

# QUEENSLAND UNIVERSITY OF TECHNOLOGY

## ASSIGNMENT 2: CLOUD APPLICATION

### CAB432 – Cloud Computing

Jeremiah Dufourq, n9960651  
25/10/2020

## Application

**Content (ParallelDots, 2020)**

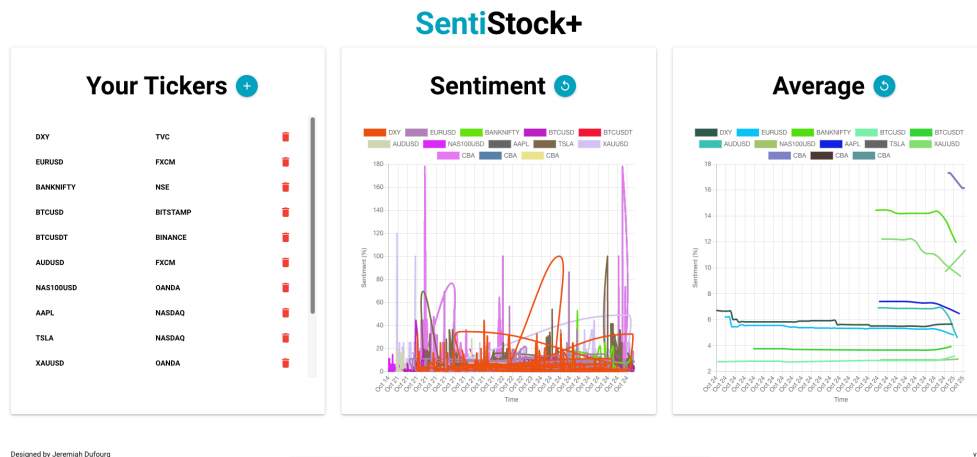
<b>1</b>	<b>Introduction.....</b>	<b>3</b>
1.1	Purpose & Description.....	3
1.2	Services used.....	4
1.2.1	Twitter standard search API (v1.1).....	4
1.2.2	TradingView search API .....	4
<b>2</b>	<b>Use Cases .....</b>	<b>5</b>
2.1	Use case 1 .....	5
2.2	Use case 2 .....	5
2.3	Use case 3 .....	6
<b>3</b>	<b>Technical Breakdown .....</b>	<b>7</b>
3.1	Overall architecture & Data Flow .....	7
3.2	Client side .....	10
3.3	Server side.....	11
3.3.1	Route Layer – APIs/Routes .....	11
3.3.2	Persistence layer .....	13
3.3.3	Business logic layer .....	15
3.4	Dockerizing + serving web application .....	15
3.5	Response filtering/data object correlation .....	16
3.6	Scaling and performance.....	18
<b>4</b>	<b>Test plan.....</b>	<b>20</b>
4.1	Functional Tests .....	20
<b>5</b>	<b>Difficulties &amp; Limitations.....</b>	<b>21</b>
<b>6</b>	<b>References.....</b>	<b>21</b>
<b>7</b>	<b>User guide .....</b>	<b>22</b>
<b>8</b>	<b>Appendix.....</b>	<b>24</b>

## Application

# 1 Introduction

## 1.1 Purpose & Description

SentiStock+ is an extension of the previous application SentiStock. Institutional and retail investors are always seeking tools to improve their choices when it comes picking assets which deliver a high return. SentiStock+ addresses these needs by allowing investors to select from a large range of assets, and to get real time sentiment feedback. The application uses natural language processing to obtain sentiment from a list of tweets on Twitter. This allows the investor to gain insight into how an asset is represented in a public forum. The application consisted of a landing page in which the investor can enter asset tickers on the left (where it will track your previous choices), and then hit the refresh buttons to the right in the two graphs. This will provide an average and collated result of the sentiment for their given tickers.



## 1.2 Services used

### 1.2.1 Twitter standard search API (v1.1)

The project used the Twitter standard search API. Below is a brief statement about the API:

*The Twitter API can be used to programmatically retrieve and analyze data, as well as engage with the conversation on Twitter. This API provides access to a variety of different resources including the following: Tweets, Users, Direct Messages, Lists, Trends, Media, Places (link to twitter reference). (Twitter, 2020)*

Endpoint → <https://api.twitter.com/1.1/search/tweets.json>

Docs → <https://developer.twitter.com/en/docs/twitter-api/getting-started/guide>

### 1.2.2 TradingView search API

Although this API is not specifically documented, it does provide a robust way to validate the tickers which the investor searches on the platform. This API was gathered by looking at the network traffic for TradingView.

Endpoint → [https://symbol-search.tradingview.com/symbol\\_search/](https://symbol-search.tradingview.com/symbol_search/)

You can enter queries to the endpoint as well. In the application, the query 'text' is used to specify the search parameters for a given ticker. The response is an array of possible matches to that query.

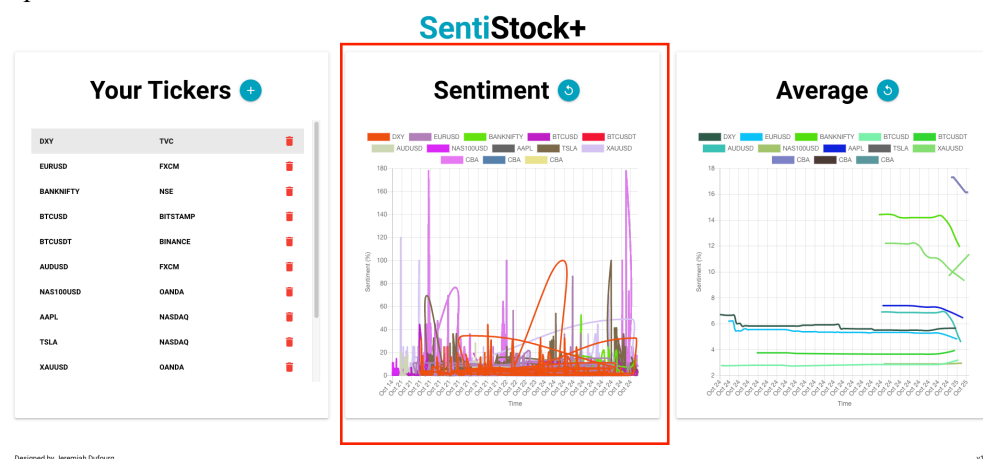
## Application

## 2 Use Cases

### 2.1 Use case 1

As an	Investor
I want	The application to display the sentiment of a stock over a period of time
So that	I can gauge the sentiment of a stock in a public forum

This use case specification was met by creating column on the left side and a plus icon. When clicking this plus icon, a dialog appears so that the user can enter a search query or select from the pre filled out stock tickers. a search box on the home page, as well as a table of a list of available stock tickers. Once selected, this will appear on the left column of the application. The user can then refresh the Sentiment section chart to update the sentiment chart.



