UNIVERSITY OF MILAN

PROJECT REPORT

WEB & MOBILE PROGRAMMING

Twitter Automa

Author: Andrei CIULPAN

Badge Number: 872394

Academic Year: 2018-2019

APP website: `https://unimitwitterbot.herokuapp.com`

Exam session of October 29, 2018

Contents

1 Introduction

This is a project based on the Twitter APIs for the Web Programming course.
The purpose of this project is to make an app using programming and markup languages designed for web technology such as HTML/CSS/JavaScript.
This app automatically tweets out random hashtags that are in the current global trending list and provides, for each tweet, some additional information like the number of retweets, favorites, tweet ID and date of creation (this information is updated automatically via a webhook connection with Twitter). This app was developed locally and then deployed to a cloud platform as a service (PaaS) called Heroku which supports several programming languages like NodeJS, an open-source, cross-platform JavaScript run-time environment that executes code outside of a browser, which I used to run server-side scripts on the cloud platform.

1.1 Project Analysis

The following sub-sections will describe some aspects of this application such as users (who the app is meant for), business model and data flow. The technological aspects will be shown in section 2.

1.1.1 Users

Technical capabilities and possibilities

This app is designed for any user who knows how to use Twitter and a browser on a very basic level; the user should be capable of understanding what a tweet or a trending hashtag is and how to access a website through a browser. Anyone can connect to the website through a browser, therefore the only limitation of this app is that the device used by the user must be able to connect to the internet and display HTML content (for example through a browser). The information is displayed entirely in text, which means that the users won't use a lot of bandwidth to access the website.

Languages

The website is displayed in the english language, therefore the user must understand basic english in order to be able to understand the contents.

Motivation

The app's only purpose is to provide free information on an entertainment level for users interested in the global trending hashtags on Twitter.

1.1.2 Business Model

The business model of this app is completely free: there will be no ads or content locked behind a subscription fee therefore it has no real value except for it being a source of information for interested users.

1.1.3 Data Flow

Obtaining the data

The data displayed on the website is automatically updated via a webhook connection with Twitter. What this means is that every time something happens on the bot's Twitter account, for example when one of its tweets gets a favorite (like) the information on the website is updated through a method called getTweets that will be explained in a later section.
Note that in the picture below (Figure 1) it took roughly 1.2 ms to get the tweets that the bot made so far from its own Twitter account and update the list which is then passed to the client where it is formatted and displayed in a human readable manner. It can be concluded that the time it takes to update the list is very insignificant and the user will not even notice it.

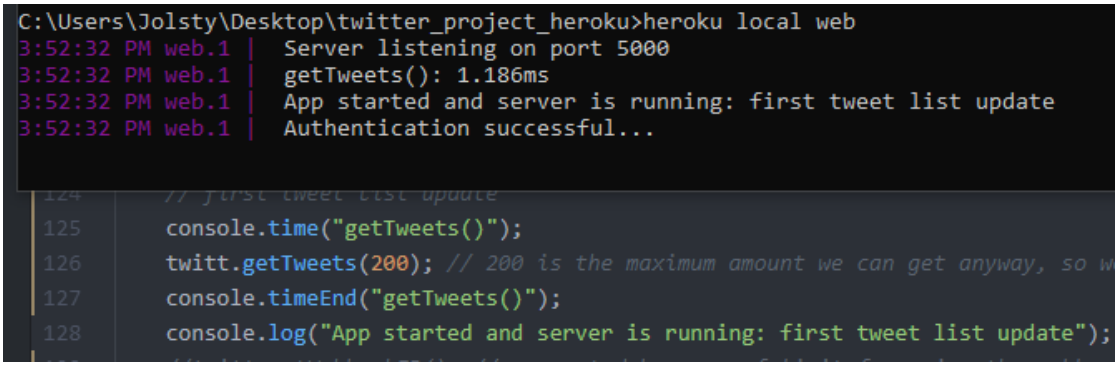

Figure 1: Time to get tweet list from Twitter by using the APIs.

