2021

Peter Nguyen & Andrew Nguyen

N9698094 & N10408673

1/1/2021

Twitter Sentiment Radar

Graphical user interface, text

Description automatically generated

CAB432 Semester 2

Assignment 2: Cloud Project

**Contents**

[Introduction 2](#_Toc86700466)

[Purpose & Description 2](#_Toc86700467)

[APIs and Services used 2](#_Toc86700468)

[Twitter Standard Search API (v.1.1) 2](#_Toc86700469)

[Redis 2](#_Toc86700470)

[AWS EC2 2](#_Toc86700471)

[AWS S3 2](#_Toc86700472)

[Natural 3](#_Toc86700473)

[Compromise 3](#_Toc86700474)

[Use cases 3](#_Toc86700475)

[Use Case 1 3](#_Toc86700476)

[Technical and Architerual breakdown 4](#_Toc86700477)

[Architectural Details 4](#_Toc86700478)

[Architectural Diagram 9](#_Toc86700479)

[Response filtering / data object correlation 10](#_Toc86700480)

[Scaling and Performance 11](#_Toc86700481)

[Test plan 12](#_Toc86700482)

[Difficulties / Exclusions / unresolved & persistent errors / 13](#_Toc86700483)

[Extensions 13](#_Toc86700484)

[User guide 14](#_Toc86700485)

[References 14](#_Toc86700486)

[Appendices 14](#_Toc86700487)

[Test screenshots 14](#_Toc86700488)

[1: Query with Data in Redis and S3 14](#_Toc86700489)

[2: Make a query that uses Redis data only 15](#_Toc86700490)

[3: Make a query that uses S3 Data only (after failing to find it from Redis) 15](#_Toc86700491)

[4: Make a query that uses data from Twitter API after failing to retrieve from both Redis and S3 16](#_Toc86700492)

[5: Query with using special characters, numbers, and letters ($BTC) 17](#_Toc86700493)

[6: Front end working 19](#_Toc86700494)

# Introduction

## Purpose & Description

Twitter Sentiment Radar (TSR) is a web application that allows users to see the emotional rating or sentiment of their desired twitter topic. This information will be displayed on a live graph that measures the sentiment value between 1 and -1. To elaborate, 1 represents that the tweet about the topic was presented positively (happy) and 0 is a negative sentiment value (sad/angry).

## APIs and Services used

The following APIs and Services were used within the NodeJs environment. All necessary packages were retrieved through their appointed versions with Node package manager.

### Twitter Standard Search API (v.1.1)

Returns a collection of relevant Tweets matching a specified query – may also be filtered based on popularity or geocoding [and whatever other obvious details we might decide to include]

Endpoint: https://stream.twitter.com/1.1/statuses/filter.json?track=query

Docs: <https://developer.twitter.com/en/docs/twitter-api/v1/tweets/search/api-reference/get-search-tweets>

### Redis

Redis is an in-memory data structure storage that is used as a form of distributed cache. This is the first layer within the Persistence hierarchy in Twitter Sentiment Radar.

Docs: <https://github.com/redis/node-redis>

### AWS EC2

AWS EC2 is Amazon’s cloud service in which the application will be hosted on.

Docs: <https://docs.aws.amazon.com/ec2/index.html>

### AWS S3

Amazon S3 is an object storage service offered by Amazon Web Services. It acts as the second layer within the Persistence hierarchy in Twitter Sentiment Radar.

Docs: <https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html>

### Natural

Natural is a natural language package for NodeJS. It contains functions that allow developers to tokenise, classify, stem, etc.

Docs: <http://naturalnode.github.io/natural/>

### Compromise

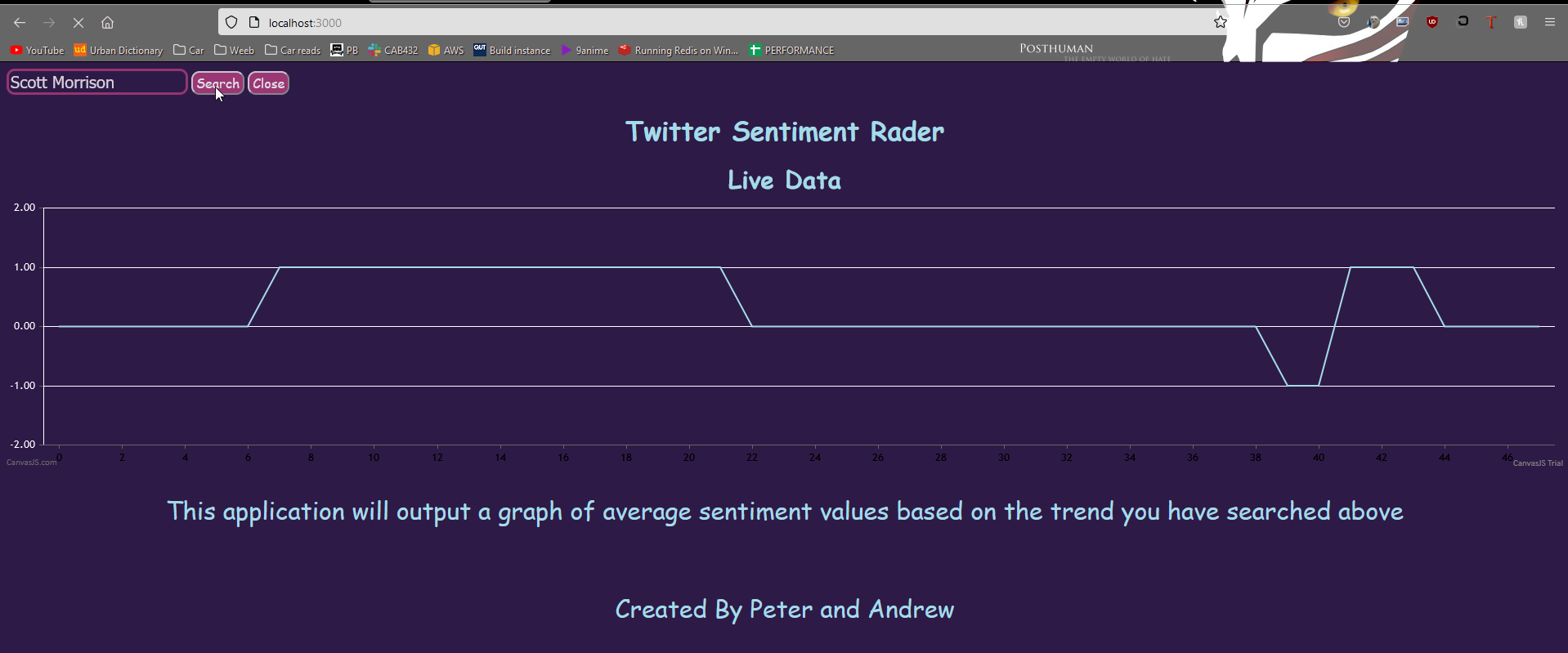
Compromise is a text parser tool that allows developers to conveniently modify, filter, and match select words and phrases within their string.

Docs: <https://github.com/spencermountain/compromise>

# Use cases

## Use Case 1

|  |  |
| --- | --- |
| As a | Twitter user |
| I want | To see sentimental values of my desired tweet of topic |
| So that | I can understand how people are feeling about that topic |

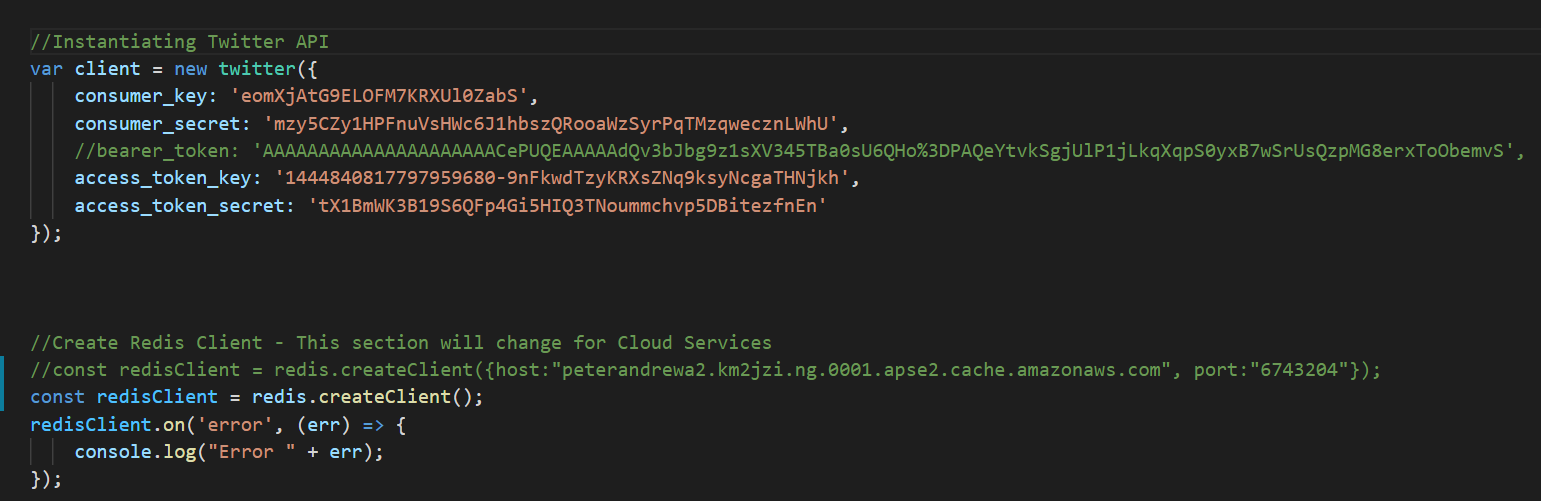


# Technical and Architerual breakdown

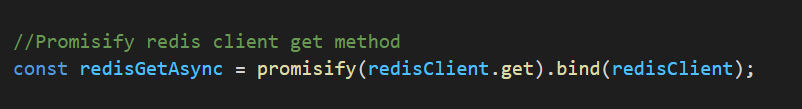
In this report – which covers the group components of the assignment - there should be some coverage of the architecture and the basic operation of the system. Some deeper analysis is now left to the individual report – please read that template and guide for details.

