# QUEENSLAND UNIVERSITY OF TECHNOLOGY

## ASSIGNMENT 1: MASHUP

## CAB432 – Cloud Computing

Jeremiah Dufourq, n9960651
16/09/2020

## Content (ParallelDots, 2020)

# 1 Introduction

## 1.1 Mashup Purpose & Description

The purpose of this project is for the user to find overall feedback on Twitter regarding a certain stock. In addition, the product must provided an intuitive layout for the user, as well as an interactive chart for quick analysis on the platform.

This project has a combination of two API's which are used to provide a mashup service. The first API which I have used is Twitter, a social networking platform, which has been combined with a second API ParallelDots, which provides an AI sentiment analysis product. Both API create a mashup service which allows users to find out the sentiment behind a stock ticker symbol.

For a user to use this service, they will enter the stock ticker symbol of a company which they might be following. The stock ticker symbol is a 3-4 digit code which identifies the company. They will be presented with a table of ticker options. After clicking a stock ticker option, the users will be presented with an analysis of their chosen company. In particular, they will have the sentiment of the stock (position, neutral and negative), as well as a list of tweets to which this sentiment is derived. In addition, there is a chart which is provided alongside the analysis, using the TradingView client widget.

## 1.2 Services used

Within this application, there were three API services which were used, as well as one client widget. The first API service was the Twitter API was used to gather tweet's about the stock tickers. The second service was the ParallelDots API was used to analyse each of the tweets using sentiment analysis. Lastly, the third service was the TradingView ticker search API which was used to validate/search tickers entered by the use. In addition, a TradingView client widget was used to display the price information of the stock in real time. See the links to each, and a brief description below:

### 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 ParallelDots API

The project used the ParallelDots sentiment API. Below is a brief statement about the API:

*ParallelDots AI APIs are the most comprehensive set of document classification and NLP APIs for software developers. Our NLP models are trained on more than a billion documents and provide state-of-the-art accuracy on most common NLP use-cases such as sentiment analysis and emotion detection.* (ParallelDots, 2020) (TradingView, 2020 )

Endpoint → http://apis.paralleldots.com/v4/sentiment

Docs → https://www.paralleldots.com/text-analysis-apis#sentiment

### 1.2.3 TradingView client widget + search API

The project used the Advanced Charting Widget. Below is a brief statement of the widget:

*Advanced Chart Widget is a free and powerful charting solution that easily embeds into any website. Simply adjust the settings and click Apply to see a preview, then copy the embed code and paste it into your site code.* (TradingView, 2020 ) (Twitter, 2020)

Docs → https://www.tradingview.com/widget/advanced-chart/

In addition to the widget above, the project also used the search API provided by TradingView. This search API was found by reverse engineering the tradingview requests on the website, and hence does not have any formal developer documentation. See the endpoint below:

Endpoint → https://symbol-search.tradingview.com/symbol_search/

This endpoint takes a query parameter called text, and hence you can search for symbols like so.

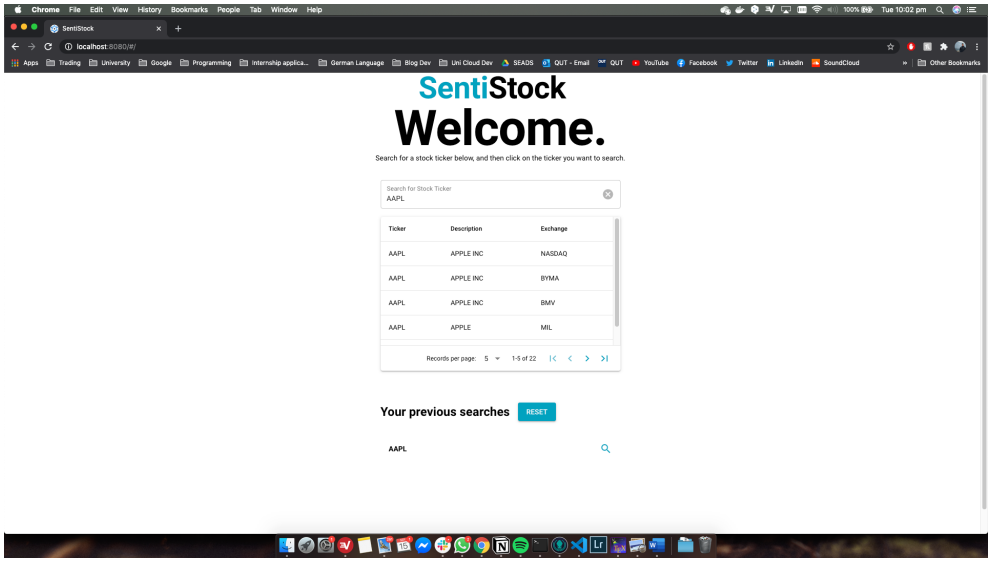
Endpoint → https://symbol-search.tradingview.com/symbol_search/?text=**<stock-symbol>**

