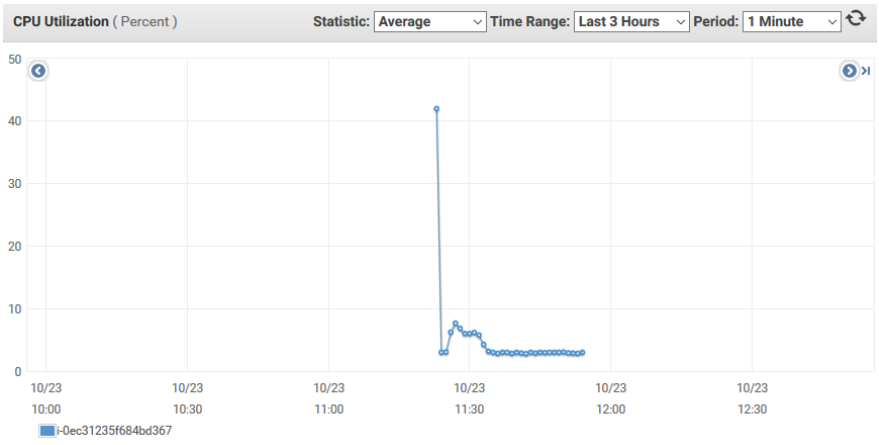


Figure 2 - CPU Utilisation

We attempted to increase the CPU usage by adding sorting and cleaning functions, and an NPM package such as NLP Compromise. As a result, there was no increase and finalised the decision to scale based on the incoming network traffic. The use of AWS CloudWatch alarms helped us monitor services and adjust policies based on what was observed.

Figure 3 - CloudWatch Alarms

Scaling policy based on incoming network traffic was the best way to demonstrate scaling up and down, as it could be controlled by the client. The trending topics and analysed data can be requested multiple times, resulting in an increase of traffic which initiates scaling.

Scaling Up

If the network traffic was greater than 1.5 million bytes for a period of 60-seconds, then it would add one instance to the auto scaling group.

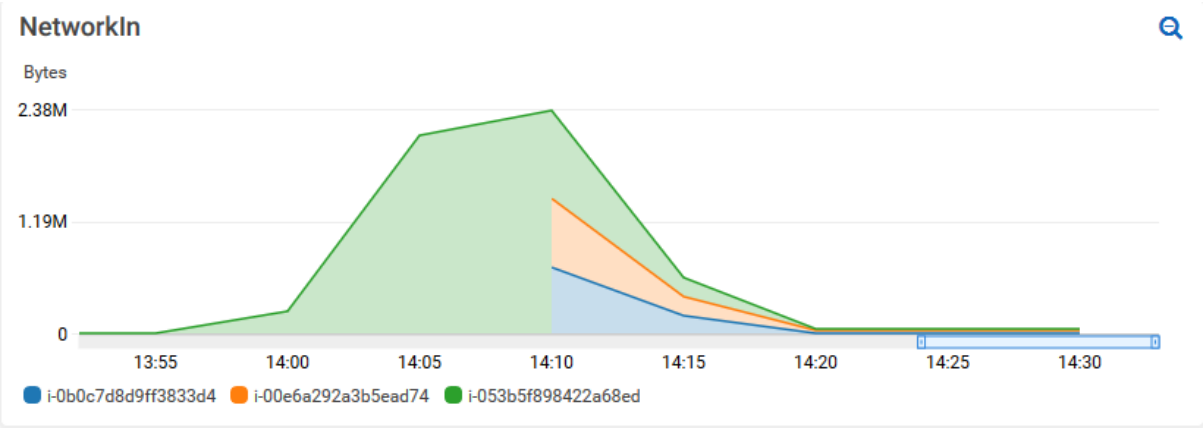


Figure 4 - Network In Alarm for all instances in auto-scaling group

Scaling Down

If the network traffic was less than 1.5 million bytes for a period of 60 seconds, then it would remove one instance of the auto-scaling group.

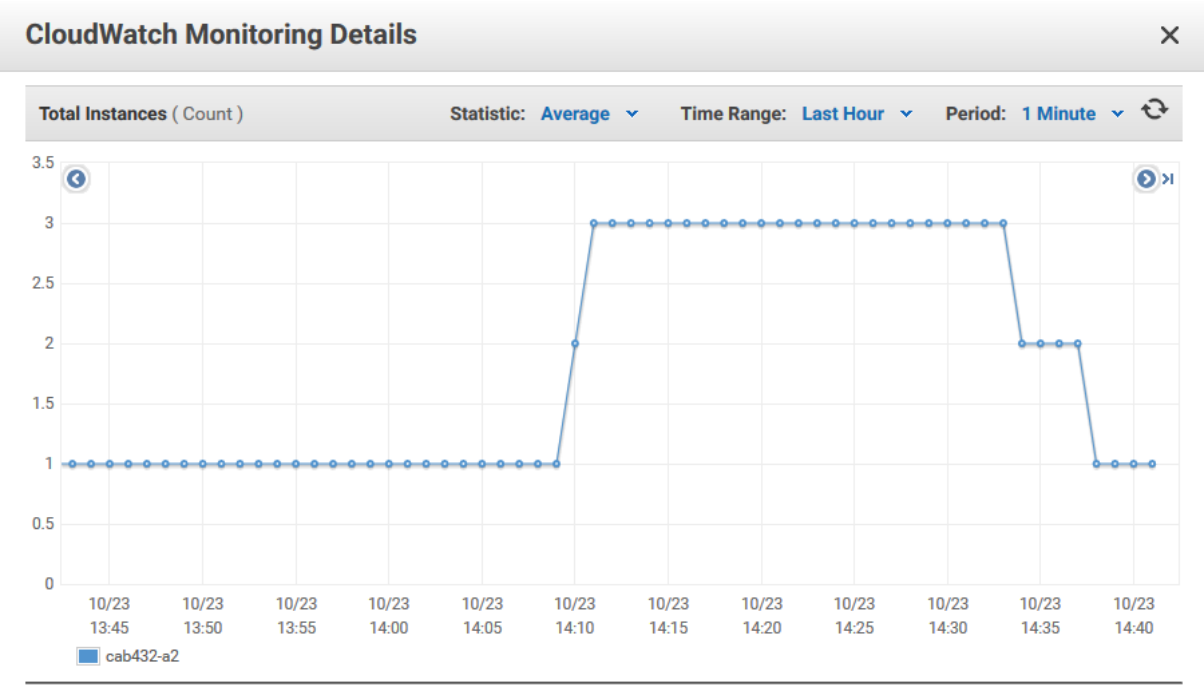


Figure 5 - Auto-scale group down

Testing and Limitations

When Tweetery was first deployed, it was expected that the application would scale based on CPU utilisation. When this was not the case, more time was invested in adding more processing (functions and libraries) and analysing the cloud infrastructure to determine the best method for scaling. As a result, error handling, restructuring code, code commenting, and bug fixing was conducted to create a robust application.

Test Plan

TASK	Expected Outcome	Result	Screenshot/s (Appendix)
Retrieve Home	Display the search box	**PASS**	**C – 1A**
	Display the trending topics	**PASS**	**C – 1B**
Search tweets	Search button sends query	**PASS**	**C – 2A**
	Graph displays	**PASS**	**C – 3B**
	Graph displays Legend	**PASS**	**C – 3C**
	Graph displays Labels	**PASS**	**C – 4A**
	Graph displays Date	**PASS**	**C – 3D**
	Selected topic adds to search box	**PASS**	**C – 5A**
	Add row for search query	**PASS**	**C – 8A**
	Delete search row	**PASS**	**C – 8A**
	Initiate search with more than one topic	**PASS**	**C – 6A, 7A**
	Multiple queries displays on the graph	**PASS**	**C – 7B**
Handle Twitter API response error	Application continues running (week and stream feature), display error	**PASS**	**C – 15A**
Handle IBM Natural Language Understanding API response Error	Application continues running, display error	**PASS**	**C – 16A**
Stream live tweets	Live tweets analysed and displayed on the graph	**PASS**	**C – 17A**
	Stream doesn't write after close	**PASS**	**C – 18A**
	Graph updates every 10 seconds	**PASS**	**C – 19A**
Data to and from Redis	Node server stores data in Redis by specified key	**PASS**	**C – 9A**
	Node server retrieves data from Redis by key specified	**PASS**	**C – 10A**
Data to and from MongoDB	Node server stores data in mongoDB	**PASS**	**C – 11A**
	Node server retrieves data from mongoDB by key specified	**PASS**	**C – 12A**
Load Balancer	Load balancer health check	**PASS**	**C – 13A**
Auto Scaling	Auto scale up	**PASS**	**C – 14A**
	Auto scale down	**PASS**	**C – 14A**

Possible Extensions

Scaling Twitter Stream

Although the twitter stream was implemented, there is still room for improving this feature. One solution is searching all the tweets based on all the trending topics and store it in the database. From there the server could query the database, buffer returned data and display to the user.
Applying this solution would require us to restructure our architecture and the node. Meaning more time developing, investigating and observing AWS configurations (load balancer, scaling policy).