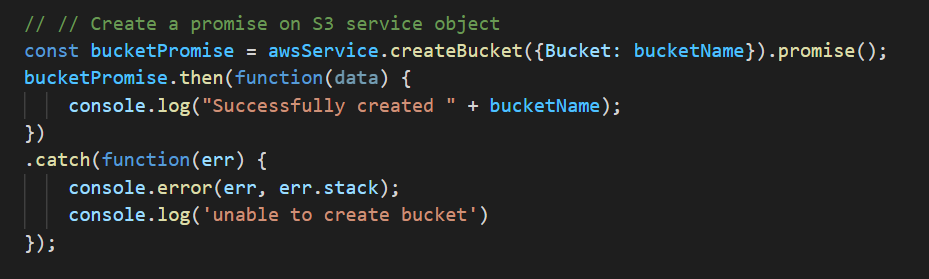
## Architectural Details

The back end of this web application starts with the initial instantiation and declaration of all the variables, libraires, and API clients like S3 Bucket and Redis storage.

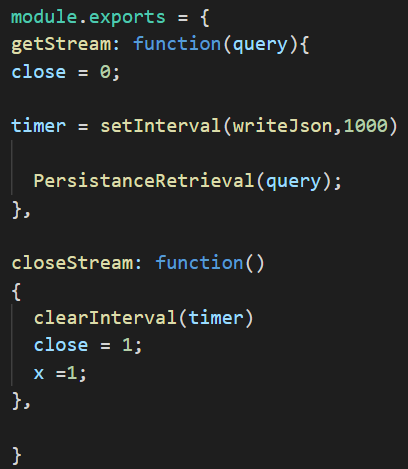
****

Promises are also declared early.

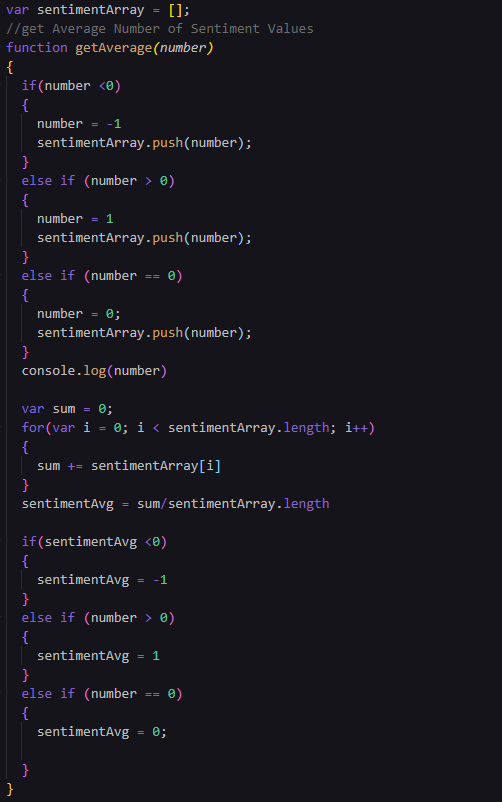


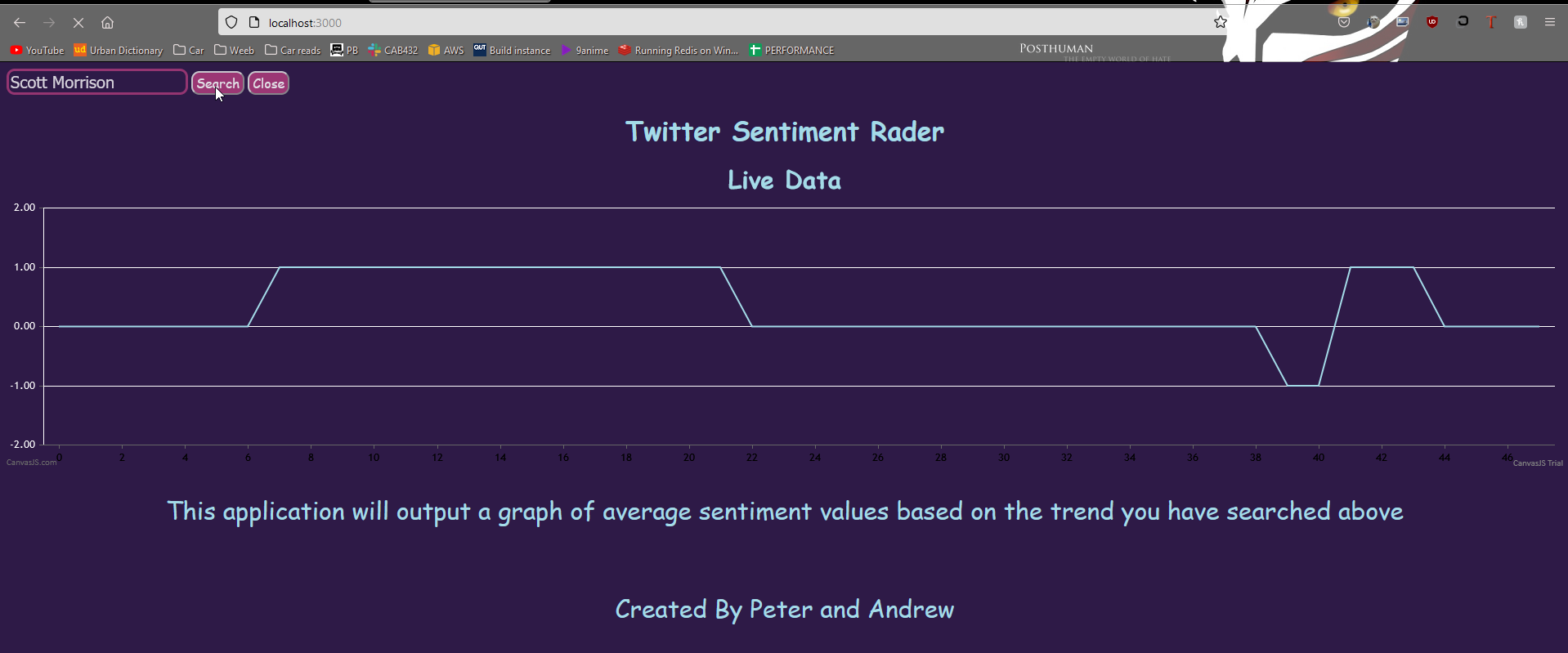
****

Most of the back end is wrapped with Module.Exports to send values in and around the project directory to where the front end requires.