Archiving the data

All the data shown on the website is received from the Twitter APIs by sending a GET request to statuses/user_timeline. Each call to this URL returns a JSON object (Figure 2) that represents metadata for multiple tweets from the caller's account and that we can handle with JavaScript.


```
[
  {
    "created_at" : "Thu Apr 06 15:28:43 +0000 2017" ,
    "id" : 850007368138018817 ,
    "id_str" : "850007368138018817" ,
    "text" :
"RT @TwitterDev: 1/ Today we're sharing our vision for the future of the Twitter API platfor
m!
n https://t.co/XweGngmxlP" ,
    "truncated" : false ,
    "entities" : {
      "hashtags" : [],
      "symbols" : [],
      "user_mentions" : [
        {
          "screen_name" : "TwitterDev" ,
          "name" : "TwitterDev" ,
          "id" : 2244994945 ,
          "id_str" : "2244994945" ,
          "indices" : [
            3 ,
```


Figure 2: Data returned as a JSON object.

This JSON object is rather large, and there's no need to use every single piece of information inside of it. The piece of code in Figure 3 makes sure to select only the necessary information like the tweet ID, number of retweets and so on.

```
// Filters the tweet data in the argument and returns an array with only the necessary information
function filterTweets(data) {
    var tweetsFiltered = [];
    for (var i = 0; i < data.length; i++) {
        tweetsFiltered.push(
            {
                text: data[i].text,
                stats: {
                    created_at: filterDate(data[i].created_at),
                    id_str: data[i].id_str,
                    retweets: data[i].retweet_count,
                    favorites: data[i].favorite_count,
                }
            }
        );
    }
    return tweetsFiltered;
}
```

Figure 3: Selection of only the necessary information.

At the end of the day we are left with an array that contains selected information for multiple tweets. All of this is done server-side, but now this data must be sent to the client in order to show it on the website.

Showing the information on the website

The data from the server is sent to the client by using a HTML div with the hidden display property as in Figure 4. Normally there would be security issues by doing it this way but this data is not sensitive so it's not important. This is done thanks to the Express and EJS modules but they will be explained later on.

```
<div id="statistics" style="display: none;"><%= tweetStatisticsString %></div>
```

Figure 4: Passing data from server to client.

Then we get this data in a client-side script and add some HTML rules so it's

interpreted nicely by the browser (Figure 5). You can see in the first line of code that we get the hidden div shown before.

```
function getStatsHTML(hashtag) {
    var statistics = JSON.parse($('#statistics')[0].innerHTML); //JSON.parse changes a json string into an object
    var hashtagStatsHTML;
    for (var i = 0; i < statistics.length; i++) {
        if (hashtag === statistics[i].text) {
            hashtagStatsHTML = "<br>\
                                <div class='w3-leftbar w3-border-purple'>\
                                    <div class='w3-margin-left fontLighter'>\
                                        <strong>Date</strong>: " + statistics[i].stats.created_at + "<br>\
                                        <strong>Tweet ID</strong>: " + statistics[i].stats.id_str + "<br>\
                                        <strong>Retweets</strong>: " + statistics[i].stats.retweets + "<br>\
                                        <strong>Favorites</strong>: " + statistics[i].stats.favorites +
                                    "</div>\
                                </div>\
                                <br>\
                                <div class='w3-center'>\
                                    <a class='w3-btn w3-purple w3-ripple w3-round-large' href=http://www.twitter.com/search?q=" +
                                    <i class='fa fa-external-link fa-lg fontLighter' aria-hidden='true'></i> Link to trend</a>\
                                </div>"; //
        }
    }
    return hashtagStatsHTML;
}
```

Figure 5: Adding some HTML rules to our data.

This is done everytime a button that shows additional information is clicked. In Figure 6 you can see how it has a small impact on performance (roughly 0.4 ms).

```
getStatsHTML(): 0.421142578125ms                                    stats.js:11
getStatsHTML(): 0.460205078125ms                                    stats.js:11
getStatsHTML(): 0.43701171875ms                                     stats.js:11
getStatsHTML(): 0.42431640625ms                                     stats.js:11
getStatsHTML(): 0.427001953125ms                                    stats.js:11
getStatsHTML(): 0.398193359375ms                                    stats.js:11
getStatsHTML(): 0.401123046875ms                                    stats.js:11
>
console.time("getStatsHTML()");
container.append("<div class='statsDiv'>" + getStatsHTML(hashtag) + "</div>");
console.timeEnd("getStatsHTML()");
```

Figure 6: Time to get tweet list from Twitter by using the APIs.