In terms of persistence implementation, this use case was implemented by saving the current searched tickers (presented on the left column) to a table. In addition, once the user updates the chart (in the middle column), this will fetch the tweets associated with that 'filter', update the analysis and then save this to a table in the database. Note that this use case obtains all of the possible tweets up to the limit of the Twitter search API. If the user refreshes the page, the application then fetches this data from the database, or if it's already stored in the cache, then it will fetch it from the cache. With regards to scaling, the more unique 'filters' (in this case tickers) the user adds, the more CPU usage the application will use, and hence the scaling policy is suited to reflect this.

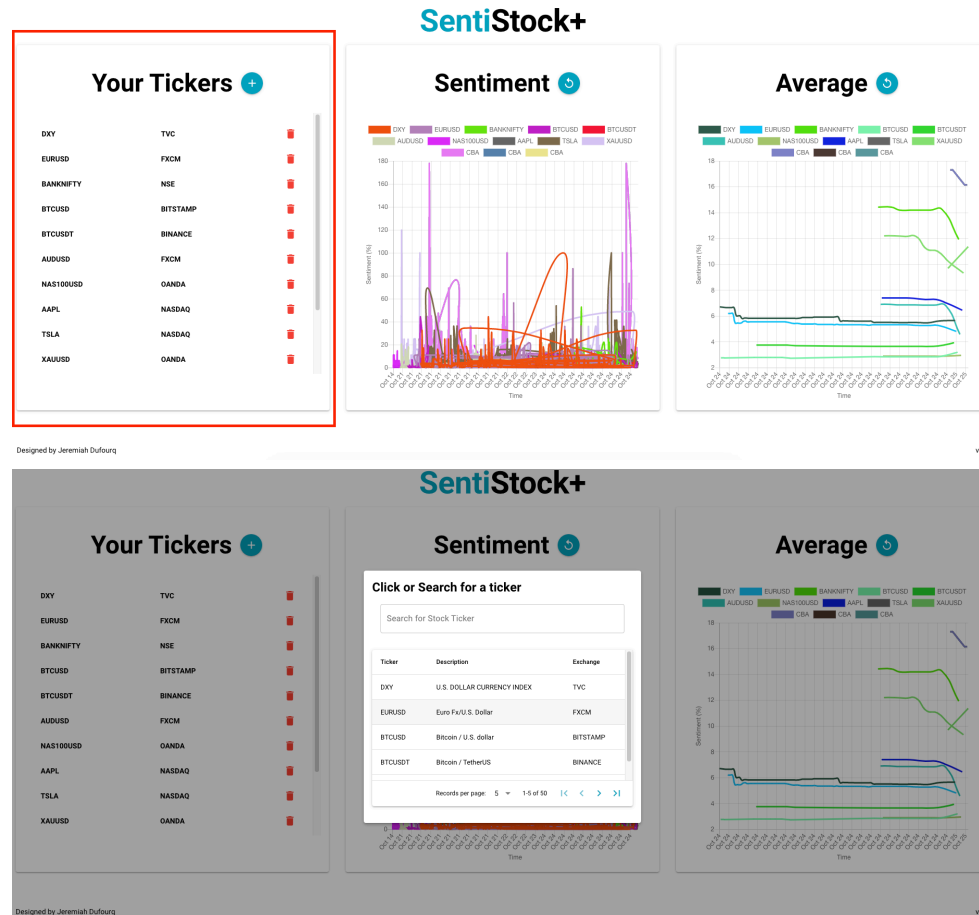
### 2.2 Use case 2

As an	Investor
I want	The system to remember my previous stock ticker searches
So that	I can go back to my analysis

Much similar to the first implementation of this application, the user is able to store the previously searched tickers. This is presented to them in the left column of the application. The user can click on the plus icon, and then they are presented with a table

## Application

and search field where they can fill out the ticker that they require. Once selected, it is stored in the left column on the application. The user also has the option to delete the ticker if they do not need it.



In terms of persistence, when the user enters a ticker and then selects it, this is sent as a request to the server, which then stores this in a table in the database. If the user refreshes the page, the application fetches the data from the database. However, if the data is there in the cache, it will fetch it from the cache instead. This use case has no real contribution to the 'scaling' part of the application, because it isn't related to the computational load. However you could argue that without selecting a ticker 'filter', then the application would not provide any load.

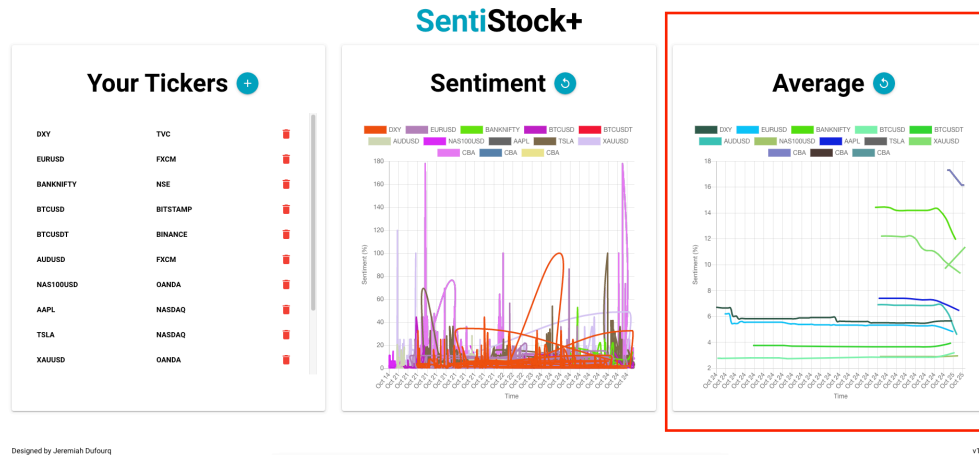
## 2.3 Use case 3

As an	Investor
I want	The system to calculate the historical average and graph it
So that	I can get a quick overview of the average sentiment for a given stock

This use case is somewhat similar to use case 1. However, the main differences to this use case is in regards to load/scaling, and what is presented to the user. In this use case, the user has the ability to see the live average sentiment for a given stock ticker. The user

## Application

must select a range of stock tickers first (using use case 2). After the user has selected these tickers, they can then refresh the graph on the right column of the application. This will then go fetch the average sentiment for those given stock tickers.



With regards to persistence, this is much the same as the first two use cases. The calculated average data is stored in a table in the database. When the user refreshes the page, this data will update depending on if it is in the cache or if it is in the database. The key difference with this use case is scaling. This use case requires more computational power because it is calculating the sentiment for each tweet, in each ticker, and then averaging this. Because of this, it has a big impact in applying load to the application, and hence scaling the application.

### 3 Technical Breakdown

#### 3.1 Overall architecture & Data Flow

There are three main parts to the application, 1) the application layer (how the web application behaves within the scaling pool instances), 2) the persistence layer (how data is managed throughout the application), 3) the scaling layer (how the instances are scaled, and requests are managed). Each of the architectures and data flows are described in detailed in the following sections. This section will cover the overall architecture with some details on data flow.