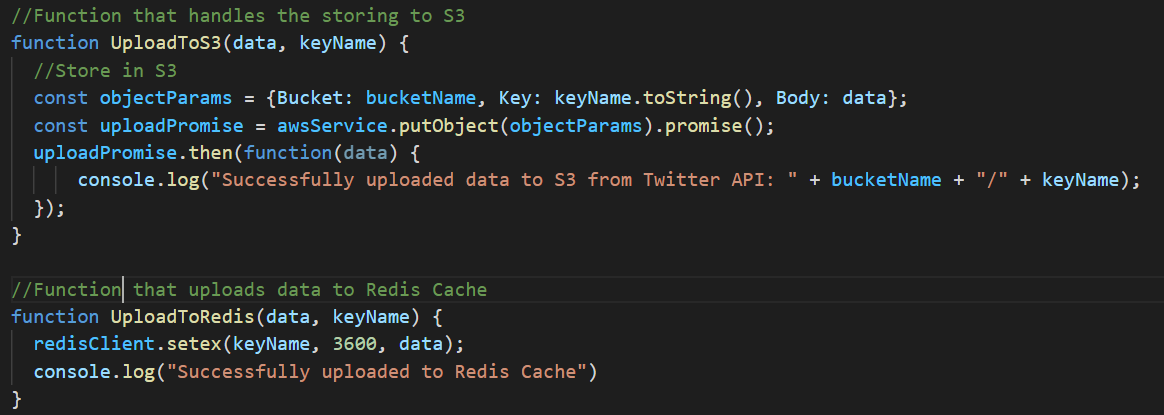


After the sentiment analysis was completed for a singular tweet, the tweet was parsed through a function which scales the values to -1,0 or 1. Since the values that the sentiment analysis produced were very small and not a whole number, this function was required for graph use. If the value of the analysis is below 0, then scale to -1. If it is 0 then leave it at 0, if it is more than 0 then scale it up to one. As shown Below

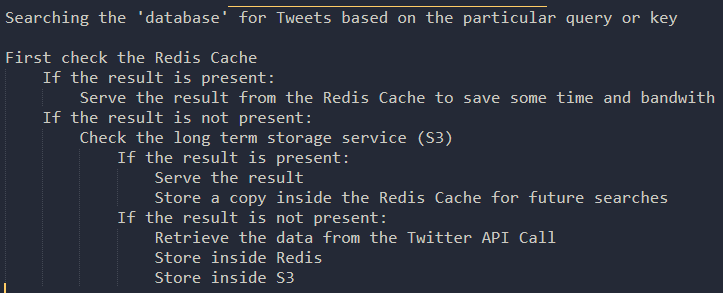
****



Since there will be frequent occurrences of uploading Tweets. S3 and Redis had their code wrapped in a function for convenience.



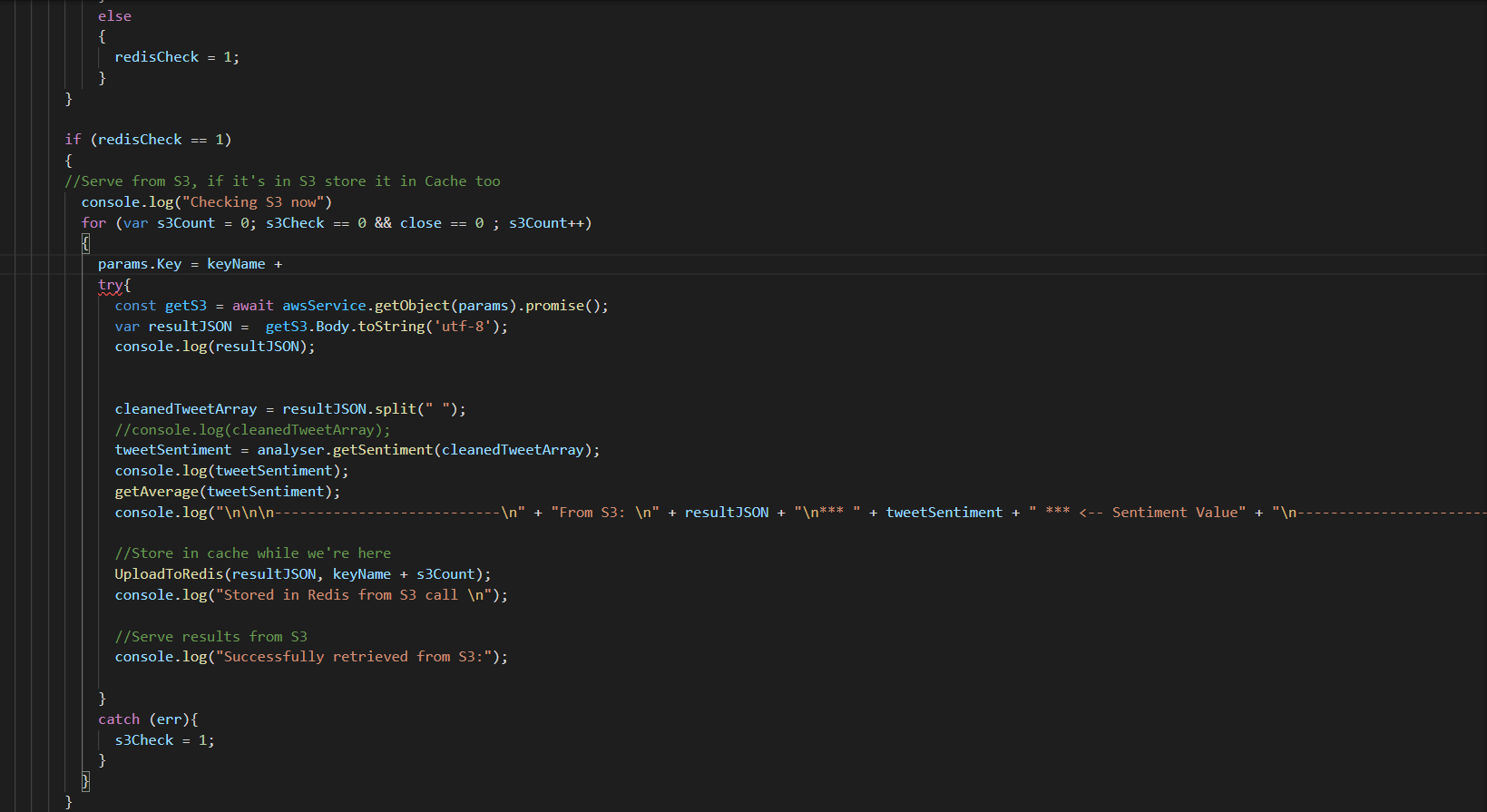
The following screenshots shows the main block of the back end. It is responsible for handling the retrieval and storage of the tweets as well as the parsing and sentiment analysis. This is all done while adhering to the persistence hierarchy order:



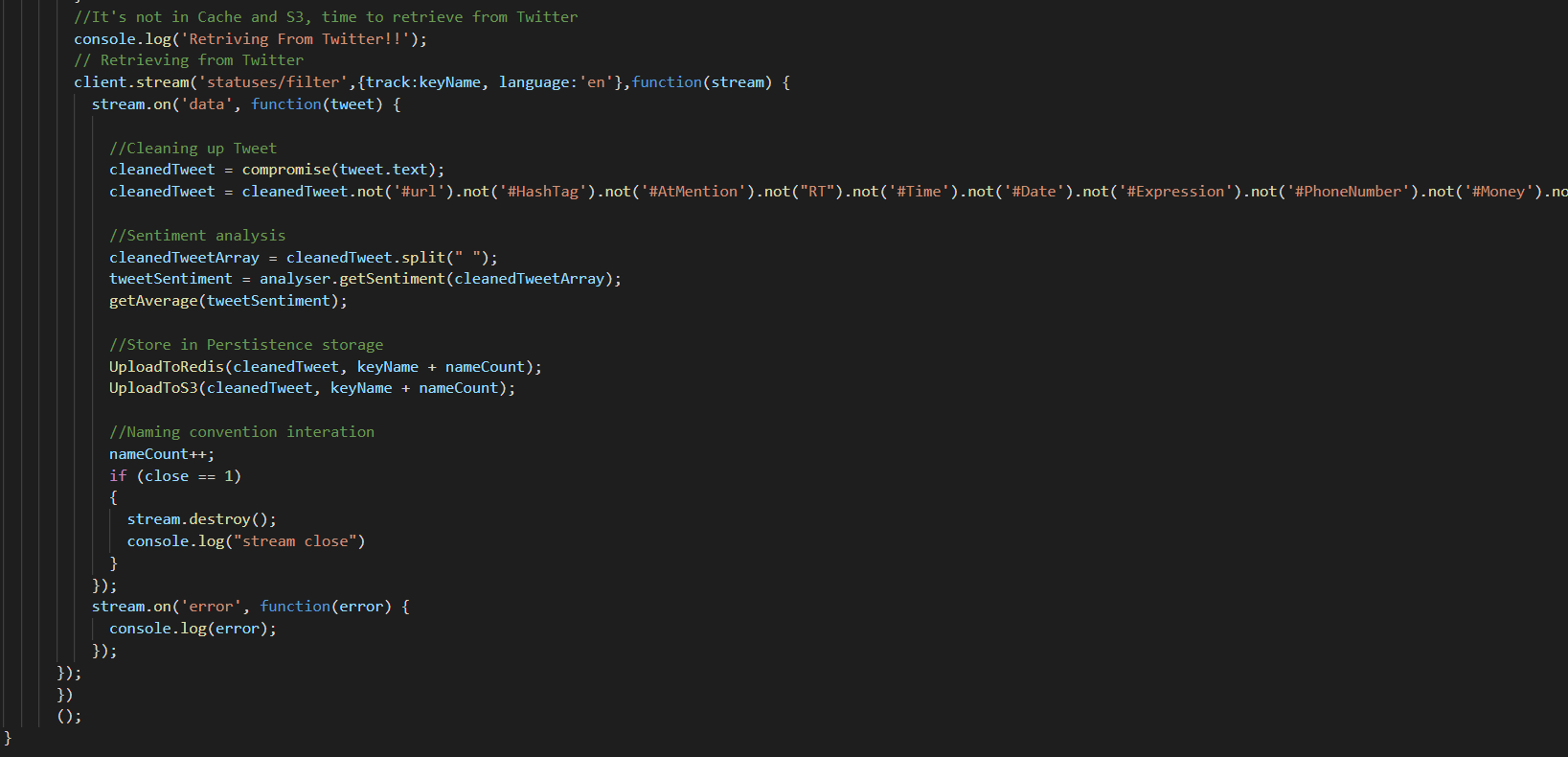
Persistence Code 1 / 3: Nested statement with boolean checks determining what route to take as well as performing parsing and sentiment analysis on retrieved Tweet. Redis.