2 Technological Aspects

2.1 MVC

The app follows an MVC pattern. The Model-View-Controller is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application.
The Model component corresponds to all the data-related logic that the user works with. In this case the model represents the data in the form of an array of JSON objects (usually the model is the database, but we don't use that in this project) that is being transferred between the view and the controller, therefore it is the object that we pass from server to client.
The View component is used for all the UI logic of the application. In this case it represents how our information is displayed on the website. It's basically our HTML/CSS code. The app also uses a template engine called EJS. EJS is a simple templating language that lets you generate HTML markup with plain JavaScript. The way it was used in this app can be seen in Figure 7. We have to set EJS as the view engine for our Express application using app.set('view engine', 'ejs');. Notice how we send a view to the user by using res.render(). It is important to note that res.render() will look in a views folder for the view. So we only have to define "index" since the full path is views/index.ejs.

```
app.get('/', (req, res) => {
    res.render('index',
        {
            title: "Trending Hashtags Bot",
            tweetStatistics: twitt.myTweets,
            tweetStatisticsString: JSON.stringify(twitt.myTweets),
            tweetCount: twitt.tweetCount,
            updatedTimeMS: twitt.updateTime
        }
    );
});
// View Engine Middleware
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));
```

```
</li><br>
<% tweetStatistics.forEach(function(tweet){ %>
<li class="w3-block w3-btn w3-animate-opacity w3-white w3-hover-white fontBold hashtag"><%= tweet.text %></li><br>
<% }) %>
```

Figure 7: EJS

The Controller accepts user input (for example visiting a website, clicking on a button or submitting a form) and converts it to commands for the model or view. In this case it can be what the user can do with it like, for example, on the website there's a button that the user can click on to show more information for a tweet as you can see in Figure 8.

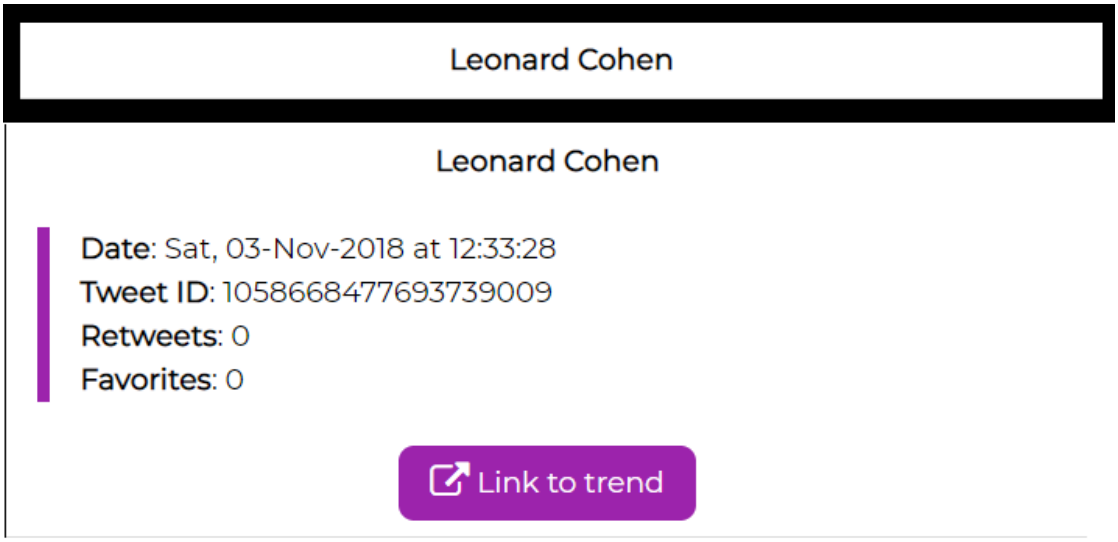

Figure 8: Example of user input.

2.2 Used technologies

This app uses all the technologies required by the professor, such as HTML5, CSS, JavaScript, AJAX, NodeJS and a cloud platform called Heroku. We will also go more in-depth into the Twitter APIs later.

2.2.1 HTML5 [1]

All pages are developed in valid HTML5 and they use the HTML5 APIs like, for example, the DOM which can be easily modified by JavaScript code (see Figure 9 for an example).

```
if (infoDiv) {
    infoDiv.click(function () {
        if (firstClick) {
            var container = $(this);
            var hashtag = container[0].innerHTML;
            console.time("getStatsHTML()");
            container.append("<div class='statsDiv'>" + getStatsHTML(hashtag) + "</div>");
            console.timeEnd("getStatsHTML()");
            firstClick = false;
        } else {
            $(".statsDiv").remove();
            firstClick = true;
        }
    });
}
```

Figure 9: DOM manipulation using jQuery. When the button is clicked by the user, $(this) takes it and appends another div to it so that it can expand.