# 2 Mashup Use Cases & Services

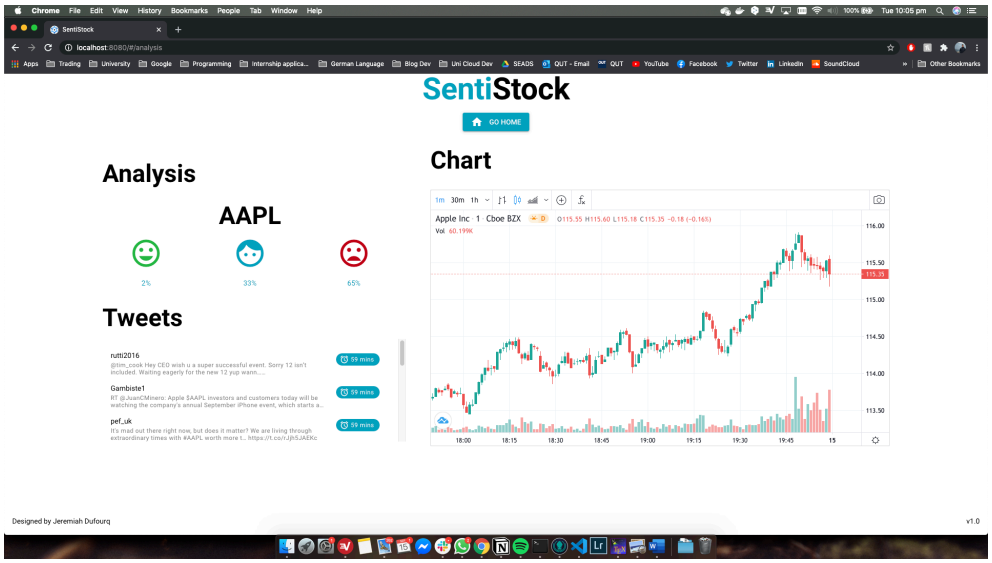
## 2.1 Use case 1

| | |
|---|---|
| As a | Investor |
| I want | The system to display the sentiment of a stock |
| So that | I can guage the sentiment of a stock in a public forum |

This use case specification was met by creating a search box on the home page, as well as a table of a list of available stock tickers. The user can enter a stock ticker into the search field, and then they can search for this ticker. In addition to this, the user can reset what they have entered in the search field to reset their query.




In addition to the input search field, the user is provided a table with the current tickers associated to their search (use case 2). They can select the ticker associated with their search, and this will redirect them to the analysis page. This is a page which displays the analysis results which have been collected from the Twitter and ParallelDots API's. There are three sections to this analysis page. The first is the analysis section which , provides the sentiment for the stock (positive, neutral, negative). The second is the tweets section, which provides a list of tweets which have been used to create the sentiment. And the third is a chart section which provides the chart for the current stock ticker.




On a more technical note, when the user selects the stock ticker on the home page, this stock ticker is sent to the /api/analysis endpoint on the server. This route consumes the stock ticker as a string query, and feeds it to the Twitter api. The data retrieved from

the twitter API is parsed, and then fed to the ParallelDots api. This endpoint then returns the sentiment data in a json object, which is displayed on the analysis page.

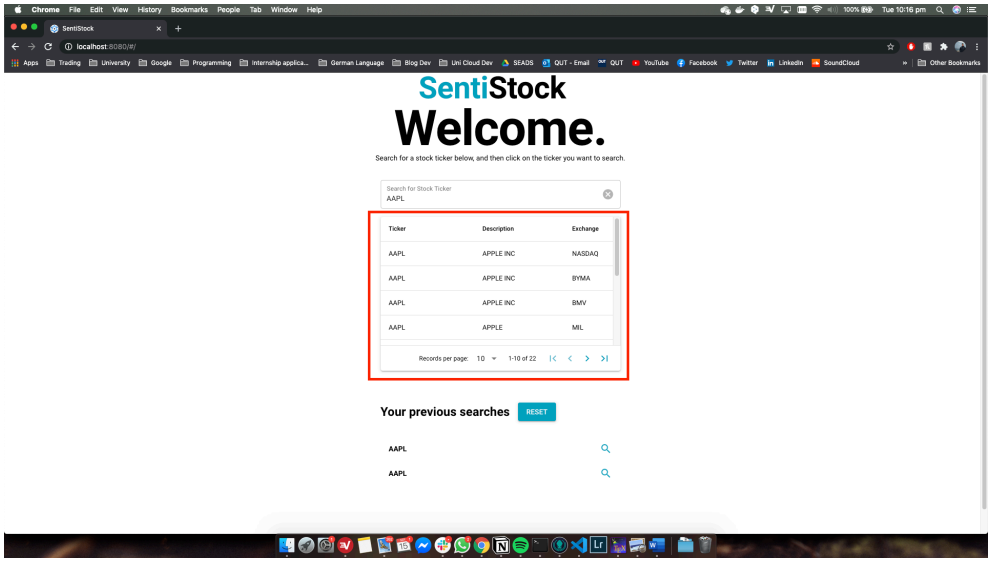
## 2.2 Use case 2

| | |
|---|---|
| As a | Investor |
| I want | The system to search stock tickers |
| So that | I can see the current stock tickers available to me |

This use case specification is in regards to the home page of the application. On the home page, the user is greated with two sections, a search bar with a table, as well as a previous search section. In a sense, this use case is indirectly related to use case 1, however they use two different API endpoints on the server side.

The user has the ability to search a stock ticker in the search bar. When they start typing in the search bar, the table below the search bar is updated with results from a search query to the TradingView search API. As a result, this table lists the current stock tickers which are associated to the users search.

In addition, they can also change the number of items which are displayed, and paginated left and right on the results. In doing this, they will update the current tickers displayed in the table.