In a typical usage of this application, the user will hit the main URL provided by the application load balancer. This load balancer will then route the request to the current available instance. The user is then presented with a landing page. This landing page will be a reflection of what is currently in the database. If there is information in the cache,

### Application

then it will load this over the database for a quick response time. If the user add's a ticker, and then refreshes the analysis and average charts, a request will be sent to the server to fetch this information. The sentiment will then be calculated, persisted into the database and cache, and then sent back to the client, where it will be displayed to the user.

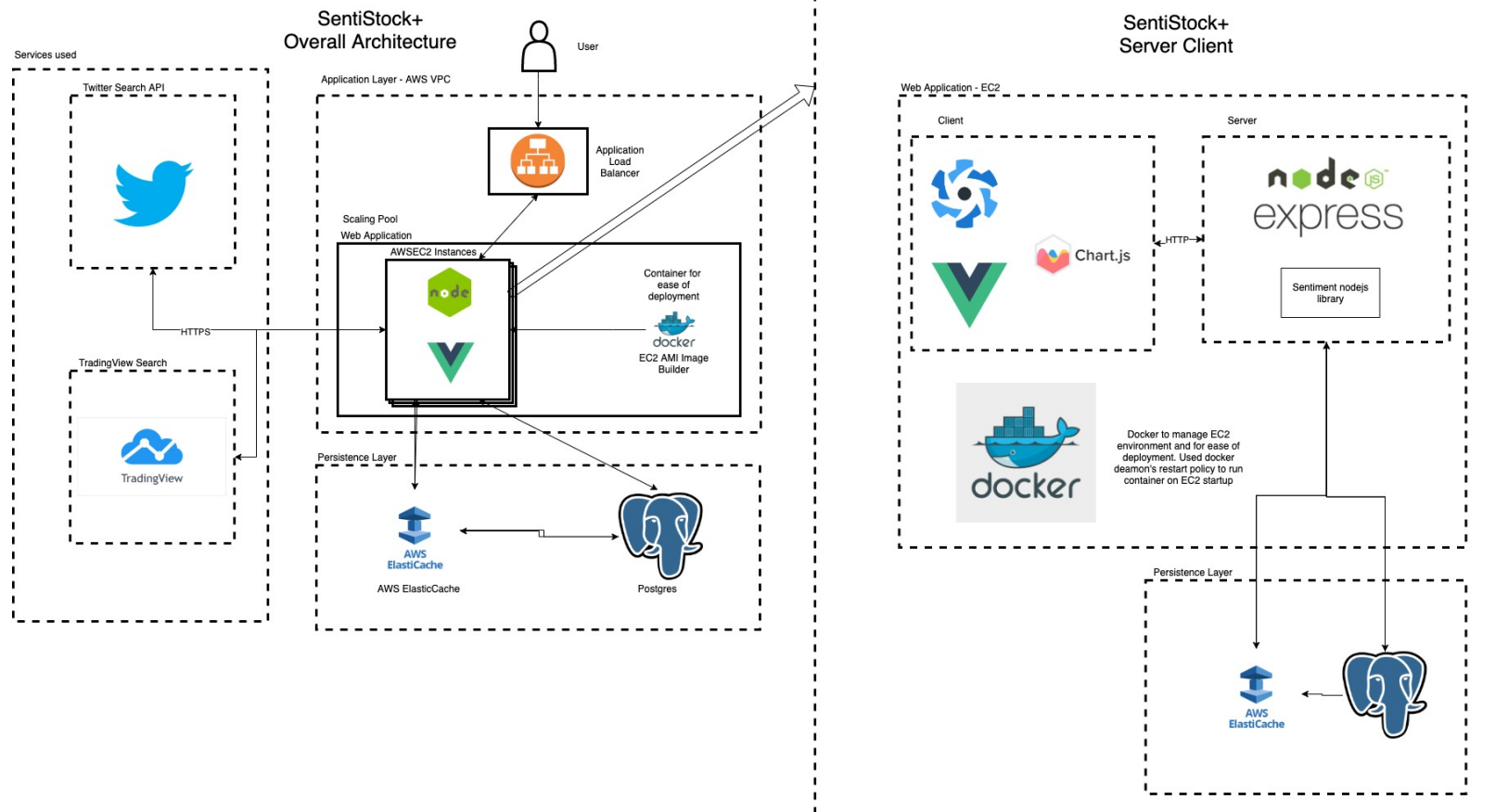
For this project, the server side of the application was written in javascript and used nodejs and express as the runtime and server frameworks respectively. In addition, the server side API endpoints were documented using Swagger. The dependancies on the server side were managed using npm. On the client side, quasar was used as the component framework, which implements vue (client side framework) and vuex (client side state management) along with VueChartJs (which is an implementation of charting library for view). In this project, quasar was used to compile and build the project, and yarn was used to manage dependencies.

For persistence, the application uses postgres as the first level of persistence, and elasticache as a chaching service. Within the postgres schema, there are three tables (tweets, analysis and tickers) which store the related information for the application. The elasticache service stores the current tickers, tweets and historical analysis to provide the user with a fast data access when they refresh the page.

For scaling, this application uses an application load balancer. Each of the instances have an image of the application. This image is created with the help of docker. The docker deamon is used with a restart policy so that whenever the instance loads, the container is run. The scaling policy scales on a CPU usage of 20% or more. This will add an instance from the pool with a maximum number of instances set to 3.

More specific details on the above can be found in the following sections.





## Application

### 3.2 Client side

The user initially hits the SentiStock+ home page, where they are greeted with three columns (Your tickers, Sentiment and Average). When these components load, they fetch the data from the database (or the cache if there is data there), using these endpoints:

- GET /api/ticker/current-tickers → gets the current tickers (displays to your tickers)
- GET /api/tweets/historical-analysis/:ticker → gets the average analysis (displays under Average)
- GET /api/tweets/analysis/:ticker → gets the analysis of all of the tweets associated to that ticker (displays under Sentiment)

Each of these methods check to see if there is data in the cache, and return that, and if not return the database data.

The user can then add a ticker to the list of tickers which they currently already have. They do this by hitting the plus icon under the ‘Your Tickers’ column. This then sends a request to the following endpoint.

- POST /api/ticker/current-ticker → where the body is the ticker being posted

In addition to the above, when the user selects a new ticker to add, they are greeted by a dialog window. In this window they can search for tickers to add to the database. This is using two endpoints on the server.

- GET /api/tradingview/search-ticker and GET /api/tradingview/search-top-tickers

If the user wants to delete a ticker, then can do this by hitting the delete button. This will send a request to the following endpoint.

- DELETE /api/ticker/delete-ticker/:tickerId → where the path parameter is the ticker id

When the user refreshes the charts, it actually sends multiple requests out to the server at once using Promise.all(). What it does is that it gets the current tickers which the user has (in the “Your Tickers “ column), and then fires the requests to the server to the following endpoint.