2.2.2 CSS [2]

The presentation of the documents is styled by using CSS files embedded into the HTML files (see Figure 10 for an example). We can see that there are some third-party CSS files like fontawesome (which is used for some icons like the HOME button), w3.css and font.css. There's also a file called custom.css which I created. Note that I didn't write all of my HTML5 & CSS code because I used a free to use template from W3.CSS called Dark Portfolio which can be found here: `https://www.w3schools.com/w3css/tryw3css_templates_dark_portfolio.htm`.

```
<link rel="stylesheet" href="css/w3.css">
<link rel="stylesheet" href="css/font.css">
<link rel="stylesheet" href="/css/custom.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
```

Figure 10: Embedding CSS into HTML.

2.2.3 JavaScript [3]

All client-side code is written in JavaScript using either vanilla JavaScript or jQuery. jQuery is a JavaScript library invented by John Resig, which is now maintained by a team of developers at the jQuery Foundation. The jQuery library makes front-end development easier by simplifing things such as Animations, AJAX Operations, DOM Manipulation, Event Handling, and lot more. As a result, you can write fewer lines of code and accomplish more in a short amount of time. There have been a lot of performance tests (benchmarks) to actually see how much slower jQuery is than vanilla JavaScript. According to Marco Trombino, a front-end developer that tested a task that puts 10.000 new elements with a class inside a target element noticed that jQuery is moderately slower than vanilla JavaScript: *"As we could have expected Vanilla performed the task in 18,99 ms, whereas jQuery did it in 195,89 ms. Ten times faster."* [8]. As we can see, the only drawback of using jQuery is that it can be slower for very large and intensive projects, but obviously this is not the case for this relatively small project and the users won't even notice it.

2.2.4 AJAX [4]

The app uses XMLHttpRequests (Figure 11) to POST form data to the server [9] (note that AJAX is the safest and most reliable way to make HTTP requests). AJAX is better than conventional ways to send form data because we can send the form without having to refresh the page (this means that the user is happy because the state of the page never changes) and the data will be sent in the background with an async request.

```
var contactForm = $("#contact");
contactForm.on('submit', function(e) {
    e.preventDefault(); // stop the browser from submitting normally
    var xHttp = new XMLHttpRequest();
    xHttp.onreadystatechange = function() {
        if (xHttp.readyState == 4) { // if request is loaded
            if (xHttp.status === 200 || xHttp.status === 304) { // 200 ok or 304 not modified
                alert(xHttp.responseText);
            }
        }
    };
    var name = $('#name')[0].value;
    var email = $('#email')[0].value;
    var subject = $('#subject')[0].value;
    var message = $('#message')[0].value;
    var data = JSON.stringify(
        {
            "name": name,
            "email": email,
            "subject": subject,
            "message": message
        }
    );
    // Clear the form
    $('#name').val('');
    $('#email').val('');
    $('#subject').val('');
    $('#message').val('');
    // Send the data to the spceified URL
    xHttp.open("POST", "/message", true);
    xHttp.setRequestHeader("Content-type", "application/json");
    xHttp.send(data);
});
```

Figure 11: Sending forms with AJAX.

As you can see this form is sent as a JSON string with a POST method on the "/message" URL. Our Express app then interprets this in NodeJS and logs the received data to console (Figure 12)

```
// Log the message sent via form
// could log to file ************ TO DO ****************************
app.post('/message', (req, res) => {
    var payload = req.body;
    if (payload.name && payload.email && payload.subject && payload.message) {
        console.log
        (
            "You have a new message!" +
            "\nFrom: " + payload.name +
            "\nE-Mail: " + payload.email +
            "\nSubject: " + payload.subject +
            "\nMessage: " + payload.message
        );
        res.status(200);
        res.send("Message received. Thank you.");
    } else {
        res.status(400); // BAD REQUEST
        res.send("Bad request");
    }
```

Figure 12: Receiving the form on the server.