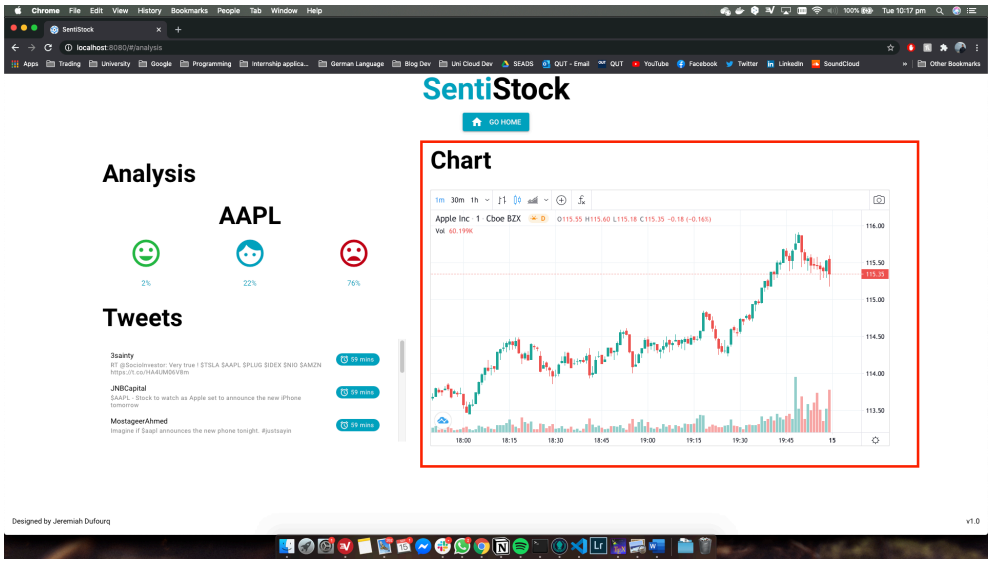

## 2.3 Use case 3

| | |
|---|---|
| As a | Investor |
| I want | The system to display a live chart |
| So that | I can see the current stock price in real time |

This use case is not directly in regards to the server side/mashup of API services, however it does provide some nice functionality for the user. This use case uses the

TradingView client widget to embedded live charts into the application. In addition to this, the client uses Vuex to store the users search history of stock tickers.

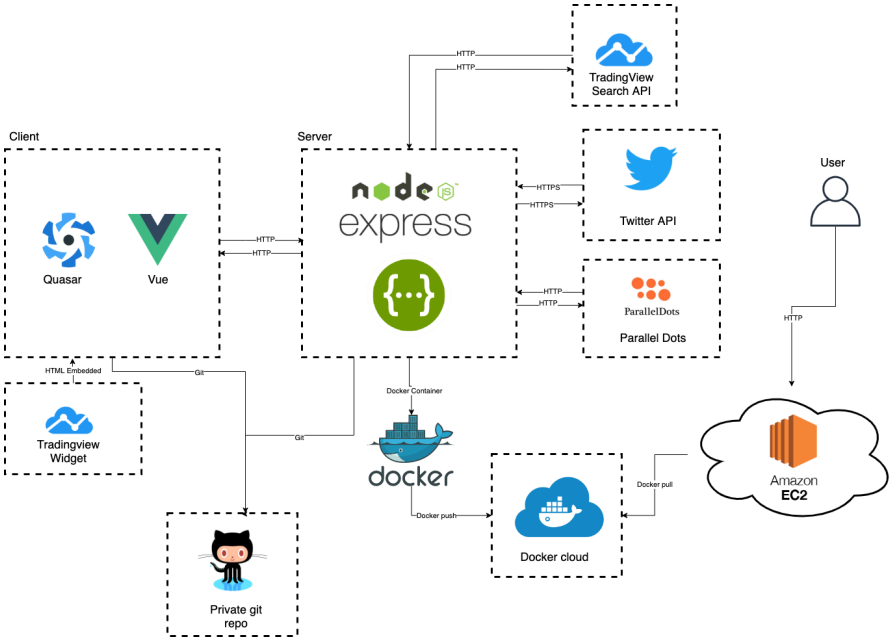
The TradingView widget can be viewed when the user searches for a stock ticker. Specifically, the chart will render when you have hit the 'analysis' page on the client. Because this functionality is implemented on the client side, there are no API endpoints form the server which are used. Hence, this use case is only to improve the users experience.




# 3 Technical Breakdown

## 3.1 Architecture & Data Flow

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). In this project, quasar was used to compile and build the project, and yarn was used to manage dependencies.



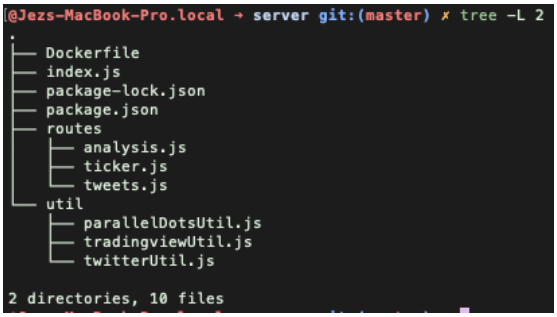

## 3.2 Client side

The user initially hits the SentiStock home page, where they are greeted with a search input field, a table, and a list of their previous searches. When the user searches in the search field, there is a watcher attached to the input field which triggers a method and calls the GET TradingView search api endpoint on the server. This returns the stock ticker data, which is then stored in the table. When the user clicks on a row in the table, it triggers a method which sends a GET request to the analysis api on the server. This then updates the analysis and tweets sections on the analysis page using the data returned from this API. The client handles any errors thrown to it as well via a red popup window.

From a technical perspective, most of the data which is stored on the client side is within a vuex store. The specific data stored is the store ticker, current tweets, stock ticker search history and analysis result. These variables are stored within vuex as they are used across multiple components. The analysis route holds the analysis, tweets and chart components. The home route holds the landing and history components. Each of these components are wrapped in the main layout.

## 3.3 Server side

The server side technical details can be broken down into API's/routes, util files, middleware, error handling. Before moving onto these details, the folder structure for the server is detailed below:




Index.js → file which starts the express server. Considered first point of entry for app.
/Routes → directory which holds all of the API routes.
/Util → directory which holds the util methods used in the API's.

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.