D3JS Graphs

The application currently uses ChartJS to display the data in a line graph. D3JS could be substituted in to display data in a more attractive way that engages users. The variety of charting options is far greater.

Appendix

Appendix A – Cloud Architecture design Iterations

Our initial design separated Node and Redis in two EC2 instances. Our Redis server was going to be used as a global cache and scales when the CPU reaches the threshold. This approach would be more accurate in terms of the data requested and reduce the number of queries on the database. The downside to this was that it would take longer for the Noe server to query both storage services.

We investigated this architecture and finalised the decision to run Node and Redis in the same instance. This was because AWS allows users to run up to five of the same instance types and would cost more money to run more. In addition, the data would be stored in the cache for a short period and are optimised to use less space. This problem to this approach would make the data on each server inconsistent, however, it can be eliminated with the use of MongoDB.

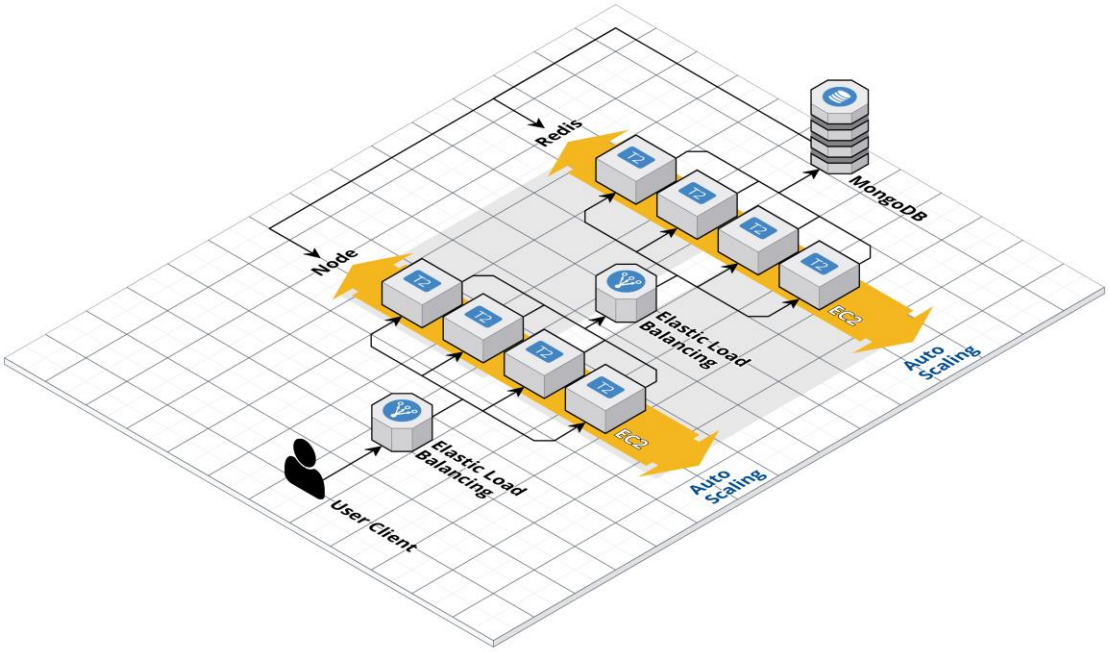


Figure 6 - First Iteration

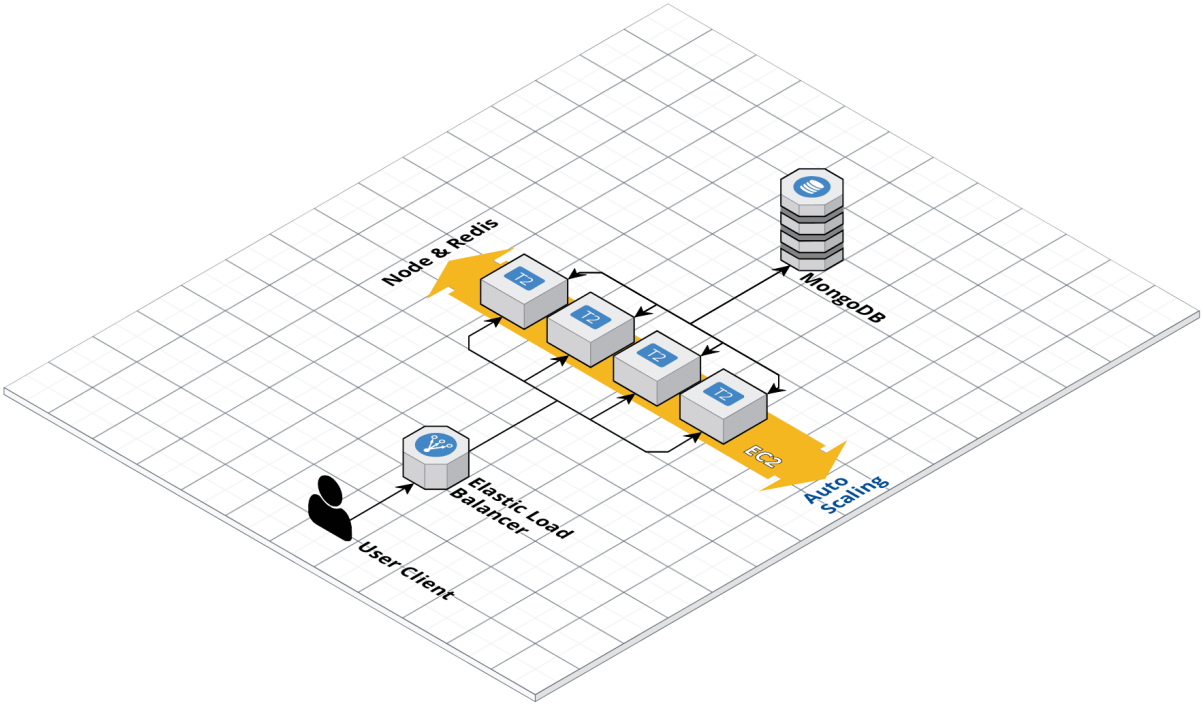


Figure 7 - Second Iteration

Appendix B – Deployment instructions

Pre-Requisites

Security Groups

It is required that three security groups are created. One for running the application and cache, another for the load balancer, and one for the auto-scaling group. This is configured in the Security Groups section of the AWS EC2 management console.

1. **CAB432-Server**
    a. Inbound Port 22 SSH – TCP Protocol – Any Source
    b. Inbound Port 5000 – TCP Rule – Any Source
    c. Outbound – Open All Traffic
2. **Load-Balancer**
    a. Inbound Port 80 HTTP – TCP Rule – Any Source
    b. Outbound – Open All Traffic
3. **Auto-Scaling-Group**
    a. Inbound Port 22 SSH – TCP Protocol – Any Source
    b. Inbound Port 5000 – TCP Rule – Any Source
    c. Outbound – Open All Traffic

AMI

An Amazon Machine Image (AMI) is needed for the auto scaling group. Create an EC2 t2.micro instance with Ubuntu 18.04 and apply the security group **CAB432-Server**.