2.2.5 NodeJS [5]

The app uses various NodeJS modules that must be installed (Figure 13) like Express, Body-Parser, Path and 2 other files that I wrote and included in the main app file. There's another module which is not shown in this image: const Twitter = require('twitter');.

app.js

```
1  // DEPENDENCIES
2  const express = require('express');
3  const bodyParser = require('body-parser');
4  const path = require('path'); // simplify file paths
5  const twitt = require('./helpers/twitt.js');
6  const security = require('./helpers/security.js');
7
8  // GLOBALS
9  const app = express();
```

Figure 13: NodeJS modules. The Express app object is created on line 9.

Express [10] is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework: 1) Allows to set up middlewares to respond to HTTP Requests. 2) Defines a routing table which is used to perform different actions based on HTTP Method and URL. 3) Allows to dynamically render HTML Pages based on passing arguments to templates.
The app uses express to, among other things, handle routing (Figure 14): whenever an user makes a GET request to "/" then it renders a file called index.ejs (this is possible because, as pointed out earlier, the app uses EJS as a view engine) that sits inside the views directory.

```
// ROUTE HANDLERS
// INDEX PAGE
app.get('/', (req, res) => {
    res.render('index',
        {
            title: "Trending Hashtags Bot",
            tweetStatistics: twitt.myTweets,
            tweetStatisticsString: JSON.stringify(twitt.myTweets),
            tweetCount: twitt.tweetCount,
            updatedTimeMS: twitt.updateTime
        }
    );
});
```

Figure 14: Using the express framework.

Another important thing that express is used for is to create a server that listens on a certain port (Figure 15). This is also possible with a module called "http" but express makes things a lot easier because it uses this module behind the scenes and provides additional abstraction.

```
// SERVER UP AND RUNNING
var port = process.env.PORT;
if (port == null || port == "") {
    port = 8000;
}
app.listen(port, () => {
    console.log('Server listening on port ' + port);
    // first tweet list update
    twitt.getTweets(200); // 200 is the maximum amount we can get anyway, so we will display up to 200 tweets on the website
    console.log("App started and server is running: first tweet list update");
    //twitt.getWebhookID(); // commented because of limit for using the webhook APIs (15 calls every 15 min)
    //twitt.getWebhookSubscription(); // commented because of limits for using the webhook APIs (15 calls every 15min)
});
```

Figure 15: Listening on a port with express.

Body-Parser [11] parses incoming request bodies in a middleware before your handlers, available under the req.body property. The app uses this to parse JSON

content so that, for example, it can read the form data sent through an AJAX call in POST (Figure 16). The second line of code in Figure 16 is a middleware and it parses UTF-8 encoded bodies. It helps to parse URL encoded data like JSON objects

```
/* Body Parser Middleware (parsing json content)
* in order to read HTTP POST data , we have to use "body-parser" node module. body-parser
* is a piece of express middleware that reads a form's input and stores it as a javascript object accessible through req.body
*/
app.use(bodyParser.json());
app.use(bodyParser.urlencoded( {extended: true} ));
```

Figure 16: Using the body-Parser module.

Path [12] provides utilities for working with file and directory paths. This app uses the path module to define a static directory for our files (Figure 17) that we can call root. The purpose of path.join(__dirname, 'public') is to create an absolute path, using the directory where app.js is located as the base. In my example this piece of code will result in C:/Users/Jolsty/Desktop/twitter_project_heroku/public. In conclusion, the line of code in Figure 17 will simply create an absolute path for the app.

```
// Static Directory Middleware (public directory is the new root for css/images/javascript)
app.use(express.static(path.join(__dirname, 'public')));
```

Figure 17: Using the path module with an Express app.

Twitter [13] is an asynchronous client library for the Twitter REST and Streaming API's (Figure 18). This module is used to create a twitter object with the OAuth credentials. With this object it's possible to make requests to the Twitter APIs in a simple manner; we can, for example, make a GET request to "statuses/user_timeline" which translates to `https://api.twitter.com/1.1/statuses/user_timeline.json` which returns metadata for tweets on our account (Figure 2). We will go more in depth on this module on the next subsection to see how the app uses the APIs to get the necessary information from Twitter.

```
// DEPENDENCIES
const Twitter = require('twitter');

const twitter = new Twitter({
    consumer_key: process.env.TWITTER_CONSUMER_KEY,
    consumer_secret: process.env.TWITTER_CONSUMER_SECRET,
    access_token_key: process.env.TWITTER_ACCESS_TOKEN_KEY,
    access_token_secret: process.env.TWITTER_ACCESS_TOKEN_SECRET
});

// Gets a certain amount of tweets and places them into exported variables available for use in app.js
module.exports.getTweets = function(cnt, callback) {
    var myTweets = [];
    var count = {count: cnt};
    twitter.get('statuses/user_timeline', count, (err, data, res) => {
        if (err) throw err;
        if (res) {
            myTweets = filterTweets(data);
            module.exports.myTweets = myTweets;
            module.exports.tweetCount = myTweets.length;
            module.exports.updateTime = new Date().getTime();
            if (callback) callback(myTweets); // not every call we make uses the callback
        }
    });
}
```