A technical challenge was getting the client SPA static assets served. It was decided that these assets would be served by the express app for ease of deployment. This was done by redirecting any routes which were not the API endpoints to the index.html file, found in the ./client/dist/spa folder.

```
// API routes
app.use("/api", analysisRouter);
app.use("/api", tweetsRouter)
app.use("/api", tickerRouter);
app.use("/docs", swaggerUI.serve, swaggerUI.setup(swaggerDocs));

// Routes which arent associated to API will redirect to static assets for SPA
app.use((req, res) => {
  res.sendFile(path.join(__dirname, '../client/dist/spa', 'index.html'));
})
```

### 3.3.1 API/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 the stock ticker, ticker file holds the stock ticker lookup endpoints and the tweets file holds the twitter endpoints.

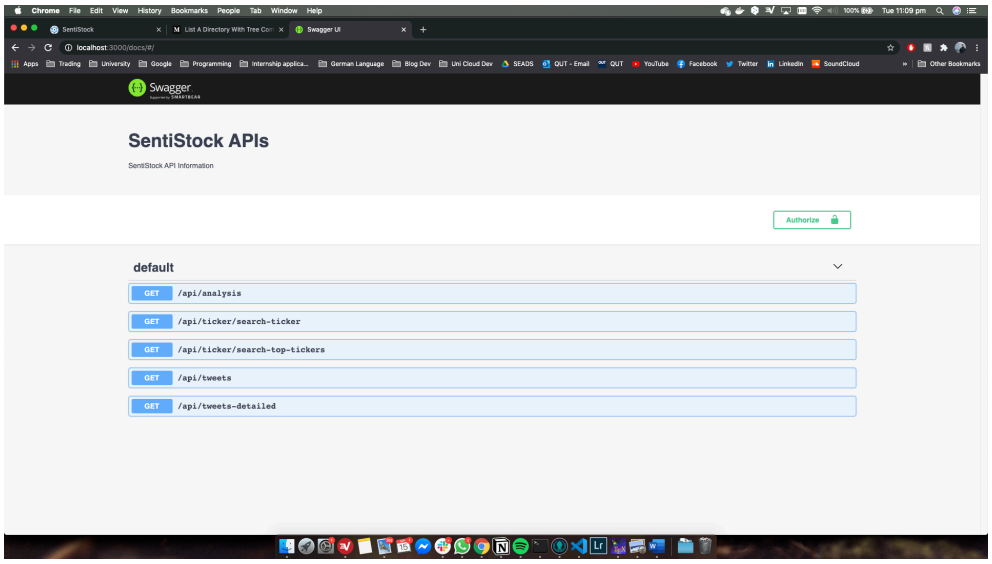
The analysis file holds the endpoints associated with the sentiment analysis of the stock ticker. This can be considered the file which combines the Twitter and ParallelDots API services. It holds one endpoint called GET /api/analysis. This endpoint accepts the stock ticker as a string query, and then feeds this into the getTweetsText method which is within the twitterUtil file. This returns a promise, and is then fed into the getParallelDotsSentiment method within the parallelDotsUtil file. Again, this method returns a promise, which is then sent as a response back from the endpoint. Because both the Twitter and parallelDots util methods are asynchronous, we need to put an await type

infront of the util methods, and then wrap the endpoint function in an asyncHandler function provided by the express-async-handler middleware.

```
// send the data to the twitter API
var twitterResp = await twitterUtil.getTweetsText(params).catch((error) => {console.log(error);});
var sentimentResp =  await parallelDotsUtil.getParallelDotsSentiment(twitterResp).catch((error) => {console.log(error);});
```

```
router.get("/analysis", asyncHandler(async function(req, res, next) {
```

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 Util folders

The util folders are there to provide helper methods for each of the APIs. The parallelDotsUtil file provides getParallelDotsSentiment() which gets the sentiment using the parallelDots sentiment API. This uses a middleware wrapper for the parallelDots API. The tradingviewUtil file provides two helper methods, searchStockTickers() which searches for stock tickers relating to a search query, and searchTopTickers() which searches the top stock tickers provided by TradingView. This uses a direct url path to the GET endpoint *'https://symbol-search.tradingview.com/symbol_search/?text='*. The twitterUtil file provides getTweetsText which returns just the text for the tweets. This is used in the /analysis api route, which feeds the tweets into the parallelDots API. The second method is getTweetsDetailed() which returns a detailed view of the tweets. This is used in the tweets section on the client side.

### 3.3.3 Middleware

The main middleware which I want to focus on are the wrappers used for the Twitter search and ParallelDots APIs. Both of these middleware functions are used to abstract away the API calls into easier to digest methods. Information to the docs are below:

Paralleldots → https://www.npmjs.com/package/paralleldots
Twitter → https://www.npmjs.com/package/twitter
In addition to the above, the application also uses async-handler to handle asynchronous functions for the api's, and swagger, to document the apis.

## 3.4 Deployment & Use of Docker

### 3.4.1 Docker

The docker file collects node as the base image for the container. The docker file pulls the latest node image from the official Docker Hub repository, which matches the version targeted during development and production. In this section, the docker file copy's the current code from the client folder in the codebase. It then sets the current working directory of the docker container to this /client folder. This /client folder holds all of the code for the frontend (client) of our application. After loading the codebase into the client folder, the docker file installs the dependencies for the client. In this case, we are using yarn, which is similar to npm and is what vue uses to manage dependencies. In addition to installing the dependicies, we also install quasar which is a vue component framework based on the material UI specification. After this, we run 'quasar build' which will compile our SPA (single page application) into a /dist/spa folder which we will serve using the express app. The server section of the dockerfile is somewhat similar to the client. In this section we copy the folders from the /server folder to the same /server folder in the container. We then update the working directory to this folder. After this, we install the dependicies for the nodejs application using npm. We then expose port 3000 on the container, which we will use to connect to the express application. The command 'npm start' is used to execute a script command which is within the package.json folder.