- POST /api/tweets/analysis → where the body is the ticker being requested for analysis
- POST /api/tweets/historical-analysis → where the body is the ticker being requested for the average analysis

The charts are then updated based on the data which is sent back from the database.

## Application

### 3.3 Server side

The server side technical details can be broken down into API's/routes, util files, middleware, error handling. Where possible, the application tried to follow a 3 layered application architecture, whereby the business logic was sepearted from the persistence layer and routes. Refer to the diagram in the appendix for an overview of how this interacts on a high level. Before moving onto these details, the folder structure for the server is detailed below:



app.js → file which starts the express server. Considered first point of entry for app.

/api → directory which holds all of the API routes

/controllers → contains the business layer of the application

/models → contains the persistence layer of the application

/config → contains configuration for the database and cache services

The index.js file hosts the express application. In addition, it defines the routes for the application, as well as the middleware used within the application.

The client SPA statis assets where served using the server. This was implemented by using a custom middleware which redirected any paths not associated to the API to the static web assets found under ./client/dist/spa.

#### 3.3.1 Route Layer – APIs/Routes

As per the folder structure above, the API's have been segmented to the routes directory. The analysis file holds the endpoints associated with the sentiment analysis of

### Application

the stock ticker, ticker file holds the stock ticker lookup endpoints and the tweets file holds the twitter endpoints. The routes have been segmented into four different sections.

- Admin → provides an endpoint to remove all data from the database and cache
- Ticker → endpoints for dealing with the current tickers associated to the application (the “Your tickers” section in the application)
- Tradingview → endpoints for fetching information from the tradingview API service
- Tweets → endpoints for all of the analysis in the application

The design pattern for the ticker and tweets routes are quite similar, so I am just going to provide 1 example of how data flows within the APIs. I will focus on the ticker file specifically. The ticker file has three endpoints:

- GET /api/current-tickers/
- POST /api/current-tickers
- DELETE /api/current-tickers/:ticker

When the user wants to add a new ticker, they hit the post endpoint. This post endpoint uses the createTicker method provided by the business layer. It then retrieves the stored entries from the database using the getCurrentTickers method provided by the persistence layer. It then creates the cache using the createCurrentTickers provided by the business layer, and then it returns this information as a response to the user. If there are an errors thrown in each of these methods, this is propagated up the request chain, and is handled by the error middleware.

```
/**
 * Store the current tickers for tweets
 */
router.post('/current-ticker', asyncHandler(async function (req, res, next) {
  const { ticker, exchange } = req.body
  // persist into the database
  await Ticker.createTicker({ ticker: ticker, exchange: exchange })
  const currentTickers = await Ticker.getCurrentTickers()
  console.log(currentTickers)
  // store the database entries into the cache
  Cache.createCurrentTickers({ data: currentTickers })
  res.status(200).send({ tickers: currentTickers })
}))
```

If the user reloads, or starts the application, this is using the GET endpoint for the current tickers (to load the “Your Tickers” section). In this endpoint, we are checking to see if the data is within the cache. If so, we serve this, if not then we fetch the data from the database, and then store it in the cache. This endpoint uses the business and persistence layers, specifically the getCurrentTickers method.

## Application

```

/**
 * Get current stored tickers for tweets
 */
router.get('/current-tickers', asyncHandler(async function (req, res, next) {
  // check in the cache
  // if not there, check in the database
  const tickerKey = 'current-tickers'
  return redisClient.get(tickerKey, async function (error, result) {
    if (error) {
      console.log(error)
      return res.status(500)
    }
    if (result) {
      console.log('cache')
      const resultJSON = JSON.parse(result)
      return res.status(200).json(resultJSON)
    } else {
      console.log('db')
      const result = await Ticker.getCurrentTickers()
      // update cache
      Cache.createCurrentTickers({ data: result })
      return res.status(200).json({ tickers: result })
    }
  })
}))

```

Each of the endpoints are wrapped in an `asyncHandler` function as some of the methods used within the endpoints are asynchronous (particularly the data access methods).

The swagger documentation provides a comprehensive overview of all the api endpoints. This can be reached by hitting the endpoint `GET /docs`.

### 3.3.2 Persistence layer

The persistence layer is a representation of what the database looks like in terms of tables. It is broken down into three main files, with another file to manage these tables.

`Db.js` → used to create and drop tables

`analysisDAO` → methods used to access and modify the analysis table

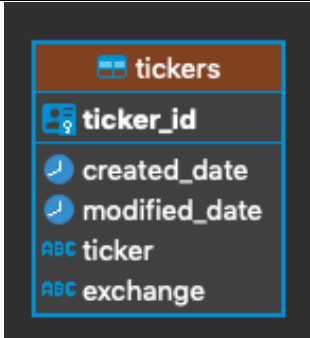
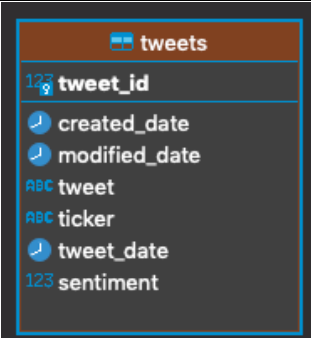
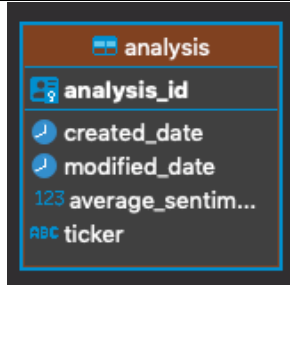
`tickersDAO` → methods used to access and modify the tickers table

`tweetsDAO` → methods used to access and modify the tweets table

In these files we have methods which perform SQL statements on each of the tables. Again, there is a standard structure across all of these files, and I will just focus on one example `tickersDAO.js`. This file has three methods, `getTickers`, `createTicker`, and `deleteTicker`. The `getTickers` method executes an SQL statement to select all the entries from the tickers table. The `createTicker` method executes an SQL statement to create an entry for 1 ticker using the `id`, `created_date`, `modified_date`, `ticker` and `exchange` (exchange being place where the stock is listed). The `deleteTicker` method deletes an entry based on the ticker of the entry.

Of more interest would be the structure of the database, and how the tables look like. For this, refer to the diagram below (obtained by using dBeaver).

## Application

Tickers	Tweets	Analysis
		

There is also the caching service which we need to discuss. In this application, Redis is used as a caching service. Most of the transactions to this service is completed in the business layer. In this service, we are storing three main things.

Historical average (average analysis)

Current tickers

Analysis (on a query param – which is a stock ticker).