1. Install docker and configure it to start on boot:

```
sudo systemctl enable docker
```

2. Install docker compose. This tool is used to define and run multi-container docker applications. Once installed, create a new file `docker-compose.yml` in `/home/ubuntu/` with the following:

```yaml
version: '2'
services:
    node-server:
        image: asianjohnboi/tweetery:latest
        ports:
          - "5000:5000"
        networks:
          tweetery-network:
            ipv4_address: "172.18.0.2"
        restart: unless-stopped

    redis-server:
        image: redis
        ports:
          - "6379:6379"
        networks:
          tweetery-network:
            ipv4_address: "172.18.0.3"
        restart: unless-stopped

networks:
  tweetery-network:
    driver: bridge
    ipam:
```

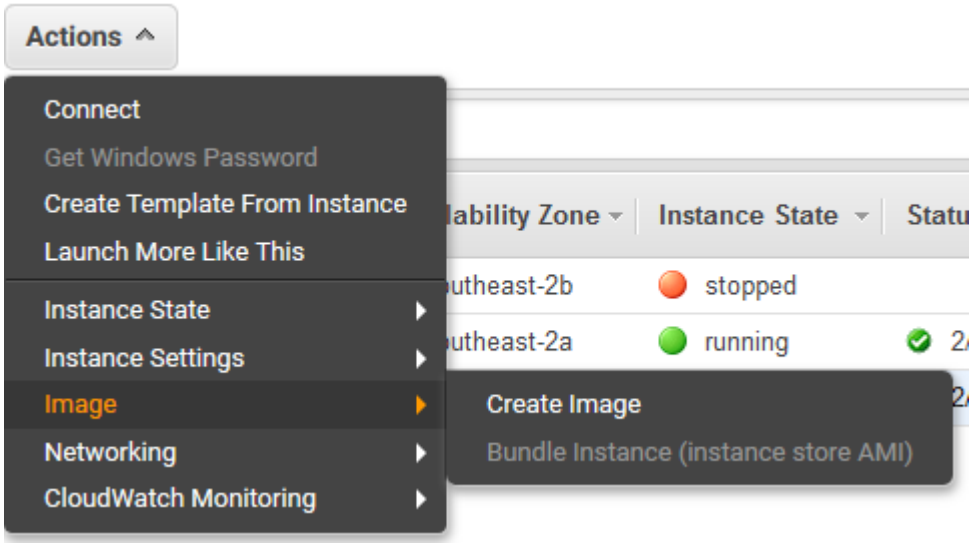
```
    config:
      - subnet: 172.18.0.0/16
```

3. Run `docker-compose up`. This will set up a new docker network, then pull and run the images inside that network.
4. Next create another file called `startup.sh` in the same directory. This bash script file will be used to pull the latest version of the tweetery image (if available).

```
#!/bin/bash

docker rm ubuntu_node-server_1 --force
docker pull asianjohnboi/tweetery:latest
docker run --name ubuntu_node-server_1 --restart=unless-stopped \
           --network=ubuntu_tweetery-network \
           --ip=172.18.0.2 \
           -p 5000:5000 asianjohnboi/tweetery:latest
```

5. Create a rule to execute to execute this file on boot.
    a. Run crontab -e
    b. Add `@reboot sudo /home/ubuntu/startup.sh >> /home/ubuntu/output.log` at the end of the file
    c. Save and reboot.
    d. Open the output.log file to view the outcome of executing the bash script.
    e. Check if the docker container is running.
6. When everything is running as expected, create an image of the running EC2 instance from AWS EC2 management console.



7. The image can then be terminated.

Load Balancer

Create a load balancer from the AWS EC2 Management console.

1. Select Application Load Balancer
2. Configure load balancer
    a. Name the load balancer
    b. Select internet-facing scheme

    c. Address type of ipv4
    d. HTTP listener on port 80
    e. Select two availability zones
3. Do not configure any security settings
4. Add **Load-Balancer** security group
5. Configure routing
    a. Create new target group and name it
    b. Target type is instance
    c. Using HTTP Protocol port 80
    d. Leave health checks by default.
6. Do not register any targets.
7. Review and create Load Balancer

Launch Configuration

Create a launch configuration from the AWS EC2 Management console.

1. Select your recently created AMI.
2. Use the t2.micro instance type
3. Name the configuration and enable CloudWatch detailed monitoring.
4. Skip additional storage.
5. Select **Auto-Scaling-Group** security group.
6. Review and Create launch configuration.

Auto-Scaling Group

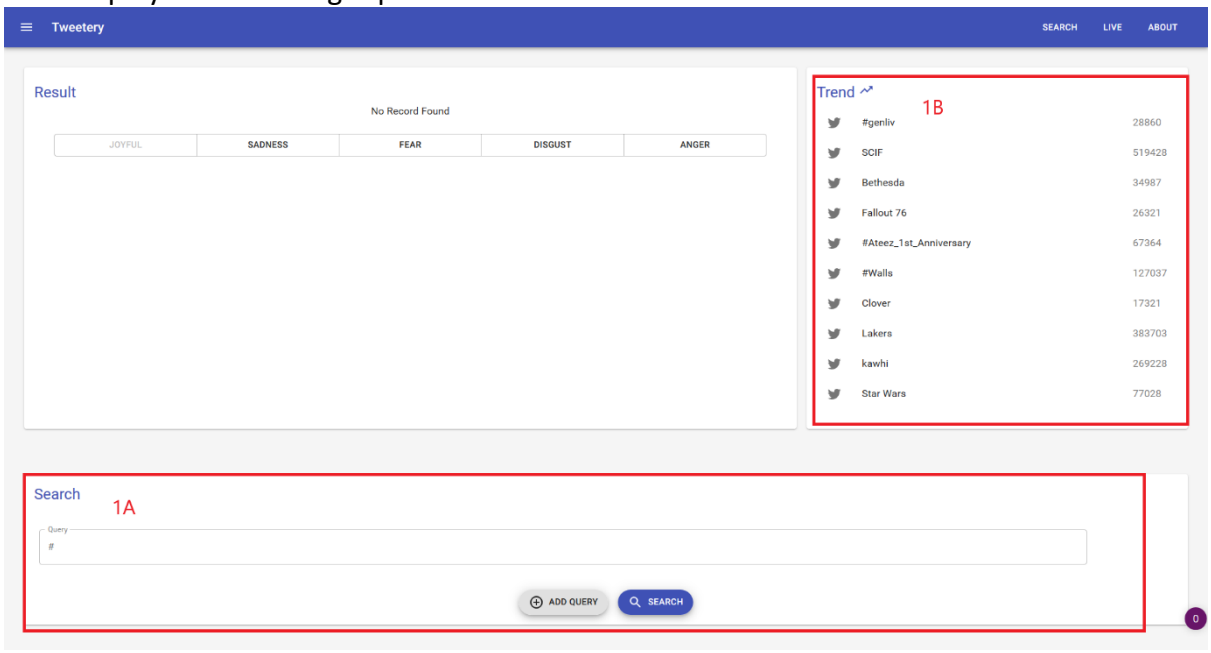
Create an Auto scaling group from the AWS EC2 Management console.

1. Select launch configuration.
2. Name the scaling group.
3. Start the group size with one instance.
4. Select a subnet (Must be in the same availability zone with the load balancer).
5. Click on advanced details
    a. Select 'Receive traffic from one or more load balancers' and add target group
    b. Health Check Grace Period of 60 seconds
    c. Enable CloudWatch detailed monitoring
6. Use scaling policies to adjust the capacity of the group
    a. Scale between 1 and 3 instances.
    b. Change metric type to Average Network In (bytes)
    c. Target value of 1,500,000
    d. Instances need 5 seconds to warm up after scaling
7. Review and Create auto scaling group.