Figure 18: Using the twitter module.

2.2.6 Heroku [6]

Heroku is a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud. I specifically chose this platform because it's very easy to use and it allows for deploying Node apps. The deployment [14] works by pushing your code to a specific git repository (which is called heroku remote, unique for every heroku app). Note that Heroku only deploys code that you push to the master branch of the heroku remote. Pushing code to another branch of the remote has no effect. After deployment, Heroku prepares the app for execution in a dyno - a smart container with a secure, curated Node stack.

2.2.7 Twitter APIs [7]

Twitter's developer platform includes numerous API endpoints and tools to help build an app and solution on Twitter. Integrating with nearly all of the Twitter APIs requires the creation of a Twitter app and the generation of consumer keys and access tokens. There are many endpoints that we can call that return various interesting data in JSON format, but we are only interested in some of them:

1. Trends (Figure 19)

Figure 19: Trends endpoint.

This endpoint returns the top 50 trending topics for a specific WOEID [15] (in our case the WOEID = 1 which represents the entire Earth), if trending information is available for it. The response is an array of trend objects that encode the name of the trending topic, the query parameter that can be used to search for the topic on Twitter Search, and the Twitter Search URL. This information is cached for 5 minutes. Requesting more frequently than

that will not return any more data. As you can see in Figure 20 we call the getTrendsAndTweet() function every hour. This function gets the current trending list and then tweets out a single random one. The actual code that gets the trends and stores them in an array that is returned to a callback function is in Figure 21.

```
const tweetTimer = 12 * 300000; // update trends every hour and then tweet out
setInterval( () => {
    getTrendsAndTweet();
}, tweetTimer);

function getTrendsAndTweet() {
    getHashtags(1, (trendingHashtags) => {
        tweetOut(trendingHashtags);
    });
}
```

Figure 20: Updating trends every hour.

```
function getHashtags(id, callback) {
    var woeid = {id: id};
    var trendingHashtags = [];
    // GET TRENDING HASHTAGS FOR SPECIFIC WOEID
    twitter.get('trends/place', woeid, (err, data, res) => {
        if (err) throw err;
        for (var i = 0; i < data[0].trends.length; i++) {
            trendingHashtags.push(data[0].trends[i].name);
        }
        if (res) callback(trendingHashtags); else throw err;
    });
}
```

Figure 21: Using the trends endpoint (code).

2. Posting tweets (Figure 22)

   This endpoint updates the authenticating user's current status, also known as Tweeting. For each update attempt, the update text is compared with the authenticating user's recent Tweets. Any attempt that would result in duplication will be blocked, resulting in a 403 error. A user cannot submit the same status twice in a row.

Example Request

```
  $ curl --request POST
--url 'https://api.twitter.com/1.1/statuses/update.json?
status=Test%20tweet%20using%20the%20POST%20statuses%2Fupdate%20endpoint'
--header 'authorization: OAuth oauth_consumer_key="YOUR_CONSUMER_KEY",
oauth_nonce="AUTO_GENERATED_NONCE", oauth_signature="AUTO_GENERATED_SIGNATURE",
oauth_signature_method="HMAC-SHA1", oauth_timestamp="AUTO_GENERATED_TIMESTAMP",
oauth_token="USERS_ACCESS_TOKEN", oauth_version="1.0"'
--header 'content-type: application/json'
```

Figure 22: Post tweets endpoint. Remember that the authentication is managed by the twitter module.