The structure for this data looks as follows

Historical average	Current tickers
<pre>{   "historical-average:{ticker-id}": [     {       "created_date": Timestamp,       "modified_date": Timestamp,       "average_sentiment": Long,       "ticker": String     }...   ] }</pre>	<pre>{   "current-tickers": [     {       "created_date": Timestamp,       "modified_date": Timestamp,       "ticker": String,       "exchange": String     }...   ] }</pre>
Analysis	
<pre>{   "query": "ticker-id",   "tweets": [     {       "tweet_id": Long,       "created_date": Timestamp,       "modified_date": Timestamp,       "tweet": String,       "ticker": String,       "tweet_date": Timestamp,       "sentiment": Long     }...   ] }</pre>	

## Application

**3.3.3 Business logic layer**

This layer is where the main computation occurs, such as the sentiment analysis. There are some main files which I want to focus on.

tweetsUtil.js → responsible for combining the tweets together with the sentiment

parserUtil.js → responsible for structure and filtering the data

analysisUtil.js → responsible for providing methods for the analysis

The main things I want to focus on here is the creation of the sentiment, and therefore I might miss out some methods which are also available in other files in this layer. For a more detailed overview of those methods, please refer to the code snippets.

The analysis is created under tweets/Util.js in the createTweetAnalysis method. This method takes in ticker, lang, result\_type and count in the request object. It then uses the twit nodejs library to obtain the tweets from the Twitter search API. Once these tweets are obtained, we then iterate over the selection of these tweets. These tweets are then tokenized and stemmed using the parserUtil.js file (see below for token and stem process).

Input tweet	Output token and stemmed
["The weather is nice"]	['The', 'weather', 'is', 'nice']

This array is then fed into the getSentiment method provided by the analysisUtil.js file. The getSentiment method uses the sentiment nodejs library to provide a numerical value for the sentiment of the text. This is then converted to a percentage, and sent back as a response. After the performance of this analysis, we then filter the data for sentiment which might be invalid (such as 0) and then we store this within the tweet table using the TweetDAO createTweet method. This whole method then returns an array of tweet objects with the sentiment.

**3.4 Dockerizing + serving web application**

The application uses docker to containerize the application. The docker file can be found in the appendix. What the docker file does is that it builds the client SPA, and then copies it to the client folder. It then copies the server to the server folder, and serves the application on port 8000. The static client SPA assets are served via the express application.

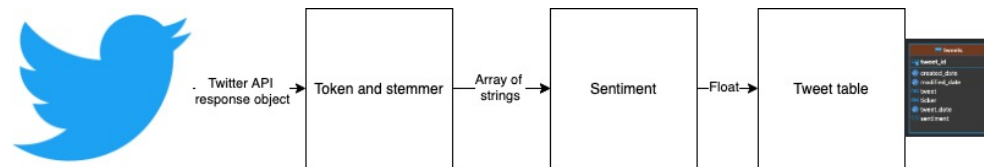
One technical challenge was moving this from a local to a production environment. This was aided with the use of environment variables. These environment variables were used to specify things like the database connection url and the redis connection url, as well as if the application was in the production or development environment. The dotenv nodejs library was used to complete this feature.

## Application

### 3.5 Response filtering/data object correlation

There was some filtering and data manipulation with regards to processing the tweets, and then processing the sentiment from those tweets. In particular, there was some manipulation with obtaining this information from twitter, performing analysis, and then storing this in the persistence services.

The raw data from the twitter API had to be filtered. The only information which was obtained from this API was the created\_date, tweet\_id and the text. The text information was then used in the token and stemmer function to create an array of single strings. This was then used in the analysis file to create the sentiment for the tweet. The final object which was persisted in the database was an object which had the tweet\_id, created\_date, text, ticker, sentiment (calculated) and the created\_date and modified\_date of the entity itself. Please see below for a process flow diagram of how the data changes (note this is just an example).





Raw tweet	Token and stemmer	Output tweet table
<pre>{   "statuses": [     {       "created_at": "Sun Feb 25 18:11:01 +0000 2018",       "id": 967824267948773377,       "id_str": "967824267948773377",       "text": "From pilot to astronaut, Robert H. Lawrence was the first African-American to be selected as an astronaut by any name.",       "truncated": true,       "entities": {         "hashtags": [],         "urls": [           {             "url": "https://t.co/FjPEwnh804",             "expanded_url": "https://www.sprinklr.com",             "display_url": "https://www.sprinklr.com",             "rel": "nofollow"           }         ],         "mentions": [],         "media": []       },       "metadata": {         "source": "&lt;a href='\"https://www.sprinklr.com\"' rel='\"nofollow\"'&gt;Sprinklr"       },       "in_reply_to_status_id": null,       "in_reply_to_status_id_str": null,       "in_reply_to_user_id": null,       "in_reply_to_user_id_str": null,       "in_reply_to_screen_name": null,       "user": {         "name": "Robert H. Lawrence",         "screen_name": "robert_h_lawrence",         "location": "Atlanta, GA",         "description": "African-American astronaut, pilot, and author.",         "url": "https://www.robertlawrence.com",         "profile_image_url": "https://pbs.twimg.com/profile_images/1011111111/robert_h_lawrence.jpg",         "profile_image_url_https": "https://pbs.twimg.com/profile_images/1011111111/robert_h_lawrence.jpg",         "profile_banner_url": "https://pbs.twimg.com/profile_banners/967824267948773377/1511111111",         "followers_count": 1234,         "friends_count": 567,         "retweet_count": 988,         "favorite_count": 3875,         "favorited": false,         "retweeted": false,         "possibly_sensitive": false,         "lang": "en"       }     }   ],   "search_metadata": {     "completed": 1,     "count": 1,     "from": "top",     "max_id": 967824267948773377,     "max_id_str": "967824267948773377",     "next_results": "?max_id=967824267948773376&amp;result_type=recent&amp;tweet_type=recent",     "query": "From pilot to astronaut, Robert H. Lawrence was the first African-American to be selected as an astronaut by any name.",     "refresh_url": "?result_type=recent&amp;tweet_type=recent",     "since_id": null,     "since_id_str": null,     "timestamp_ms": "2018-02-25T18:11:01.000Z"   } }</pre>	<pre>{   "result": [     {       "tokens": [         "From", "pilot", "to", "astronaut",         "Robert", "H", "Lawrence",         "was", "the", "first", "African-American", "to",         "be", "selected", "as", "an", "astronaut", "by", "any"       ]     }   ] }</pre>	<pre>{   "tweet_id": 967824267948773377,   "created_date": "Sun Feb 25 18:11:01 +0000 2018",   "modified_date": "Sun Feb 25 18:11:01 +0000 2018",   "tweet": "From pilot to astronaut, Robert H. Lawrence was the first African-American to be selected as an astronaut by any name. https://t.co/FjPEwnh804",   "ticker": "\$BTC",   "tweet_date": "Sun Feb 25 18:11:01 +0000 2018",   "sentiment": 34.059002093 }</pre>

## Application

### 3.6 Scaling and performance

The application's scaling policy came with regards to the load. The specific metrics which were used to scale the application in and out was the CPU % usage. This metric was used because the application does a lot of resource intensive tasks to analyze the sentiment. The application load mainly comes from the endpoints associated with the analysis of the tweets. As stated previous, the average analysis endpoint is associated with the most amount of load because this endpoint uses the data already within the database to produce the average. The normal analysis endpoint does not do this and just uses the tweets provided by the twitter API which is limited to 100 tweets.