Appendix C – Test Cases

1A – Display the search box

1B – Displays the trending topics

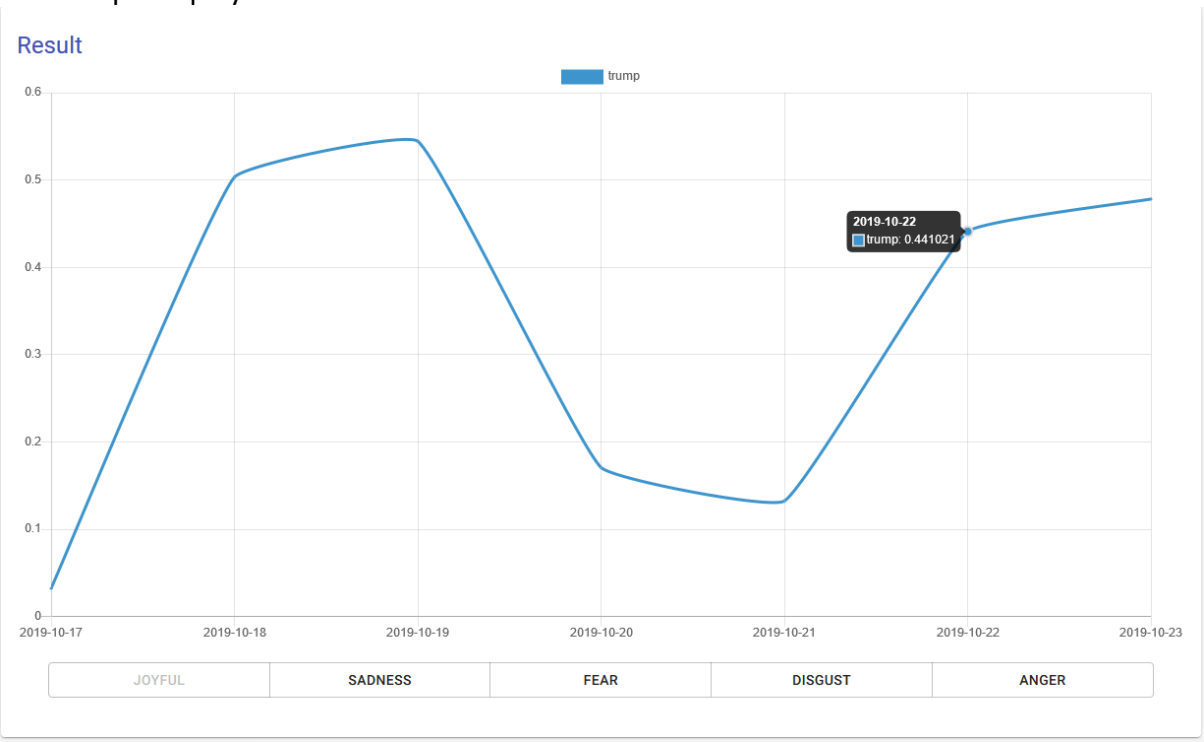


2A – Search button sends query

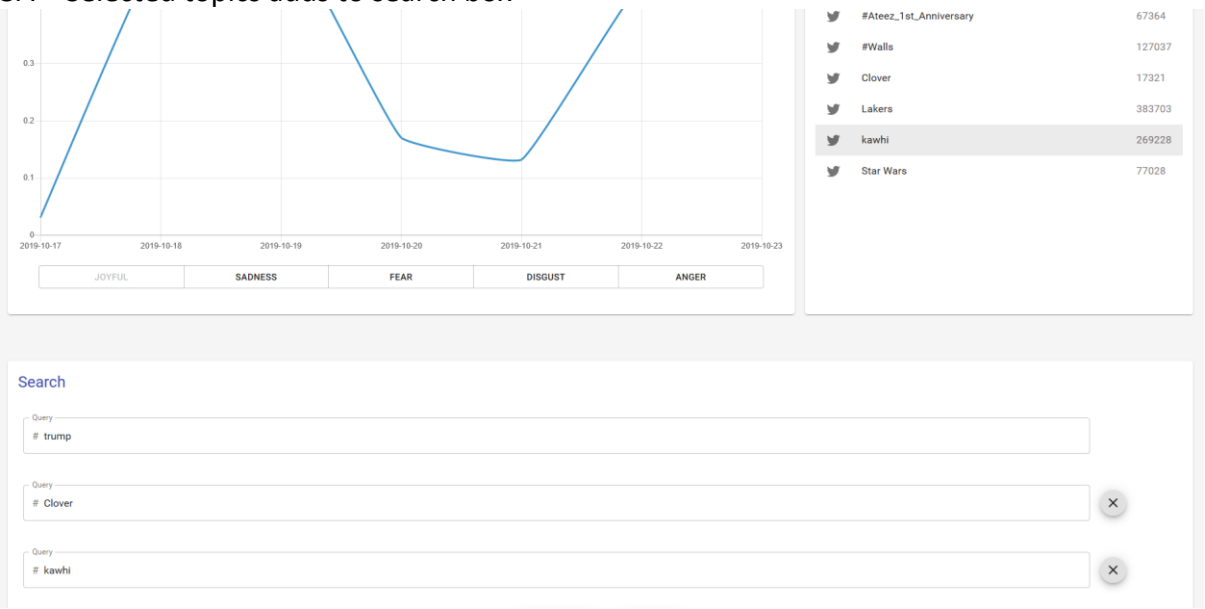
```
POST /api/tweets/analyse 200 2756.915 ms - 2867
GET / 304 1.732 ms - -
GET /static/css/main.2024c935.chunk.css 304 0.710 ms - -
GET /static/js/main.3d0585c0.chunk.js 304 0.550 ms - -
GET /static/js/2.92f1d3c6.chunk.js 304 0.524 ms - -
GET /api/tweets/trends 200 120.757 ms - 7267
GET /logo192.png - - ms - -
GET /favicon.ico - - ms - -
GET / 304 0.360 ms - -
GET /static/css/main.2024c935.chunk.css 304 0.432 ms - -
GET /static/js/2.92f1d3c6.chunk.js 304 0.530 ms - -
GET /static/js/main.3d0585c0.chunk.js 304 0.436 ms - -
GET /api/tweets/trends 200 120.909 ms - 7047
GET /logo192.png - - ms - -
GET /favicon.ico - - ms - -
[ 'trump' ]2A
```