The code that does the actual tweeting can be seen in Figure 23.

```
function tweetOut(trendingHashtags){
    // GET RAND TWEET
    var randomTweet = getRandInt(trendingHashtags.length);
    // BODY OF TWEET
    var status = { status: trendingHashtags[randomTweet] };
    // TWEET OUT STATUS
    twitter.post('statuses/update', status, (err, data, res) => {
        if (err) throw err;
        console.log("Tweeted out '" + status.status + "' on " + new Date());
    });
}
```

Figure 23: Code to post tweets. The status is chosen randomly from the array of trends that is passed to the callback function in Figure 21.

3. Getting tweet metadata (Figure 24)

   This endpoint returns a collection of the most recent tweets posted by the user indicated by the screen_name or user_id parameters. User timelines belonging to protected users may only be requested when the authenticated user either owns the timeline or is an approved follower of the owner. The timeline returned is the equivalent of the one seen as a user's profile on twitter.com. This method can only return up to 3,200 of a user's most recent tweets.

Example Request

GET https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=twitterapi&count=2

Example Response

```
[
  {
    "created_at" : "Thu Apr 06 15:28:43 +0000 2017" ,
    "id" : 850007368138018817 ,
    "id_str" : "850007368138018817" ,
    "text" :
"RT @TwitterDev: 1/ Today we're sharing our vision for the future of the Twitter API platfor
m!
n https://t.co/XweGngmxlP" ,
    "truncated" : false ,
    "entities" : {
      "hashtags" : [],
      "symbols" : [],
      "user_mentions" : [
```

Figure 24: Get tweets endpoint. Returns tweet metadata for the caller's account.

In Figure 25 you can see the code to get the metadata. This function gets the recent tweets (the number is specified by the variable called cnt) from the account and, after filtering them through the filterTweets function, puts them in an exported variable that we can read from the main app.js file (this is so that we can render it for the client). We will also export a variable that holds the time when the list was updated and one to keep track of how many tweets were read (this can be different from the cnt variable because, for example, we can call this function with cnt = 200 but we don't actually have 200 tweets that can be returned).

```
// Gets a certain amount of tweets and places them into exported variables available for use in app.js
module.exports.getTweets = function(cnt, callback) {
    var myTweets = [];
    var count = {count: cnt};
    twitter.get('statuses/user_timeline', count, (err, data, res) => {
        if (err) throw err;
        if (res) {
            myTweets = filterTweets(data);
            module.exports.myTweets = myTweets;
            module.exports.tweetCount = myTweets.length;
            module.exports.updateTime = new Date().getTime();
            if (callback) callback(myTweets); // not every call we make uses the callback
        }
    });
}
```

Figure 25: Code to get tweets metadata.

4. Account activity APIs (based on webhooks) [16]

   Webhooks are "user-defined HTTP callbacks". They are usually triggered by some event, such as pushing code to a repository or a comment being posted to a blog. When that event occurs, the source site makes an HTTP request to the URL configured for the webhook. Users can configure them to cause events on one site to invoke behaviour on another. The action taken may be anything. Common uses are to trigger builds with continuous integration systems or to notify bug tracking systems. Since they use HTTP, they can be integrated into web services without adding new infrastructure. However, there are also ways to build a message queueing service on top of HTTP—some RESTful examples include IronMQ and RestMS. (source: Wikipedia)

   In the Summer of 2018 Twitter decided to replace (Figure 26) the usual streaming APIs with the Account Activity APIs, a webhook-based API that sends account events to a web app you develop, deploy and host. The Account Activity API provides you the ability to subscribe to realtime activities related to a user account via webhooks. This means that you can receive realtime Tweets, Direct Messages, and other account events from one or more of your owned or subscribed accounts through a single connection. All activity types can be seen in Figure 27.