With respect to scaling, a scaling pool was created with a minimum amount of servers being 1 and a maximum number of servers being 3. Each of these servers has an AMI image which was created using a stand alone EC2 instance. On this EC2 instance was the deployed application wrapped in a docker container. This docker container has a restart policy attached to it so that it could restart when the instance fired up. A scaling policy was attached to the scaling group. This scaling policy executes when the CPU usage is above 20%. When this executes, a new instance is added to the scaling pool using the AMI image (Group details – instance count, and scaling policy can be found below)

Group details		Edit
Desired capacity	1	Auto Scaling group name n9960651-sentistock
Minimum capacity	1	Date created Sat Oct 24 2020 10:58:27 GMT+1000 (Australian Eastern Standard Time)
Maximum capacity	3	Amazon Resource Name (ARN) arn:aws:autoscaling:ap-southeast-2:901444280953:autoScalingGroup:27e93c85-0790-4642-942e-a366bad7365a:autoScalingGroupName/n9960651-sentistock

#### n9960651 Scaling Policy

Policy type:  
Target tracking scaling

Enabled or disabled?  
Enabled

Execute policy when:  
As required to maintain Average CPU utilization at 20

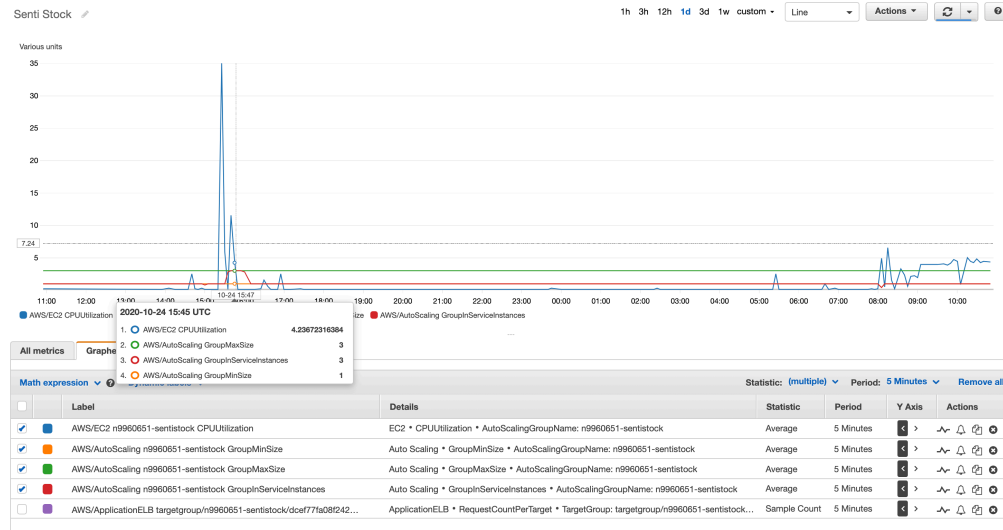
Take the action:  
Add or remove capacity units as required

Instances need:  
15 seconds to warm up before including in metric

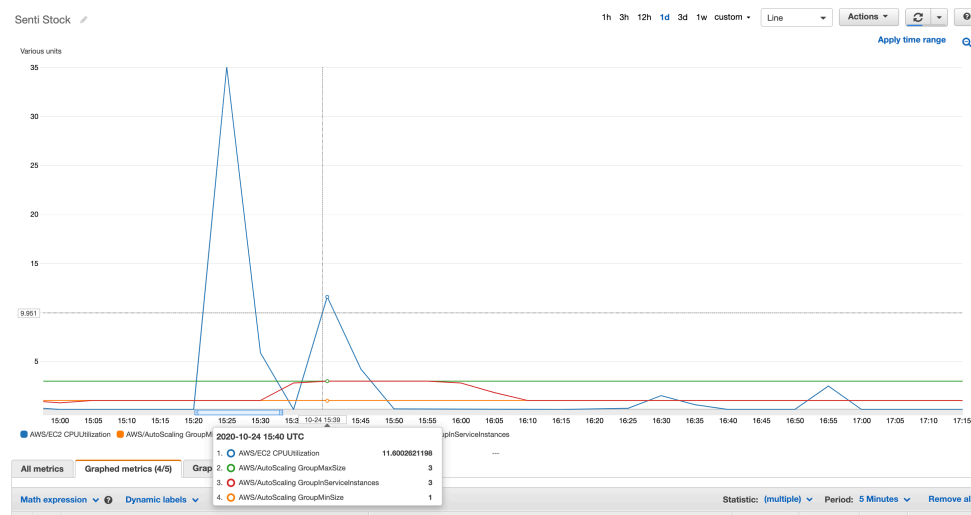
Scale in:  
Enabled

## Application

To test for this, a scenario was created whereby the POST /api/tweets/historical analysis was used to test the load. This endpoint, given a certain ticker in its body, will go out and fetch the new tweets, and then use the already existing tweets in the database to calculate the average sentiment of the ticker. Therefore, the more data in the database for the tweet, the more resources it will take to analyze the average, and hence the increase in the load. In practice, this meant iterating over a list of tickers in a 3 second period to make sure that we were not violating the tweet API rate limits.



As it can be seen above, there is a period in which the application scaled out as the CPU usage exceeded the threshold of a max 35% CPU usage. For a closer look, please see below.



## 4 Test plan

### 4.1 Functional Tests

Test #	Description	Expected	Pass/Fail	Appendix (figure)
1	Searching for the ticker displays list of tickers	AAPL should be displayed	PASS	1
2	You tickers section comes up with tickers you have searched for	AAPL, TSLA, CBA should be displayed under your tickers	PASS	2
3	Clicking update Sentiment chart updates the sentiment	Sentiment for AAPL, TSLA and CBA should be displayed	PASS	3
4	Clicking update Average chart updates the average sentiment	Average Sentiment for AAPL, TSLA and CBA should be displayed	PASS	4
5	Reloading the webpage displays the same data as currently	All charts should have AAPL, TSLA, CBA and current tickers should have AAPL, TSLA and CBA on reload	PASS	5

## 5 Difficulties & Limitations

There were some difficulties accounted within the application development process. The main difficulties came with deploying the application to the cloud environment. Initially I had a fair amount of issues connecting to the elasticsearch service on the cloud. I spent a fair amount of time trying to debug this issue (2 whole days exactly). What I found was that I was not calling the redis client in my code the way I thought I was, and hence it was defaulting to the default redis url and port. This had to do with asynchronicity within the application. I did end up fixing these issues.

Another difficulty I had was actually providing enough load for the application to scale. This was challenging as the sentiment analysis did not consume the amount of resources which I thought it would, it actually consumed less. However I mitigated this issue as I implemented the average sentiment feature which actually usually the historical data stored in the database to provide the average sentiment, and hence has more data to work with.