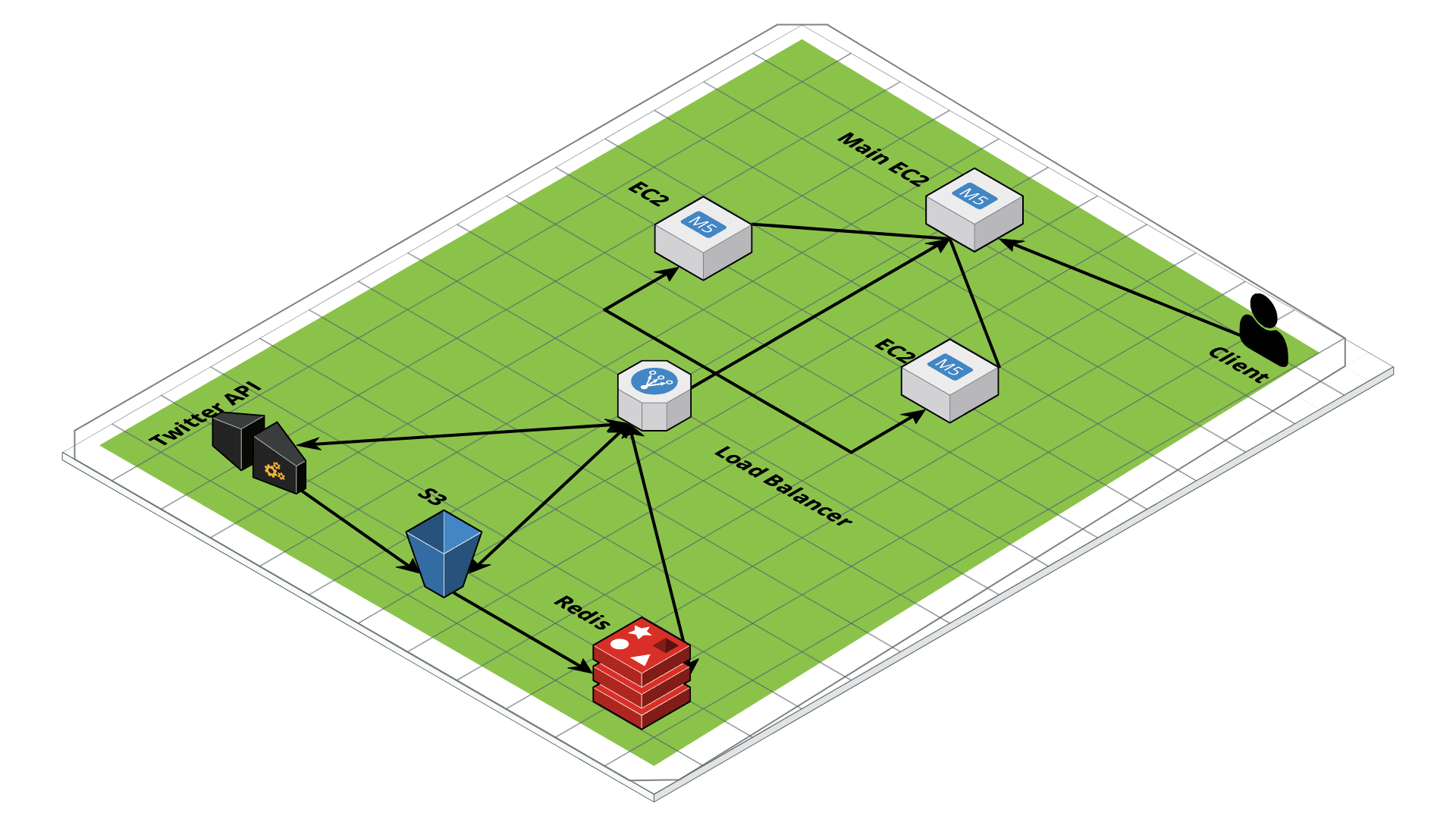
Persistence Code 2 / 3: Nested statement with boolean checks determining what route to take as well as performing parsing and sentiment analysis on retrieved Tweet. S3.



Persistence Code 3 / 3: Nested statement with boolean checks determining what route to take as well as performing parsing and sentiment analysis on retrieved Tweet. Twitter.



## Architectural Diagram

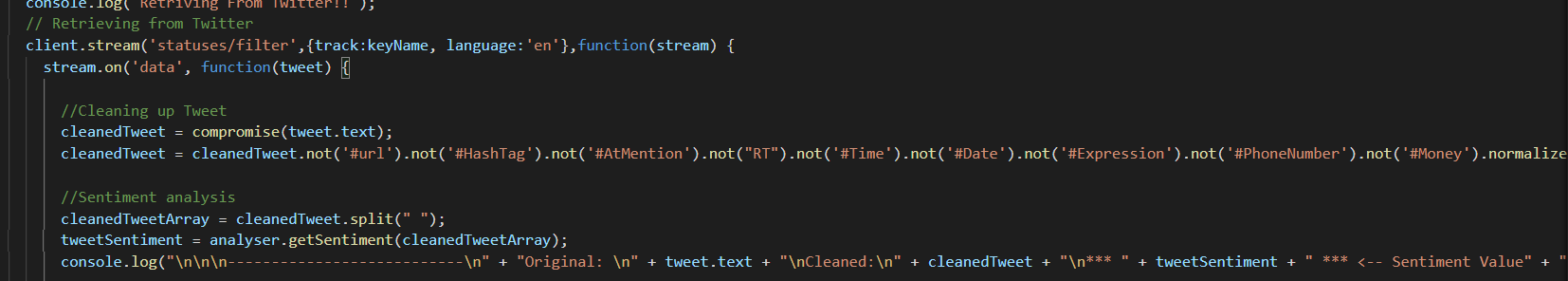


This Architectural Diagram shows a top down view of not only how the Twitter API, S3 storage, and Redis cache interacts with one another to apply persistence, but also how data flows between the three sources into the load balancer which handles the scaling. This procedure is initiated by the Client entering in their query.

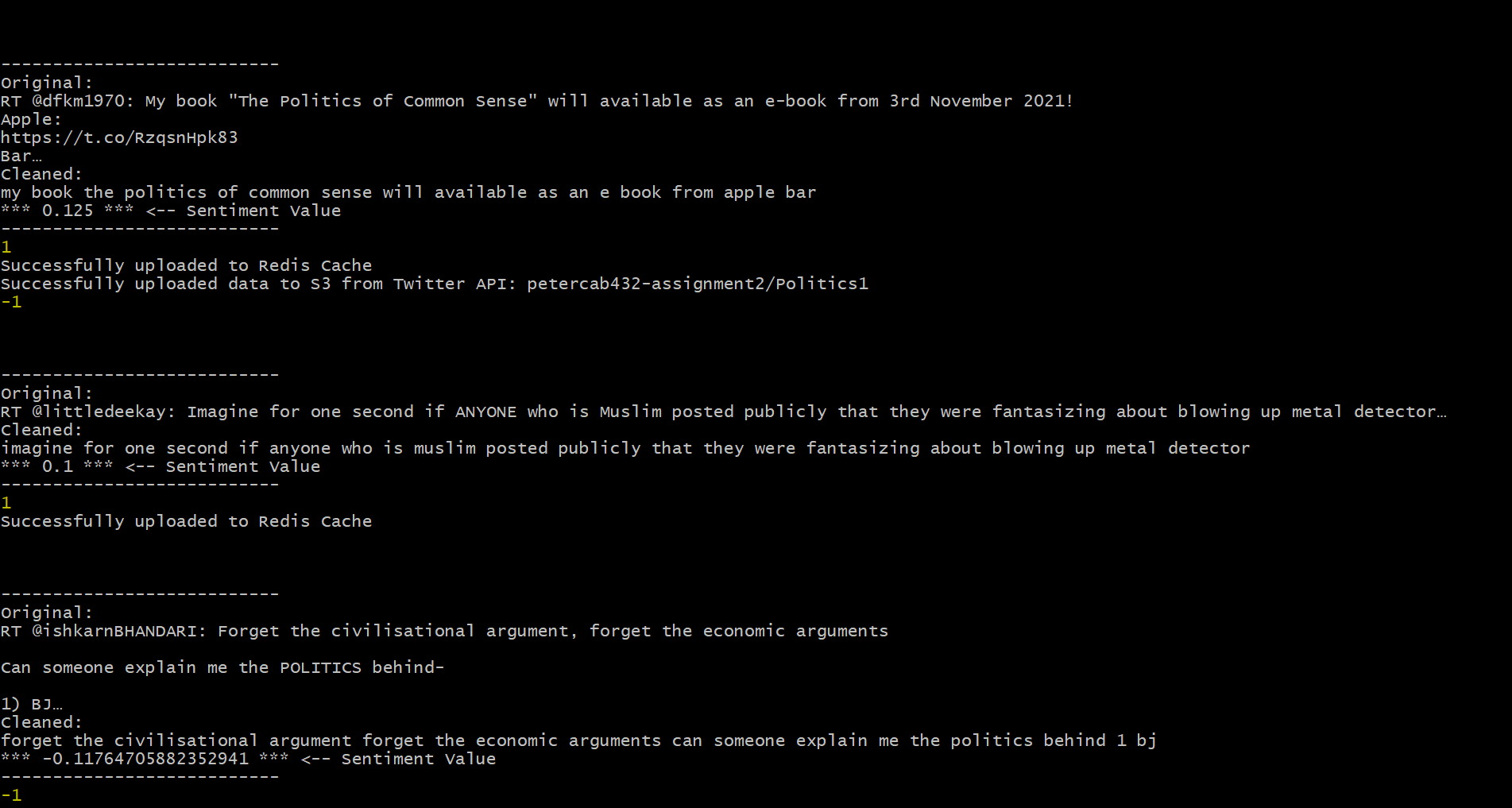
## Response filtering / data object correlation