3B – Graph Displays
3C – Graph displays legend
3D – Graph displays Date

4A – Graph displays labels

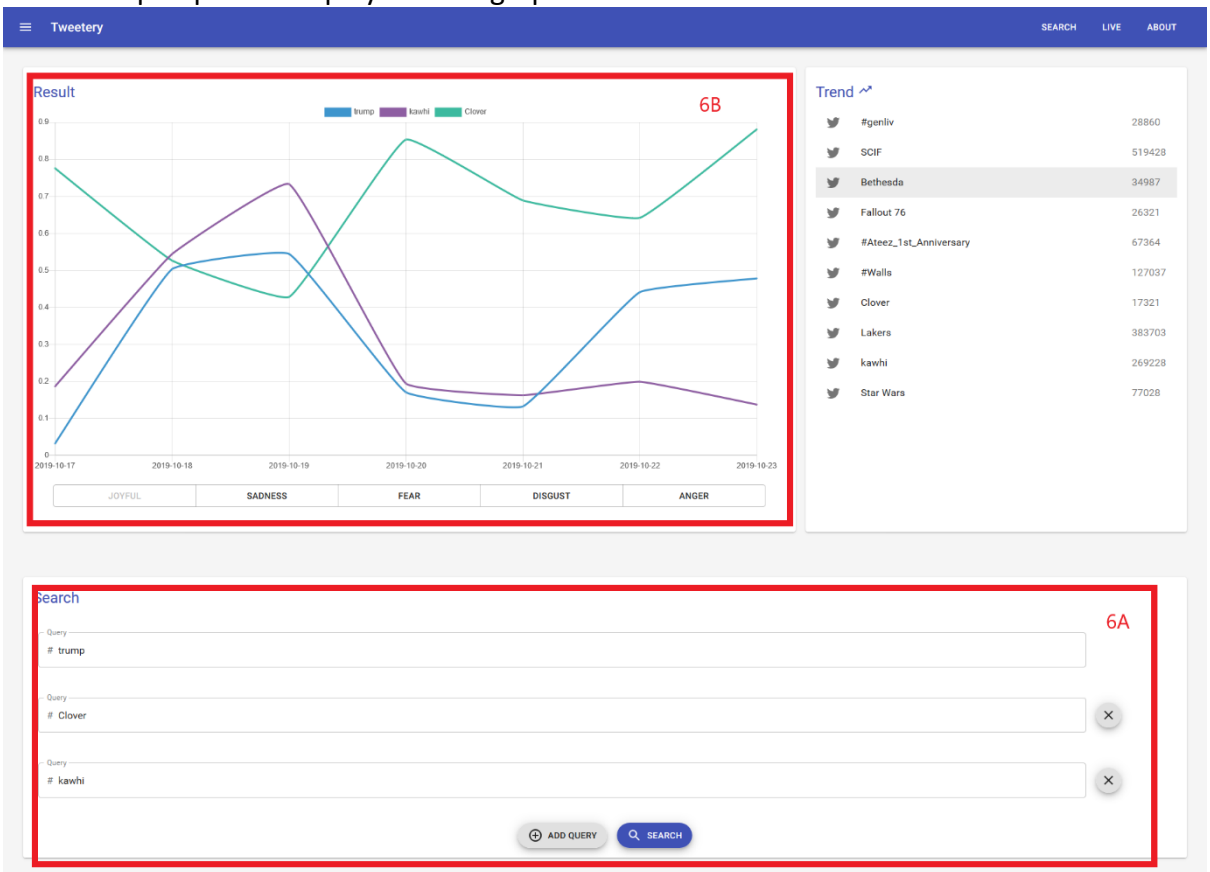


5A – Selected topics adds to search box



6A – Initiate search with more than one topic

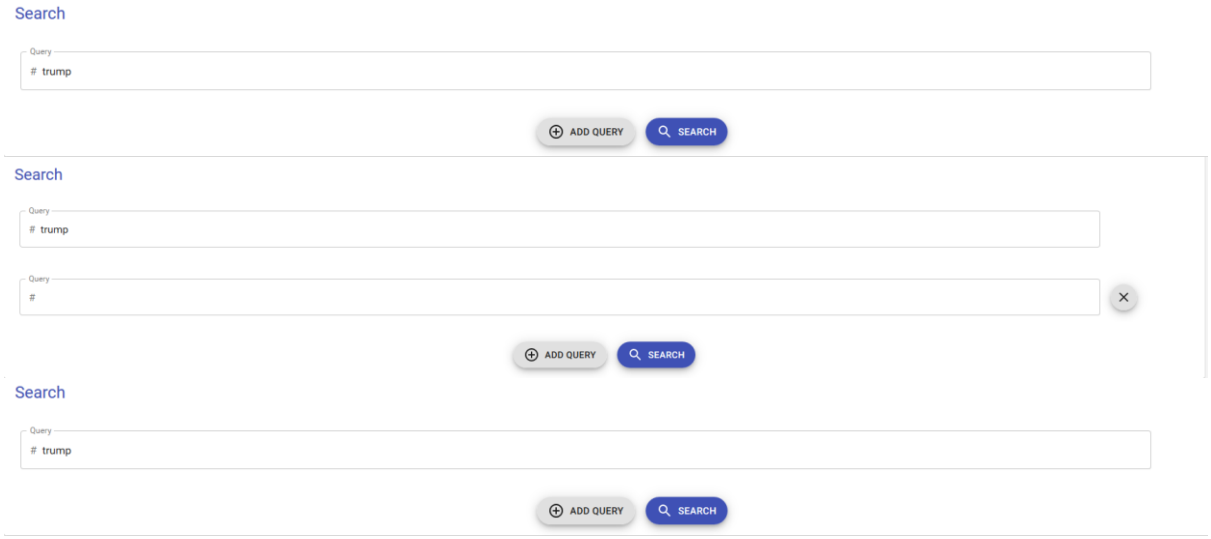
6B – Multiple queries displays on the graph



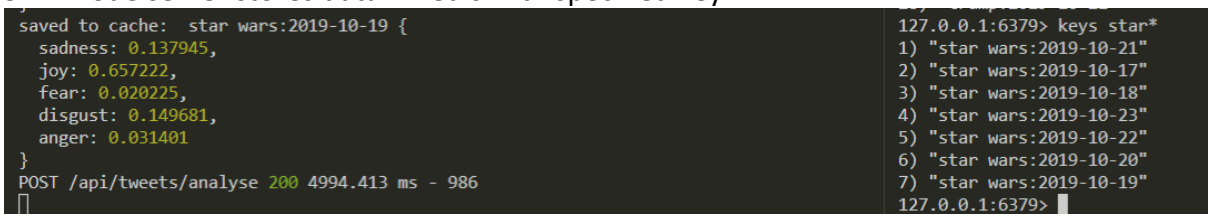
7A – Initiate search with more than one topic

```
POST /api/tweets/analyse 200 5142.672 ms - 1204
[ 'trump', 'Clover', 'kawhi' ]
```

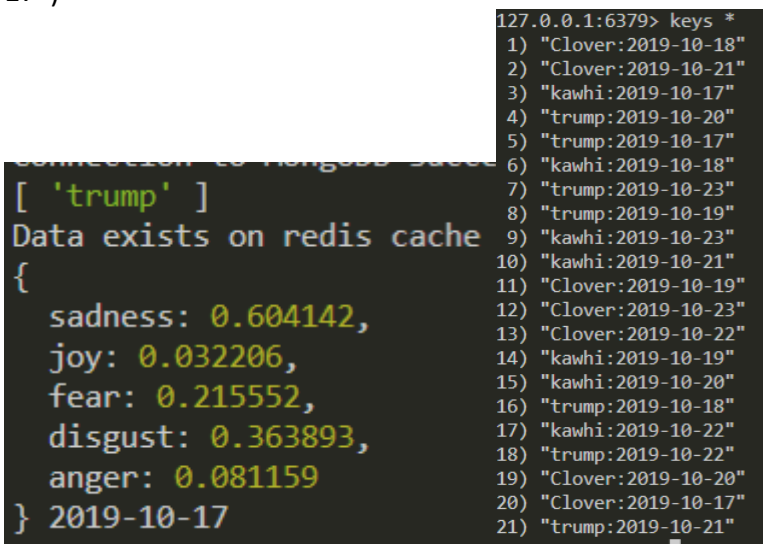
8A – Add for search query, Delete search row



9A – Node server stores data in redis with specified key

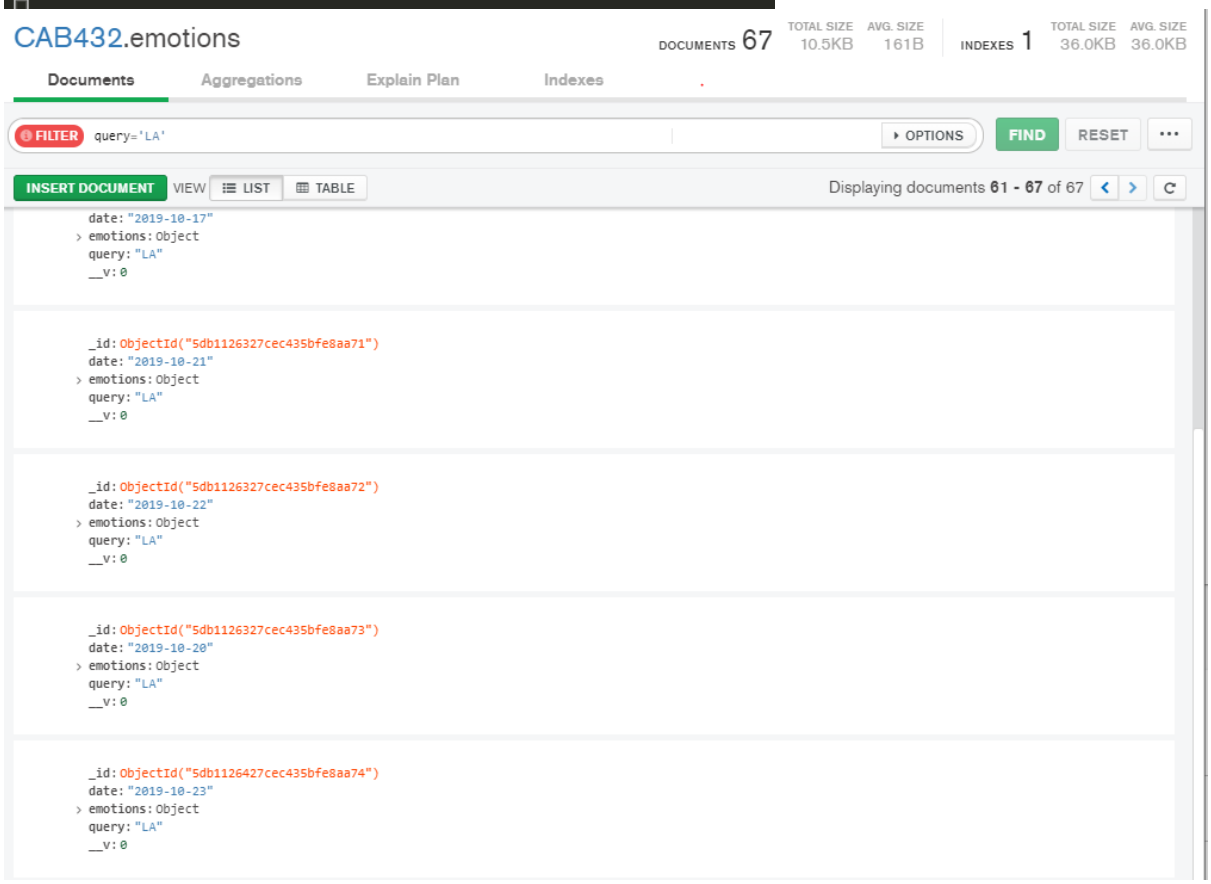


10A – Node server retrieves data from Redis by key specified. In this case ("trump:2019-10-17")