There is a current bug which is in the application. The bug is if you enter two of the same stock tickers in the 'Your Tickers' section, then the graphs will not update. I did not have time to investigate this any further.

A limitation of the application is the tradingview search API. Because this does not have any specific documentation, the application is up to the mercy of whatever was found when I analysed the network traffic. In addition, because its not documented, tradingview might blacklist the IP address of the server which is trying to connect to the API.

Another limitation to the application is the limits which Twitter places on the standard search API. There would be times where I would be testing the scaling of the application and I would be limited by the rate at which I could contact this API. I would then receive a 500 error because I would not be allowed to contact the Twitter API.

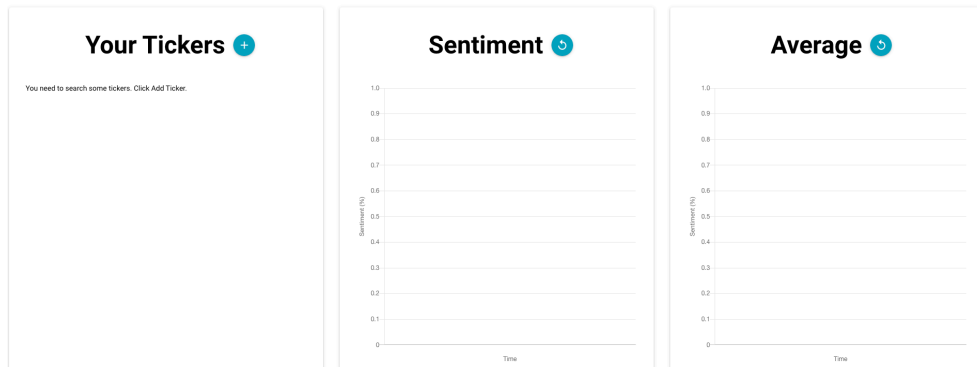
## 6 References

- TradingView. (2020 , na na). *Advanced Real-Time Chart Widget*. Retrieved from TradingView: <https://www.tradingview.com/widget/advanced-chart/>
- Twitter. (2020, na na). *Twitter API v1.1*. Retrieved from Twitter Developer: <https://developer.twitter.com/en/docs/twitter-api/v1>

## Application

## 7 User guide

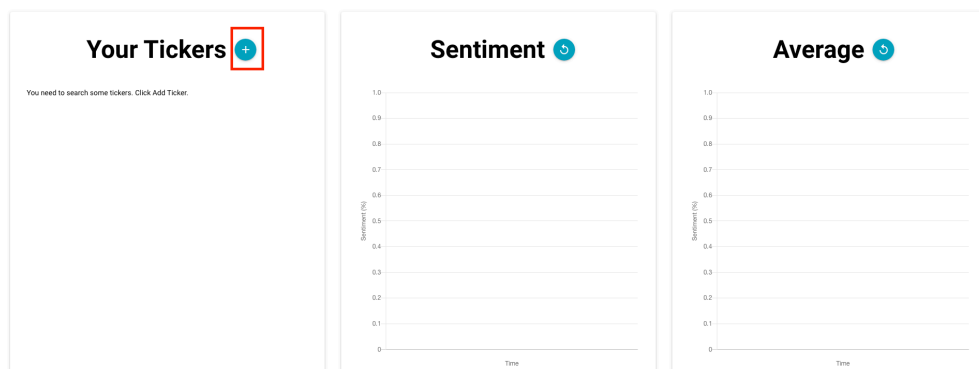
1. User loads the application where they are presented with the following

**SentiStock+**

Designed by Jeremiah Dufourq

v1.0

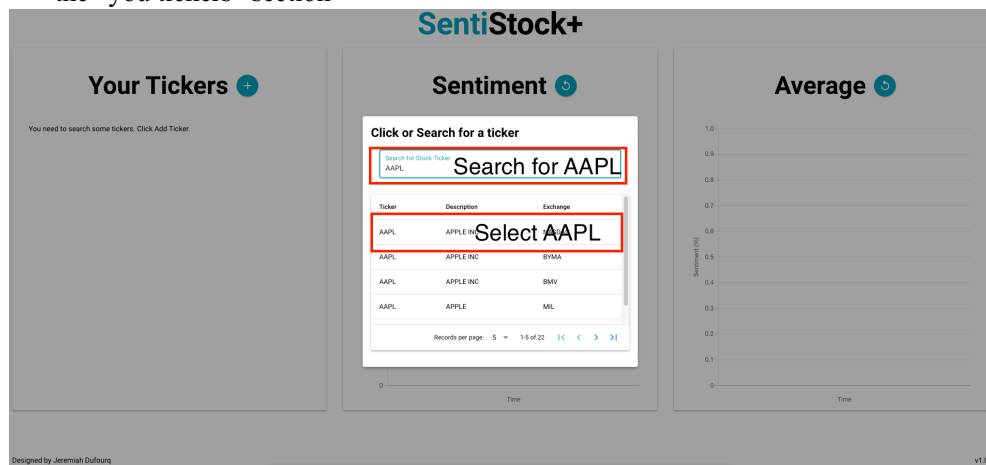
2. They will navigate over to the “Your Tickers” section, and select the plus icon

**SentiStock+**

Designed by Jeremiah Dufourq

v1.0

3. Upon hitting this, they will be prompted with the following dialog. They will search for a ticker and then select this ticker. You should see your tickers in the “you tickers” section



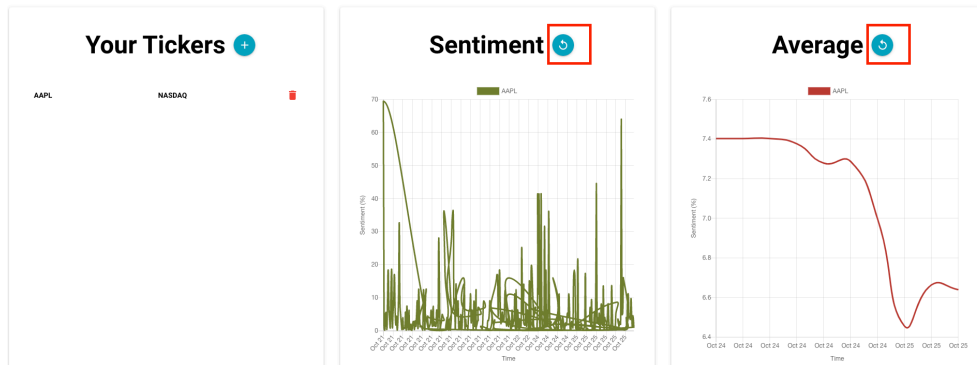
Designed by Jeremiah Dufourq

v1.0

### Application

- Hit the reload button on the sentiment and analysis charts, and this will update