When a Tweet is retrieved from the Twitter API it goes through a filtering and parsing process. This procedure allows the Tweet to be “cleaned” making sentiment analysis not only less prone to error but more accurate.

After filtering and parsing. The tweet is then split into words and each word is stored as a single element in an array. This is a necessary input format for sentiment analysis.

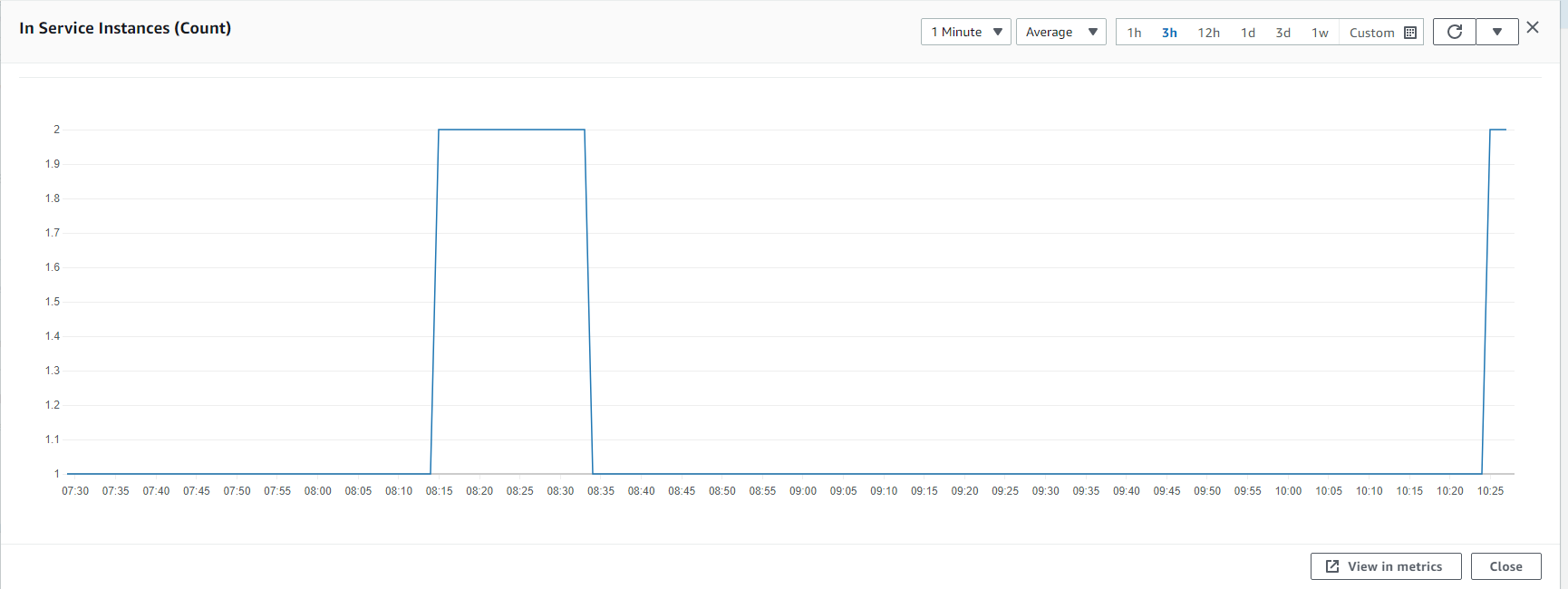


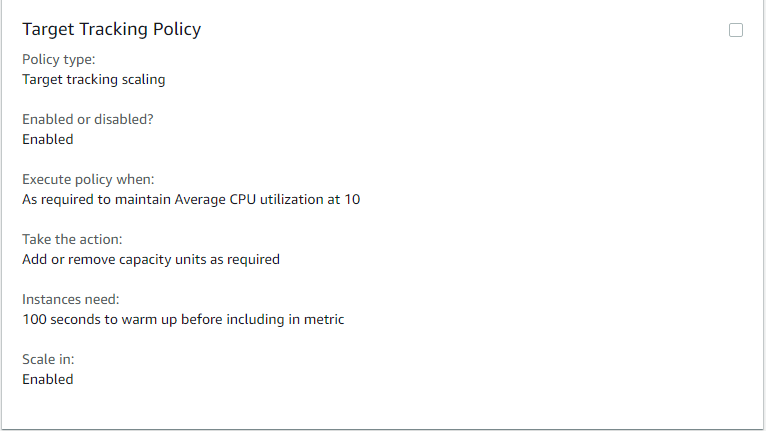
The above code outputs the following console log validating the clean, parsing, and sentiment analysis procedure.

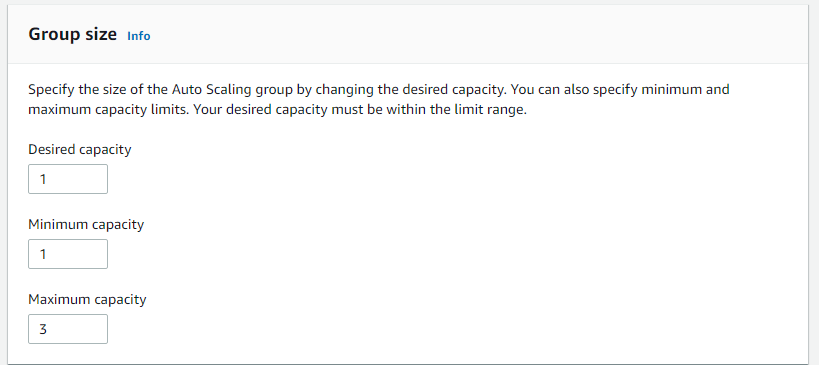


## Scaling and Performance

In terms of scaling, we have set the max amount of instances to 3 with a minimum of 1. Our CPU utilization was also set to roughly 10% usage before the auto scaler policy was executed. We believed these were the optimal settings for autoscaling with our application in mind. Below you can see that when there is only one tab of our application running, only one instance is used. Once we start another query on a separate window, the auto scaling kicks and uses a second instance. Once we cleared all the queries and stopped the application from searching redis, s3 and the Twitter API.







## Test plan

Manual testing is fine and our expectations are in line with the example grid below. You can show the results through a screen shot and point us to these from the table.

Your tests should include

* Positive outcome cases
* Negative outcome cases (error scenarios)
* Edge cases
* Non-functional cases

Note that the grid below is unrelated to this application.