11A – Node server stores data in mongoDB

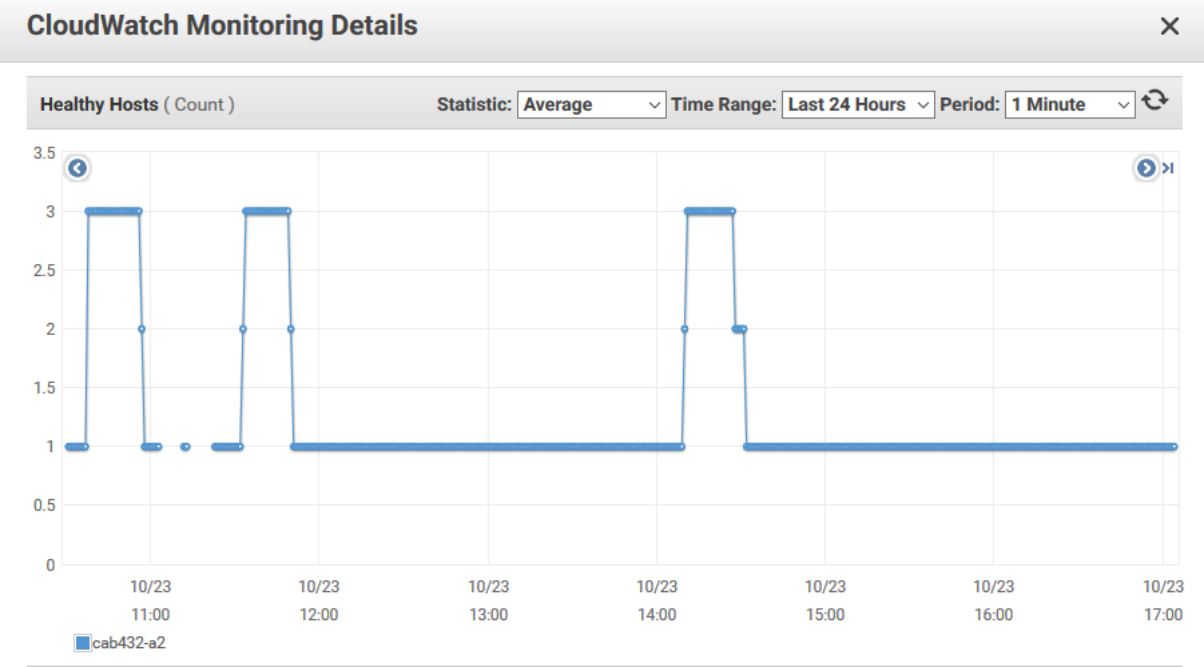
```
saved to db:  {
  date: '2019-10-23',
  query: 'LA',
  emotions: {
    sadness: 0.619207,
    joy: 0.250376,
    fear: 0.02693,
    disgust: 0.199004,
    anger: 0.013275
  }
}
POST /api/tweets/analyse 200 3966.349 ms - 934
```



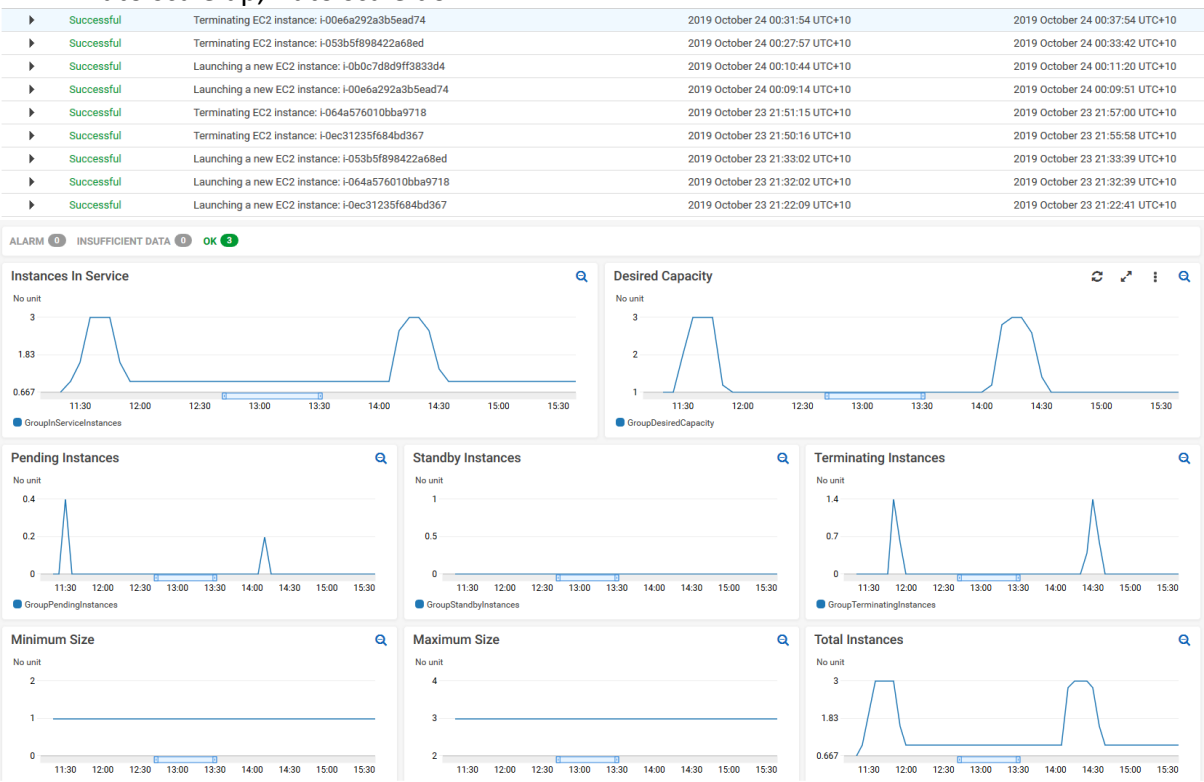
12A – Node server retrieves data from mongoDB by key specified

```
Data exists on mongodb
{
  date: '2019-10-18',
  emotions: {
    sadness: 0.169736,
    joy: 0.532161,
    fear: 0.07939,
    disgust: 0.502175,
    anger: 0.557925
  },
  _id: 5db1126227cec435bfe8aa6f,
  query: 'LA',
  __v: 0
```