### 3.4.2 Deployment

The application is deployed to an aws EC2 instance by pulling an image from a docker repository. Deployment follows the following process:

1. The image is created on the local machine by running the following command in the root directory.
   *Docker build -t <image-tag> .*
2. From here this image is pushed to the remote docker repository using,
   *Docker push <docker-username>/<image-tag>*
3. Then the image can be pulled into the aws EC2 instance using,
   *Docker pull <docker-username>/<image-tag>*
4. And then it is run using (we also search for current images before running using docker images)
   *Docker run -it -p 3000:3000 <image-id*

# 4 Test plan

## 4.1 Functional Tests

| Test # | Description | Expected | Pass/Fail | Appendix (figure) |
|---|---|---|---|---|
| 1 | Searching ticker displays correct data in the table on home page | AAPL should be displayed | PASS | 1 |
| 2 | Previous searches section comes up with previous searches on home page | AAPL, TSLA, CBA should be displayed under previous section | PASS | 3 |
| 3 | Clicking the ticker in the table on the home page routes to the analysis page | AAPL displayed on analysis page after clicking AAPL | PASS | 2 |
| 4 | Analysis data is displayed correctly (positive - green, neutral - blue, negative - red) | Client displays data in request for AAPL | PASS | 2 |
| 5 | Chart displays the correct ticker/company | Chart should display AAPL | PASS | 2 |

## 4.2 Error tests

| Test # | Description | Expected | Pass/Fail | Appendix (figure) |
|---|---|---|---|---|
| 1 | Entering the incorrect ticker returns an error | 400 error message - You have entered an invalid stock ticker. | PASS | 4 |
| 2 | Entering no ticker produces an error | 400 error message - Required query params missing. You must enter a stock ticker. | PASS | 5 |

## 4.3 Api tests

| Test # | Description | Expected | Pass/Fail | Appendix |
|---|---|---|---|---|
| 1 | Analysis API returns sentiment, correctly displayed on frontend | Sentiment json object with positive, negative and netural | PASS | 6 & 7 |
| 2 | Tweets-detailed API returns tweets with ticker (10 tweets all in English) | AAPL tweets array of length 10 (shows 9 because of indexing) | PASS | 8 |
| 3 | Ticker-search returns a list of tickers | List of tickers objects (json objects) relating to AAPL | PASS | 9 |

# 5 Difficulties & Limitations

The main issue encounted with this project is the limitations of the ParallelDots API. The ParallelDots API gives you a daily quota of 1000 hits on the free tier. In addition to this, when more then 600 characters are passed to the API, the hits are counted at a higher rate, and therefore you can use up your daily quota faster. Therefore, when I was developing, I had to decrease the amount of tweets/text I was sending to the ParallelDots API. This was done by reducing the number of tweets I was getting from the twitter API by setting the count query to a much smaller number.

## 5.1 Statement on Assignment Demo

Demoeing **FACE TO FACE**. I have added myself to the 4-5pm availability on Monday 21st .

# 6 References

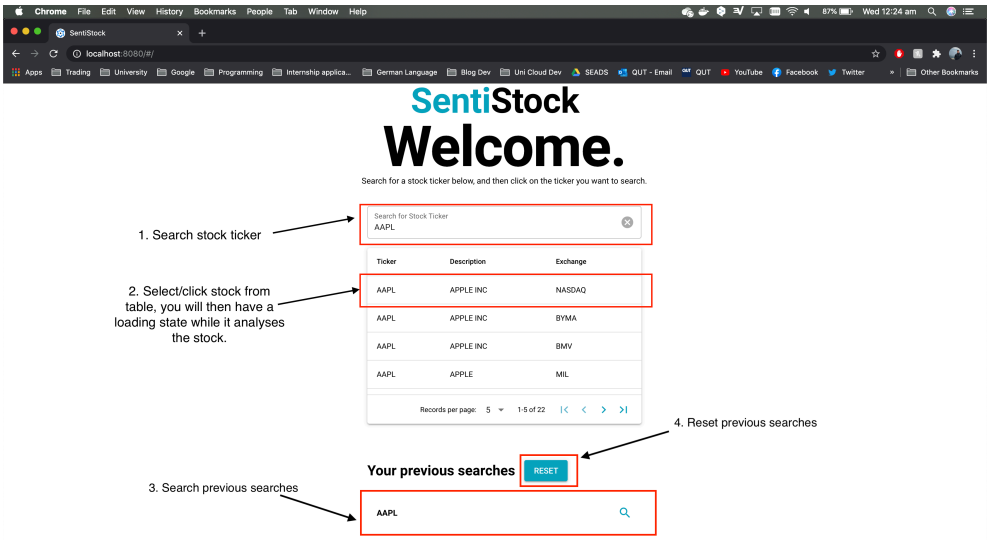
ParallelDots. (2020, na na). *Sentiment Analysis ParallelDots*. Retrieved from ParallelDots: https://www.paralleldots.com/sentiment-analysis