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Expected Outcome | Pass? | Outcome and Screenshots |
| Make a query that retrieves data Twitter API and stores it in S3 and Redis | Identical data within Redis and S3 |  | Console log shows that storage is identical in both S3 and Redis.  View Appendices -> Test Screenshots -> 1: |
| Make a query that uses data from Redis only | Console log within code demonstrating that the query is using data from Redis |  | Console log shows data was retrieved from Redis.  View Appendices -> Test Screenshots -> 2: |
| Query with Data usage from S3 only. (Redirected after failing to find data in Redis) | Console log within code demonstrating that the query is using data from S3 |  | Console log shows data was retrieved from S3.  View Appendices -> Test Screenshots -> 3: |
| Query with no data in Redis or S3, go to Twitter API | Occurs with the initial search of a brand new Query. |  | Console log shows code makes it past the boolean checks of S3 and Redis and resorts to getting Tweets from Twitter.  View Appendices -> Test Screenshots -> 4: |
| Query Random combination of letters and numbers | No errors, searches as usual. |  | Tested using $BTC  View Appendices -> Test Screenshots -> 5: |
| Front end: Graph working | Graph should display as per query search |  | Front end working as expected  View Appendices -> Test Screenshots -> 6: |

## Difficulties / Exclusions / unresolved & persistent errors /

While developing our application, some of the major roadblocks included implementing our persistence. Our application keeps checking the S3 bucket sequentially for the keyword but after it fails to check our keyword it would stop our application. Another roadblock within the persistence was implementing all the searching sequentially, in most cases doing this once is acceptable however our application needed to constantly look for keys inside redis and s3 this did not work at first but was resolved by the use of async and promises into our functions

With regards to functionality that was not finished, our application was not able to be built with multiple users/instances in mind, this is apparently on the frontend with the graph was the graph reads a singular JSON file. This JSON file is written by multiple users and affects the client side.

Some bugs that occur with the application include the close button, the close button originally was fully functional however it simply was not able to execute its function when deployed to aws.

## Extensions

In this section, you can tell us if you wish to how you might extend your app and make it better. This is an opportunity to tell us about good ideas that you had that you didn’t have time to tell us about.

## User guide

When the user is directed to the main page, there is a textbox which tells the user to type in their desired trend.



The user then clicks the search button to begin the graph display on the page below. The user can then stop the display of data and refresh by clicking on the close button.

## References

Use a standard approach to referencing – see the guidance at <https://www.citewrite.qut.edu.au/cite/>.

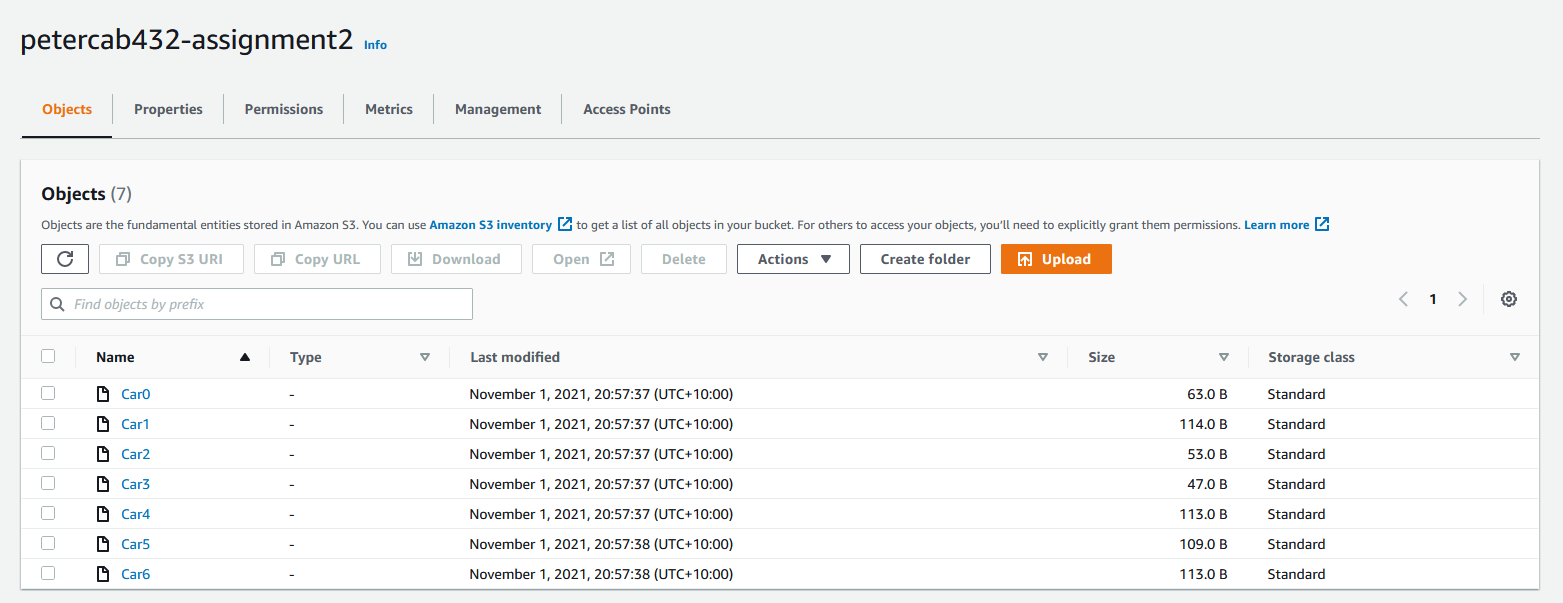
## Appendices

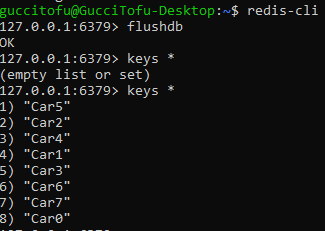
Stuff you want to include, but is too long or too complex to include in the main report text. The full Docker file, some longer excerpt from API docs. Whatever helps.

[Our thanks to those students who allowed us to use their work in the examples presented above.]