13A – Load balancer health check

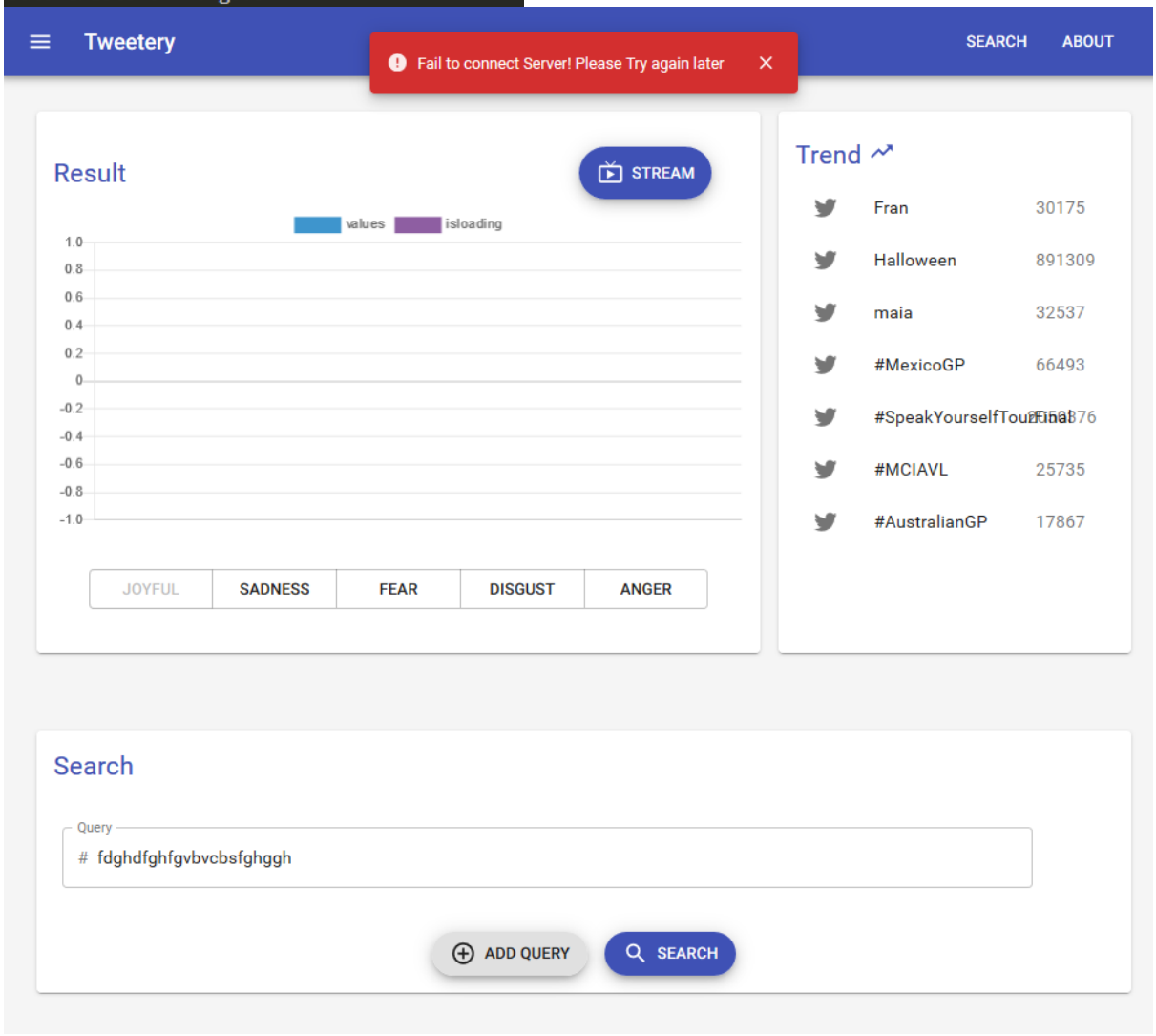


14A – Auto scale up, Auto scale down



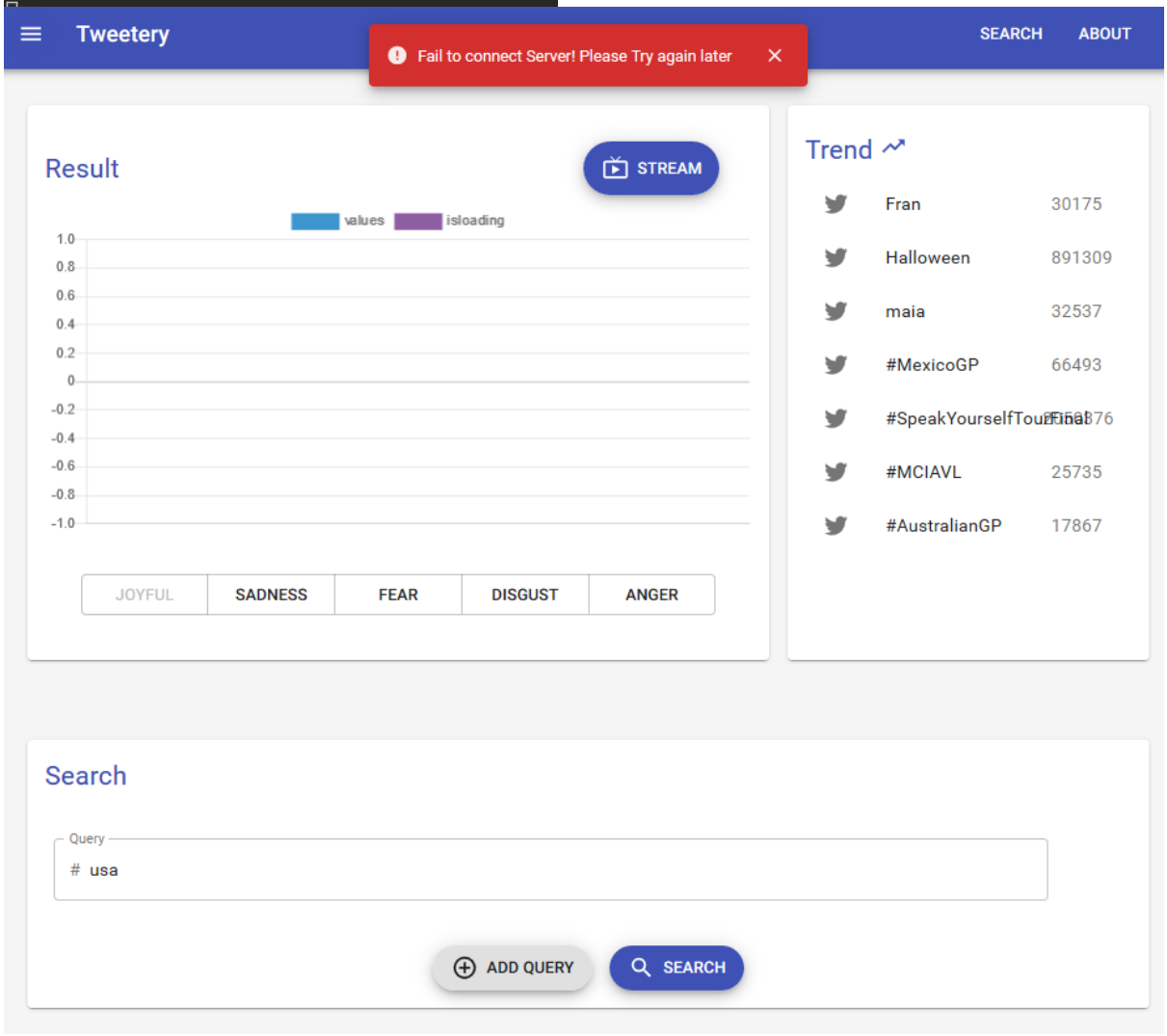
15A – Twitter API. Application continues running (week and stream feature), display error

```
[ 'trump' ]
Data exists on mongodb
Data exists on mongodb
Data exists on mongodb
Data exists on mongodb
Fetch The Raw data and save to cache and db
Data exists on mongodb
Data exists on mongodb
```