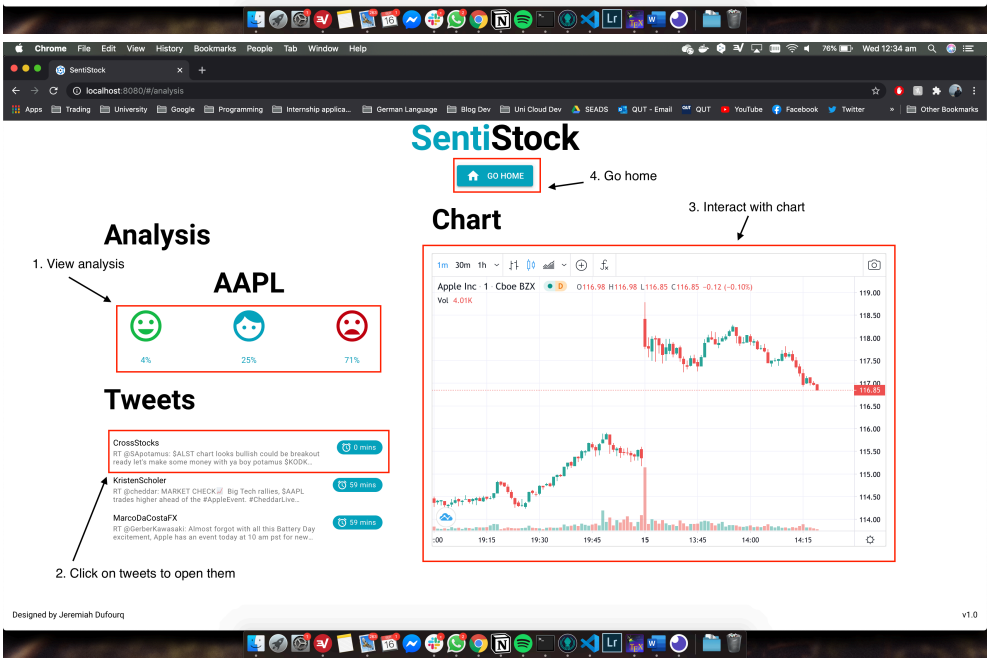
TradingView. (2020 , na na). *Advanced Real-Time Chart Widget*. Retrieved from TradingView: https://www.tradingview.com/widget/advanced-chart/

Twitter. (2020, na na). *Twitter API v1.1*. Retrieved from Twitter Developer: https://developer.twitter.com/en/docs/twitter-api/v1

# 7 User guide

The application can be used doing the following steps as per the screenshots.