### Test screenshots

#### 1: Query with Data in Redis and S3

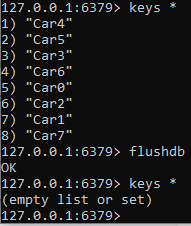


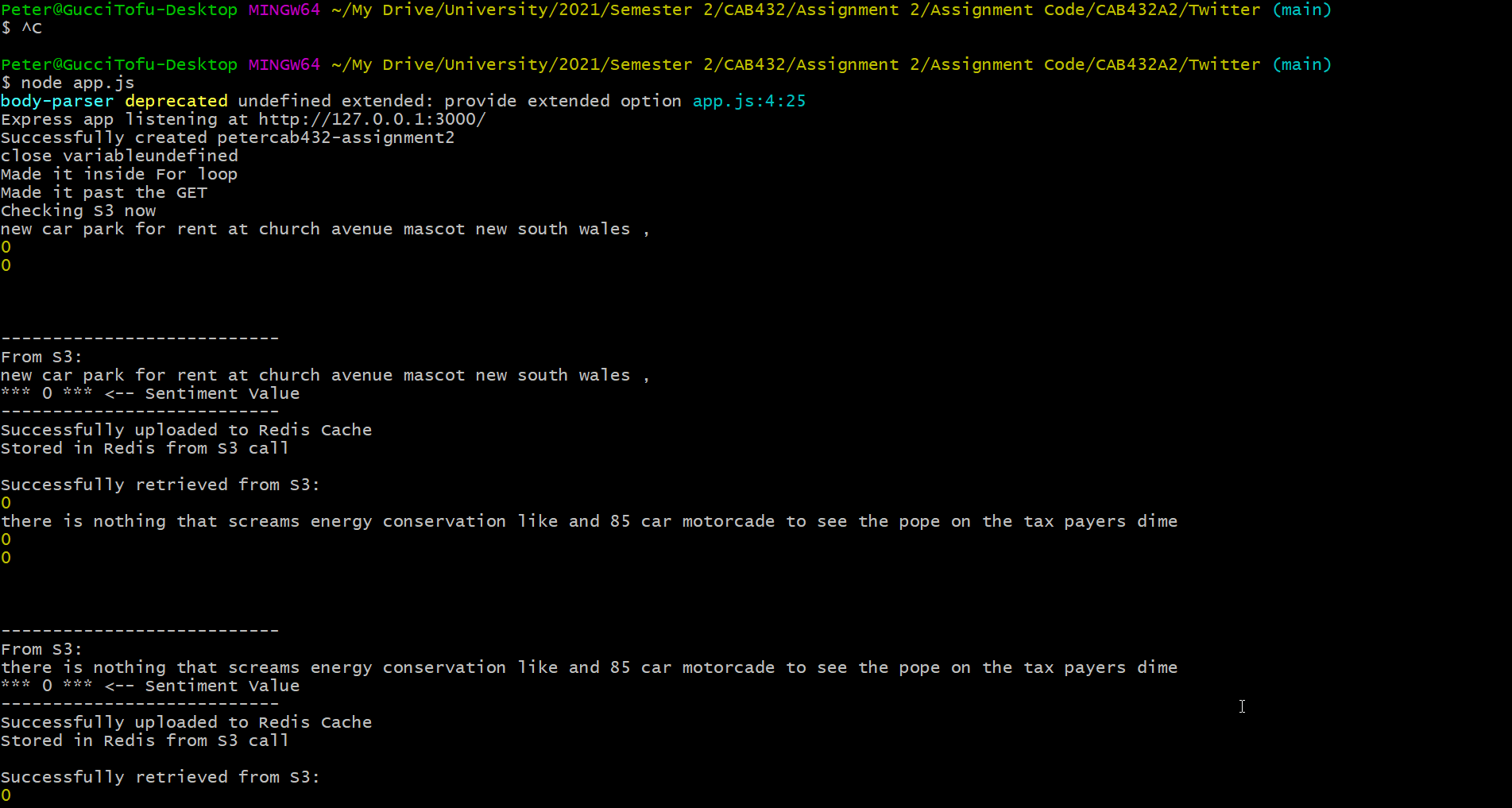


#### 2: Make a query that uses Redis data only

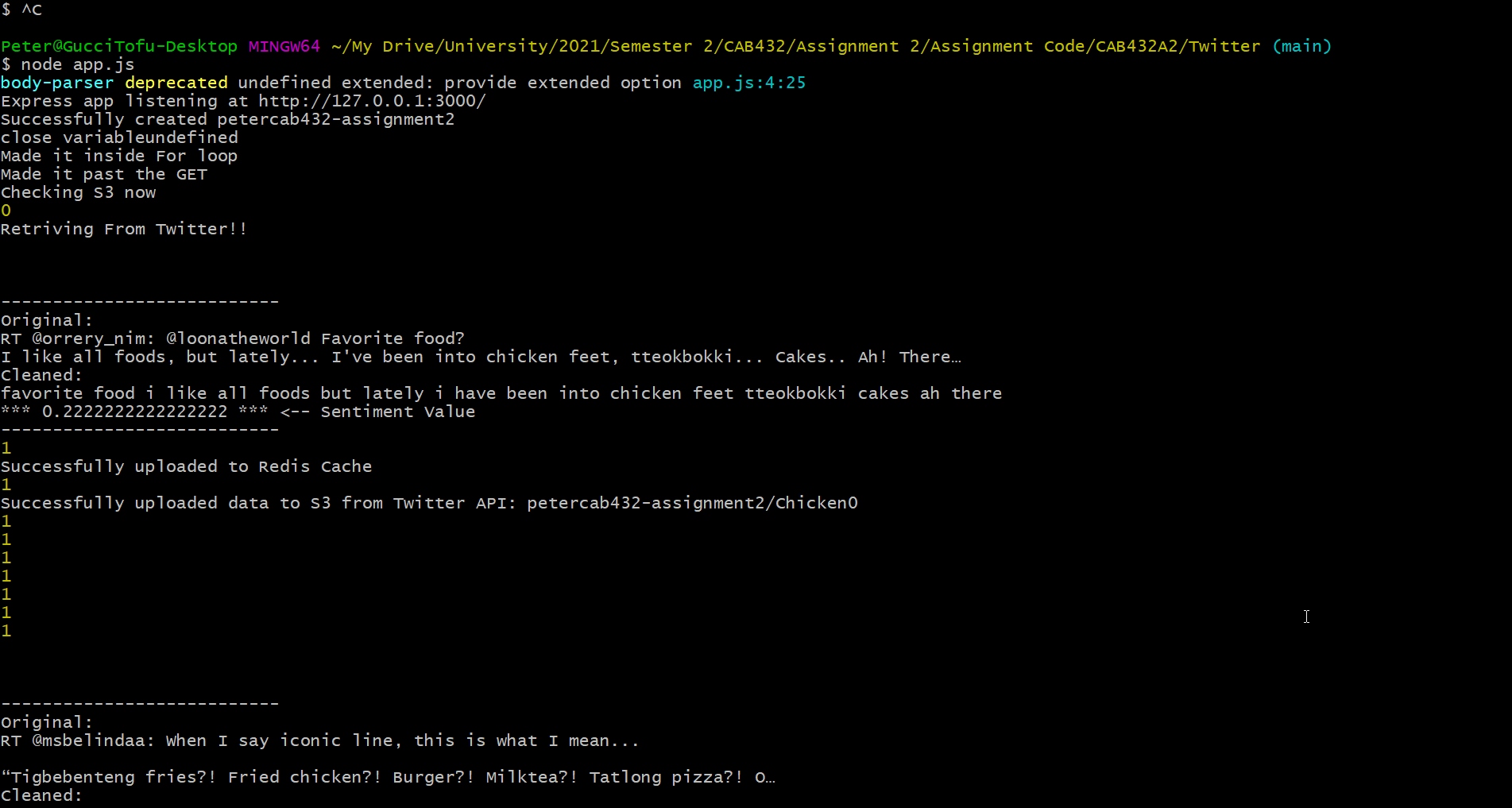


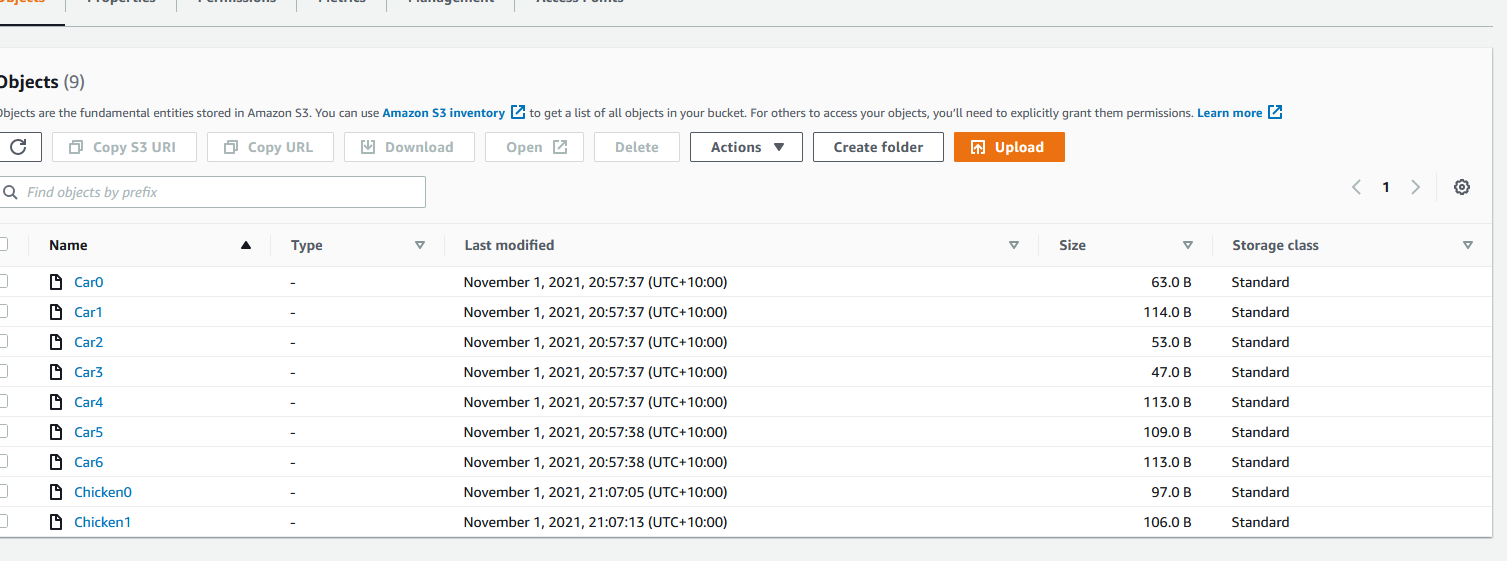
#### 3: Make a query that uses S3 Data only (after failing to find it from Redis)

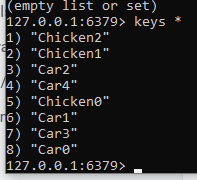




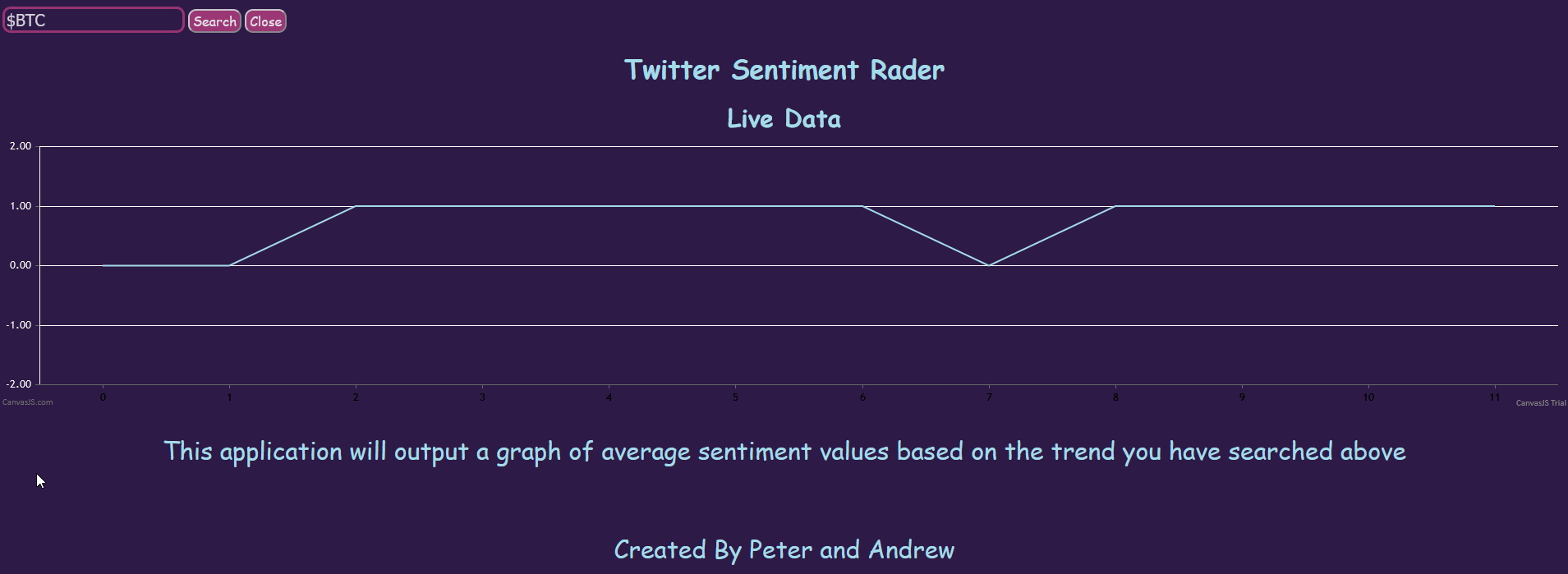
#### 4: Make a query that uses data from Twitter API after failing to retrieve from both Redis and S3