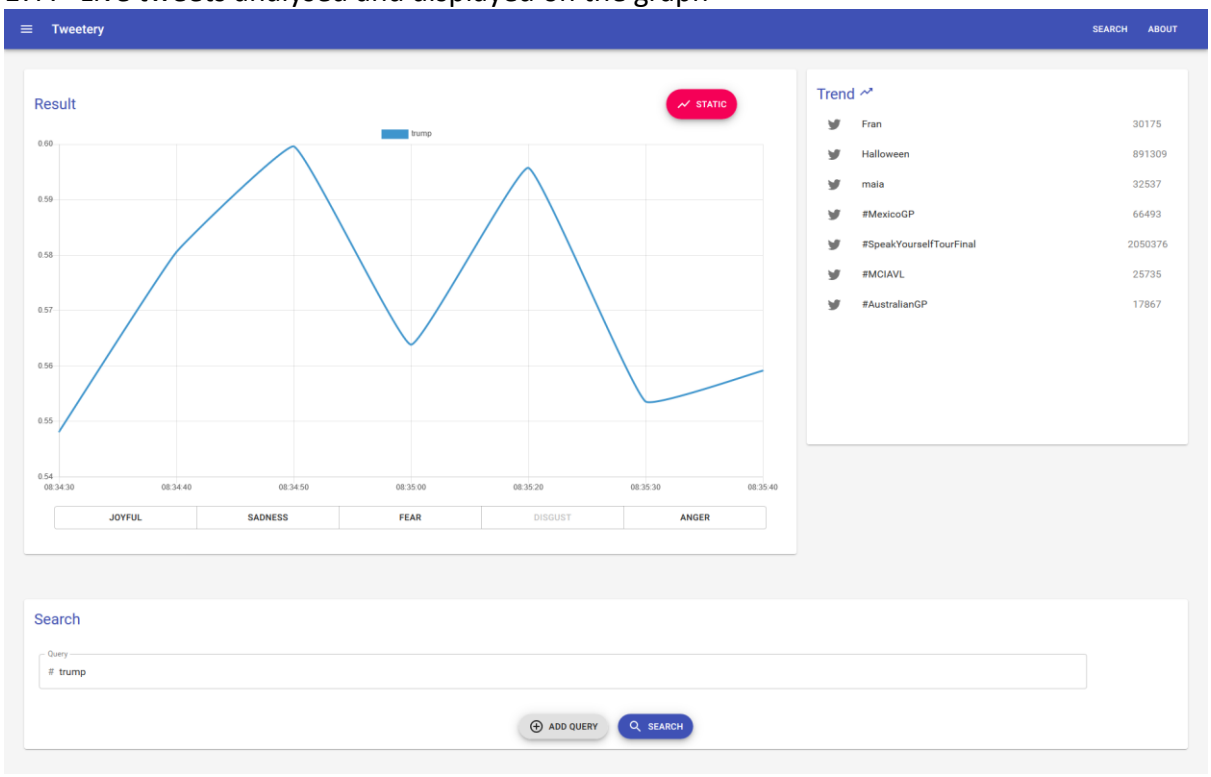


16A – IBM API. Application continues running, display error

```
[ 'usa' ]
Fetch The Raw data and save to cache and db
Fetch The Raw data and save to cache and db
Fetch The Raw data and save to cache and db
Fetch The Raw data and save to cache and db
Fetch The Raw data and save to cache and db
Fetch The Raw data and save to cache and db
Fetch The Raw data and save to cache and db
POST /api/tweets/analyse 404 1640.467 ms - 696
```



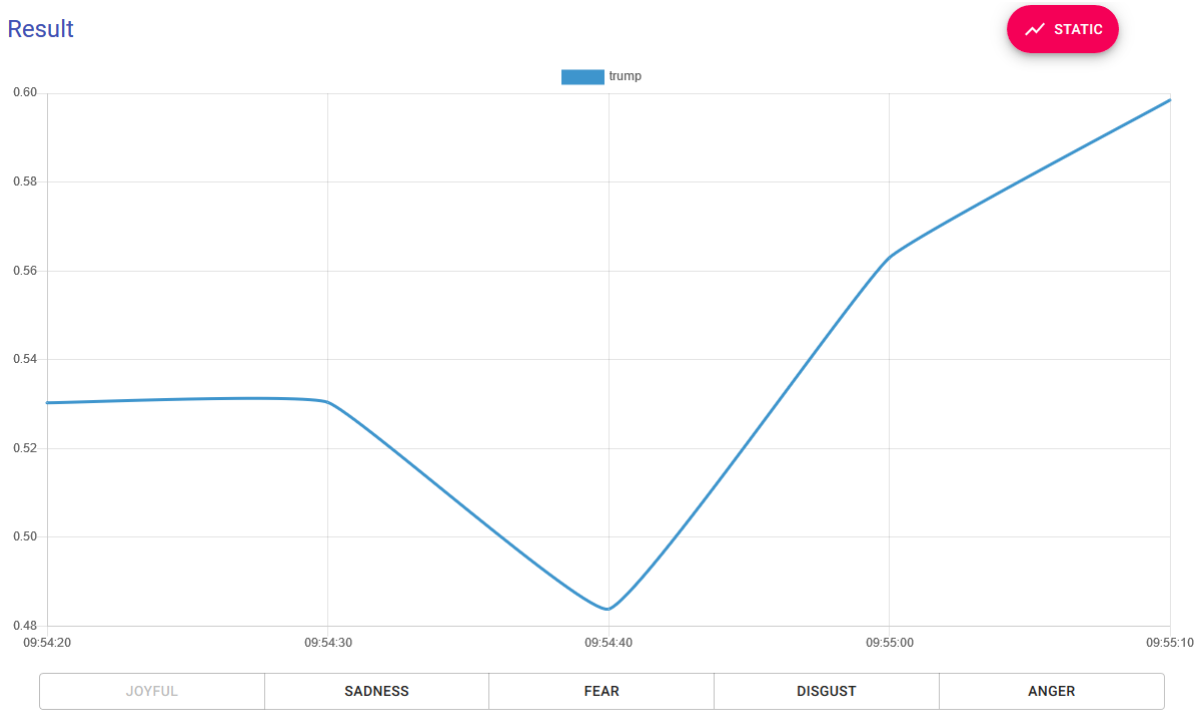
17A - Live tweets analysed and displayed on the graph

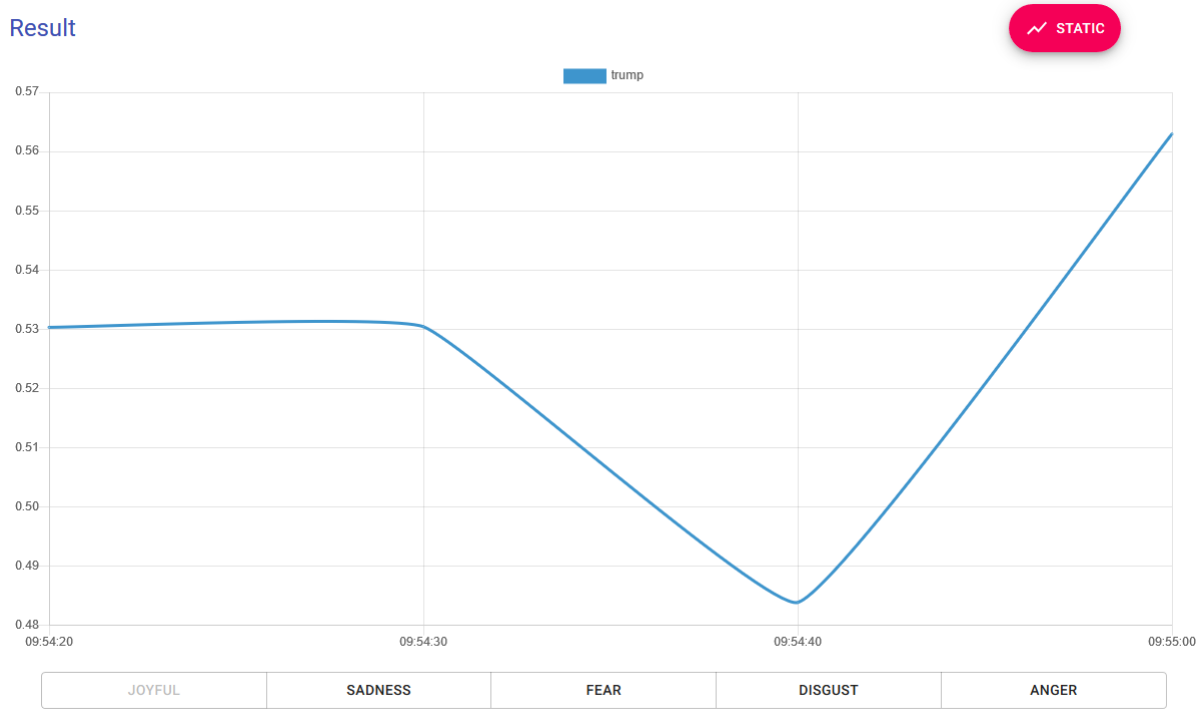


18A - Stream doesn't write after close

```
------------ Socket.io -----------------
Tracking Items :
[ 'usa' ]
0 Connected Users searching: undefined
Closing Stream
------------ Socket.io -----------------
Tracking Items :
[ 'trump' ]
0 Connected Users searching: trump
Closing Stream
GET / 304 0.380 ms - -
GET /static/js/2.c8324f6d.chunk.js 304 0.459 ms - -
GET /static/js/main.953ca3e8.chunk.js 304 0.274 ms - -
GET /static/css/main.2024c935.chunk.css 304 0.240 ms - -
GET /api/tweets/trends 200 194.841 ms - 6896
```

19A - Graph updates every 10 seconds





20A – Stream can be initiated after closing a stream