# 8 Appendix

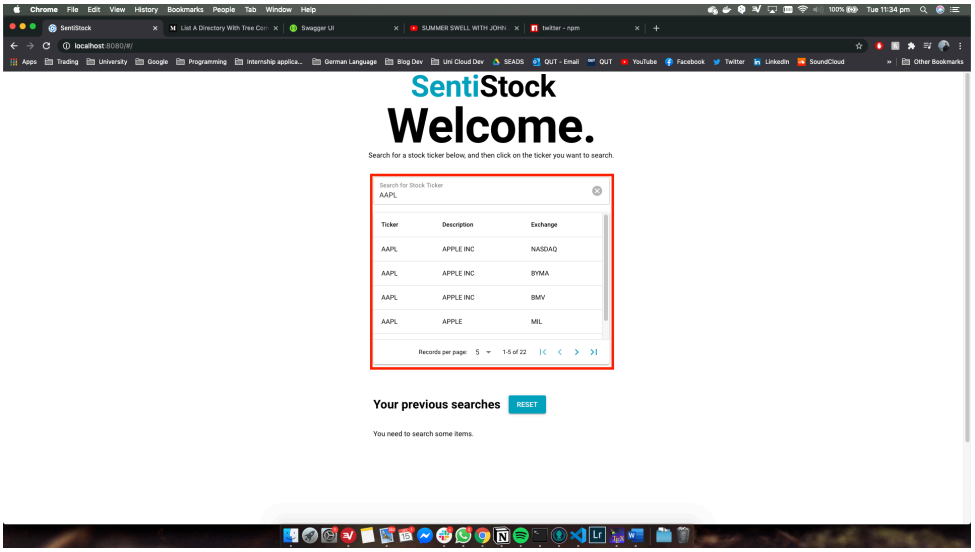



Figure 1

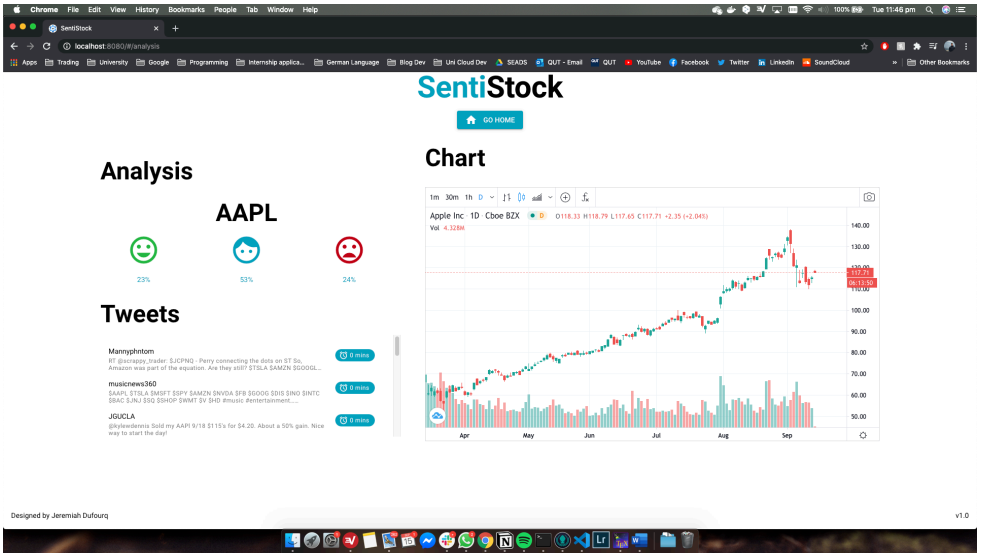



Figure 2

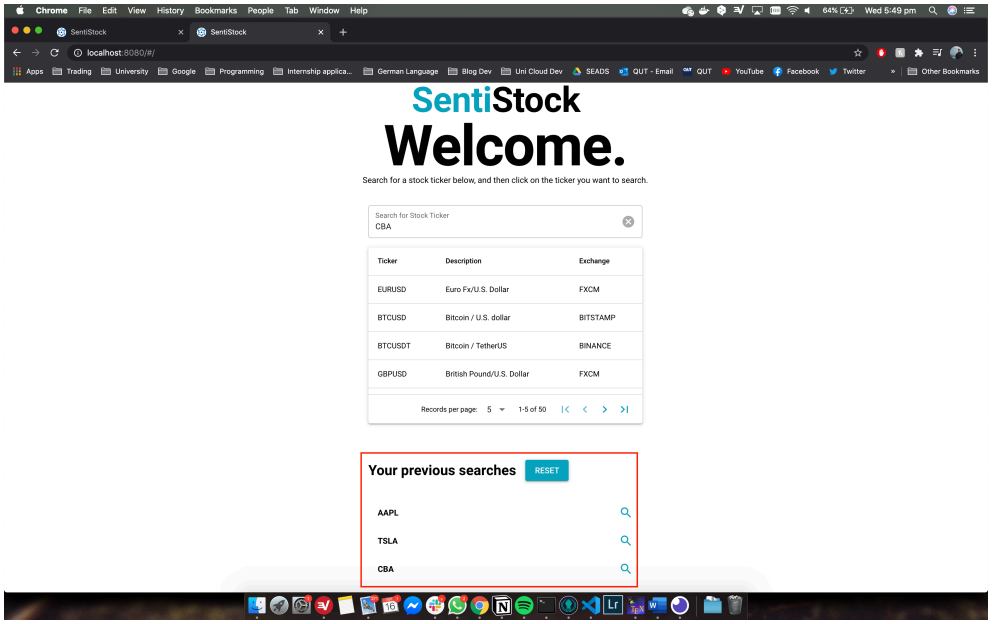



Figure 3

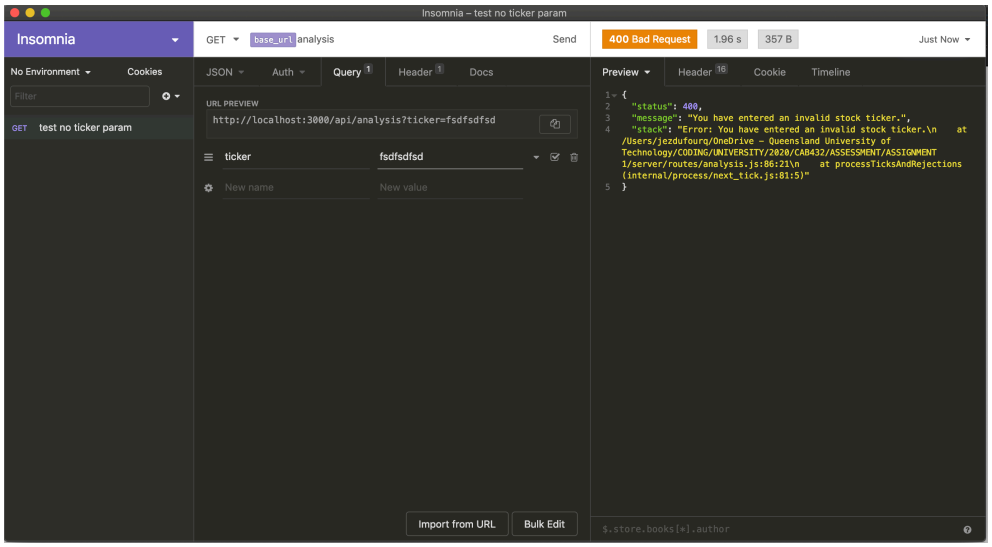



Figure 4

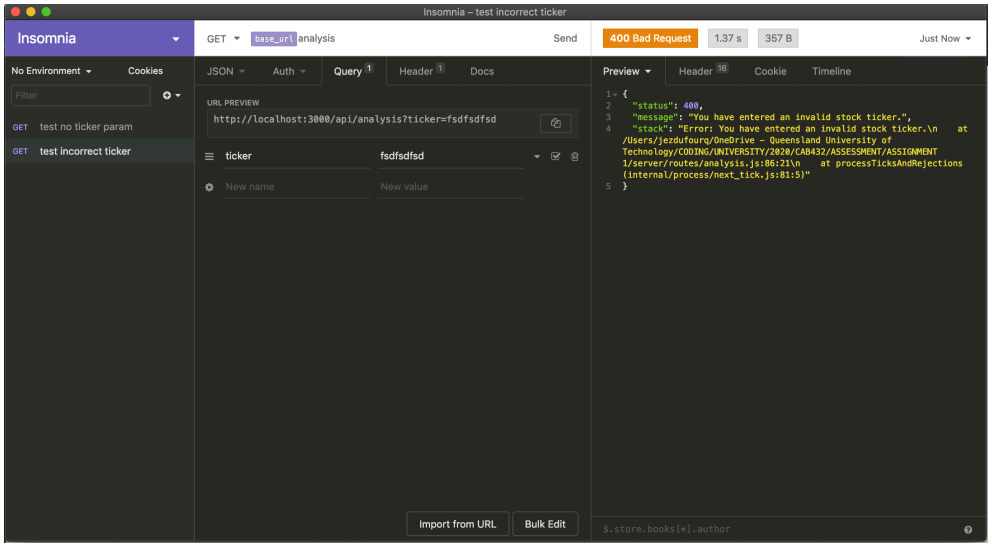



Figure 5

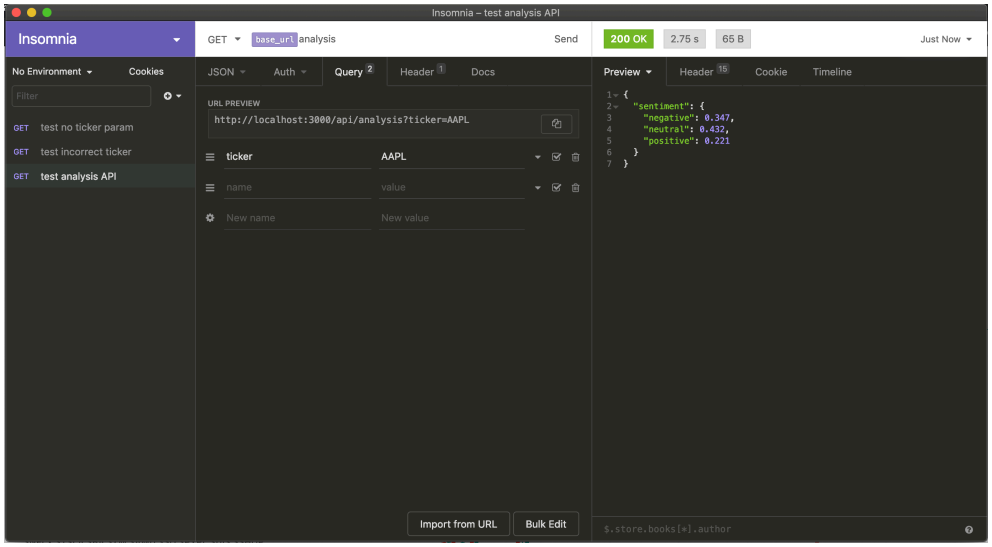



Figure 6

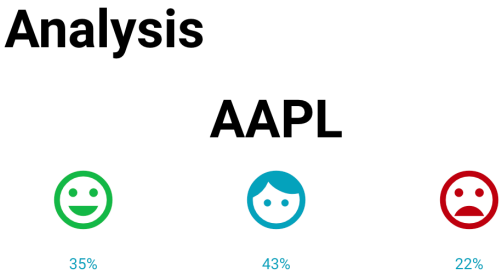



Figure 7

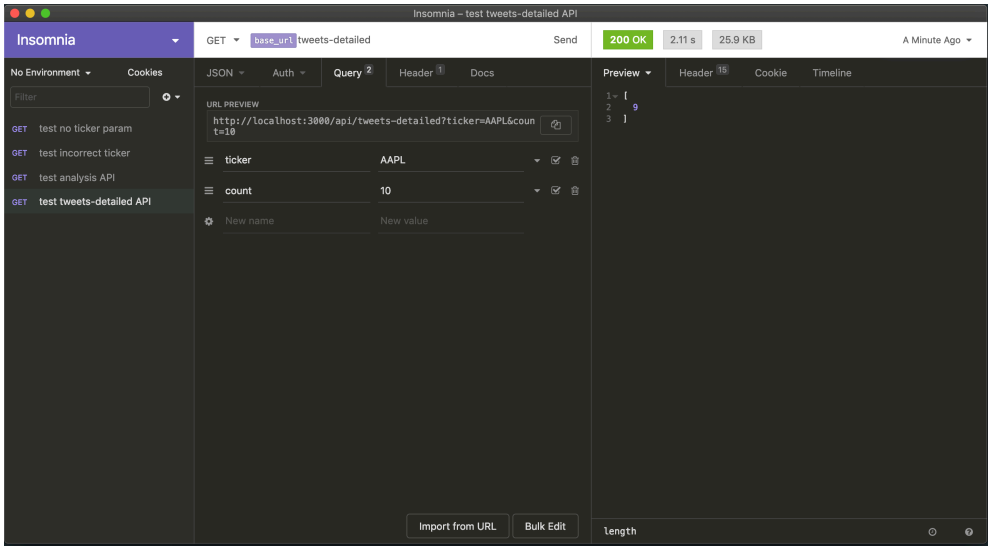



Figure 8

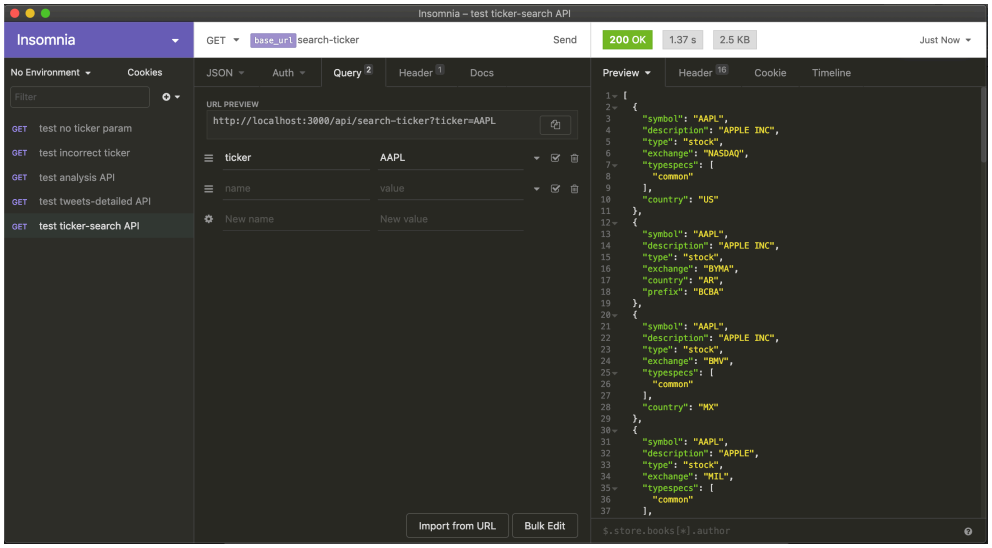



Figure 9

Dockerfile

UNIVERSITY > 2020 > CAB432 > ASSESSMENT > ASSIGNMENT 1 > Dockerfile > ...

```
# Downloading latest node version
FROM node:latest

# Client files
COPY ./client /client
WORKDIR /client
# Installing client dependicies
RUN yarn
RUN yarn global add @quasar/cli
# Building quasar SPA
RUN quasar build

# Server files
COPY ./server /server
WORKDIR /server
# Installing server dependicies
RUN npm i
# Exposing port and running express application
EXPOSE 3000
CMD ["npm", "start"]
```

Figure 10: Dockerfile