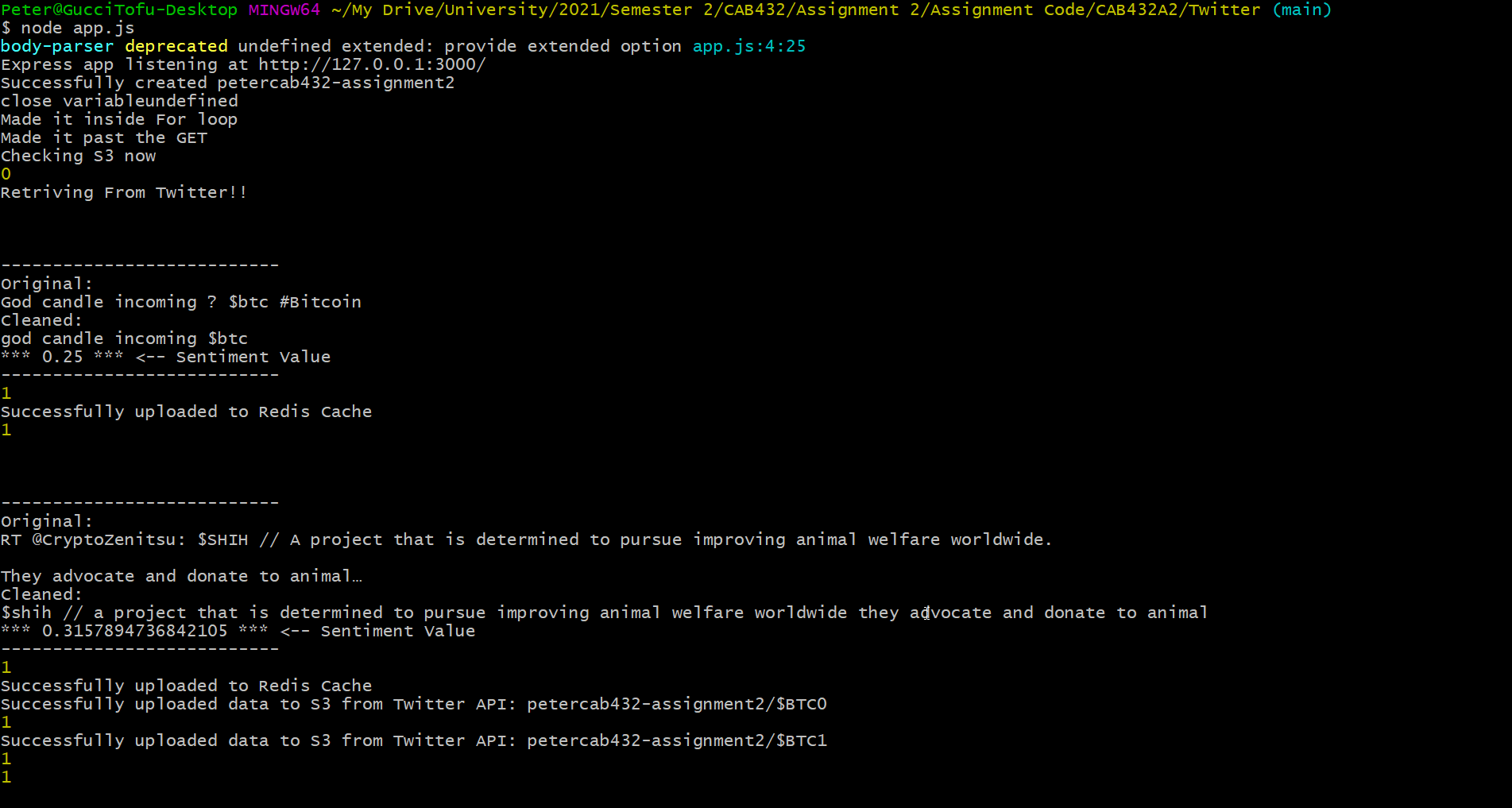


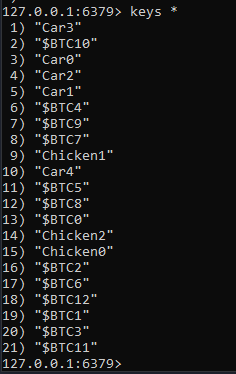


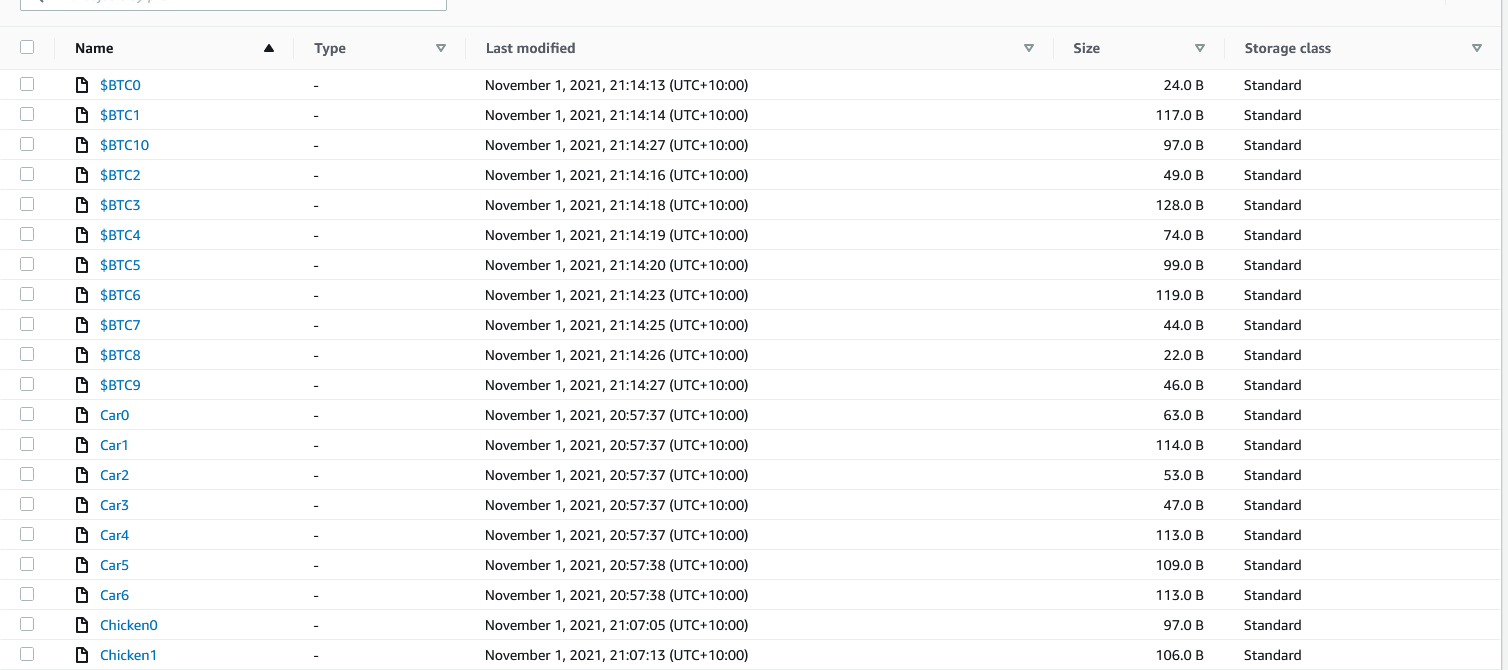


#### 5: Query with using special characters, numbers, and letters ($BTC)









#### 6: Front end working