**SentiStock+**



Designed by Jeremiah Dufourq

v1.0

8 Appendix

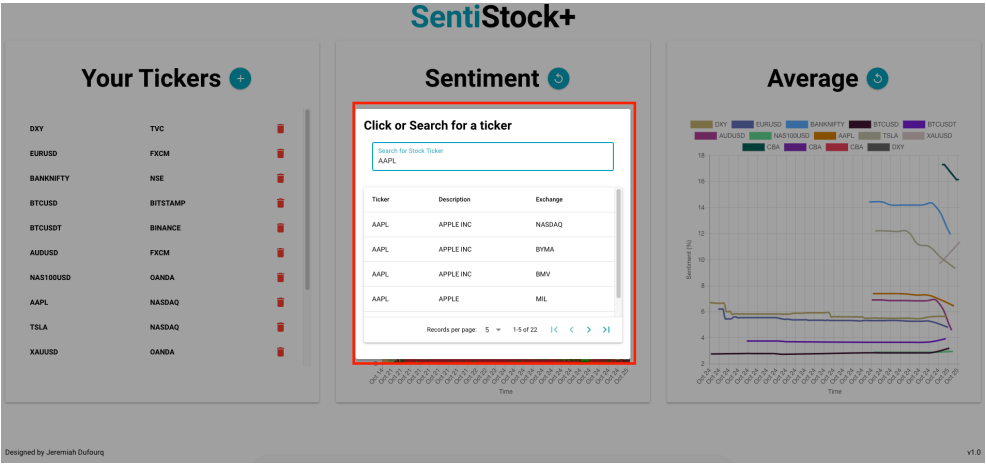


Figure 1

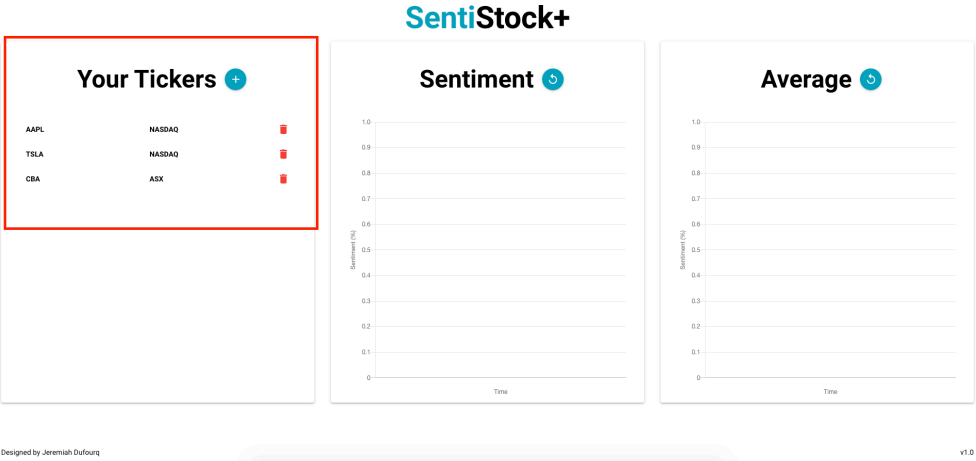
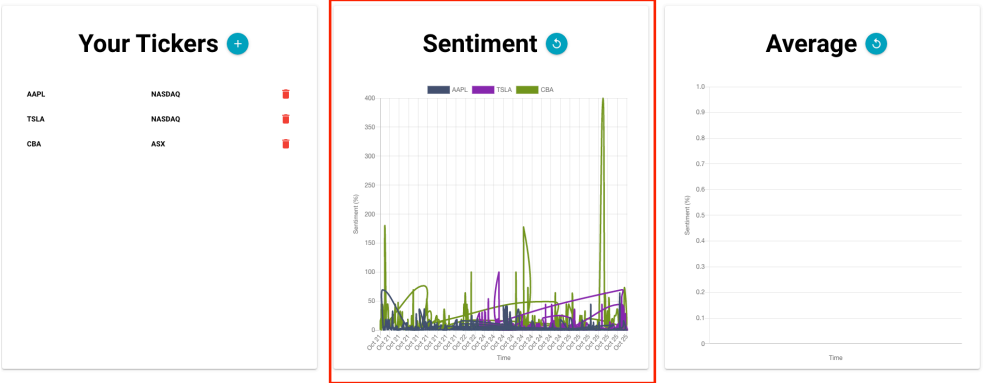


Figure 2



Application  
SentiStock+

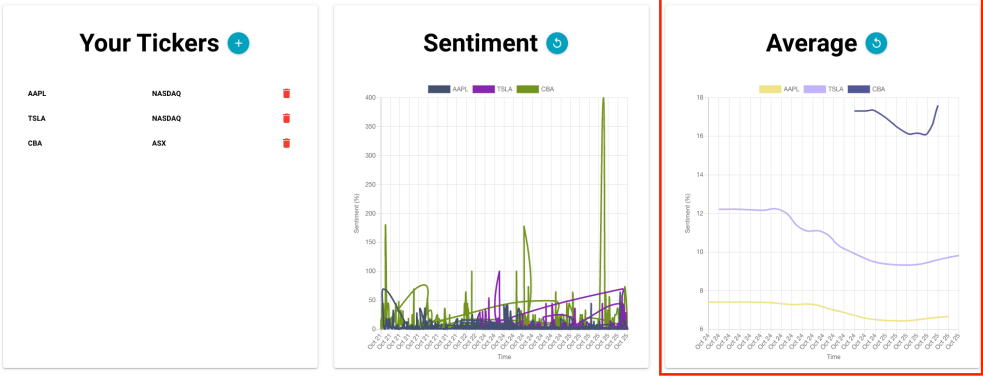


Designed by Jeremiah Dufourq

v1.0

Figure 3

SentiStock+

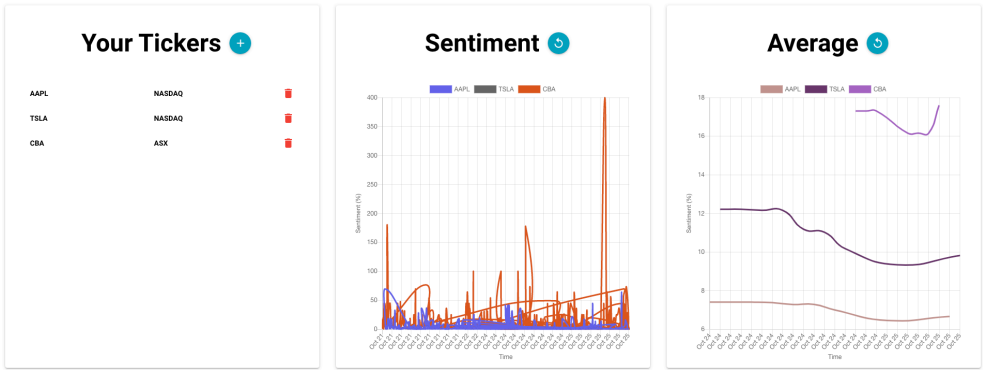


Designed by Jeremiah Dufourq

v1.0

Figure 4

SentiStock+



Designed by Jeremiah Dufourq

v1.0

Figure 5