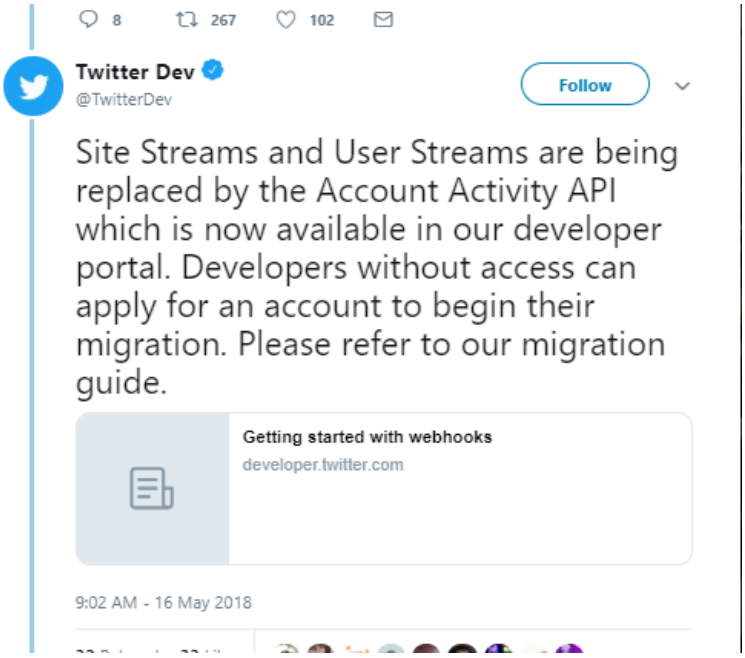

Figure 26: Twitter replaces streaming APIs.

Activity types

- Tweets (by user)
- Tweet deletes (by user)
- @mentions (of user)
- Replies (to or from user)
- Retweets (by user or of user)
- Quote Tweets (by user or of user)
- Retweets of Quoted Tweets (by user or of user)
- Likes (by user or of user)
- Follows (by user or of user)
- Unfollows (by user or of user)
- Blocks (by user)
- Unblocks (by user)
- Mutes (by user)
- Unmutes (by user)
- Direct Messages sent (by user)
- Direct Messages received (by user)
- Typing indicators (to user)
- Read receipts (to user)
- Subscription revokes (by user)

Figure 27: Events included with the account activity APIs.

In order to get access to these APIs you must request for access. The free version allows for 1 webhook connection at a time, which is enough for us since we are only interested in managing one account. Once you have received Account Activity API access, you need to develop, deploy and host a web app that will receive Twitter webhook events (the steps to follow can be seen in Figure 28).

- Create a web app with a URL to use as your webhook to receive events. This is the endpoint deployed on your server to listen for incoming Twitter webhook events.
  - The URI *path* is completely up to you. This example would be valid: https://mydomain.com/*service/listen*
  - If you are listening for webhooks from a variety of sources, a common pattern is: https://mydomain.com/webhook/twitter
  - Note that the specified URL can not include a port specification (https://mydomain.com:5000/NoWorkie).
- As described in our Securing Webhooks guide, a first step is writing code that receives a Twitter Challenge Response Check (CRC) GET request and responds with a properly formatted JSON response.
- Register your webhook URL. You will make a POST request to a /webhooks.json?url= endpoint. When you make this request Twitter will issue a CRC request to your web app. When a webhook is successfully registered, the response will include a webhook `id`. This webhook `id` is needed later when making some requests to the Account Activity API.
- Twitter will send account webhook events to the URL you registered. Make sure your web app supports POST requests for incoming events. These events will be encoded in JSON. See HERE for example webhook JSON payloads.
- Once your web app is ready, the next step is adding accounts to receive activities for. When adding (or deleting) accounts you will make POST requests referencing the account id. See our guide on adding subscriptions for more information.

Figure 28: Steps to follow to implement a webhook connection with Twitter.

I created a web app with an URL to use my webhook to receive events (Figure 29). Everytime the app receives an event, it updates the internal list of tweets by using the getTweets function (this is limited to once per second to avoid spam and therefore avoid blacklisting of my account).

```
/*
* RECEIVES EVENT NOTIFICATIONS FROM TWITTER
* TWEET_CREATE_EVENTS: tweets, retweets, replies, @mentions, quotetweets
* FAVORITE_EVENTS
* TWEET_DELETE_EVENTS
*/
app.post('/webhook/twitter', (req, res) => {
    var payload = req.body;
    if (payload.tweet_create_events || payload.favorite_events || payload.tweet_delete_events) {
        // update tweets (limit to one update per second if it's being spammed to avoid getting blacklisted)
        if (timeout === 0) {
            twitt.getTweets(200);
            console.log("Updated tweets list via webhook");
            timeout = 1;
            setTimeout( () => {
                timeout = 0;
            }, 1000);
        }
    }
    res.send('200 OK')
});
```

Figure 29: The app listens for POST requests on the specified URL, which refers to https://unimitwitterbot.herokuapp.com/webhook/twitter.

Then I implemented the code that resolves the Twitter Challenge Response Check: first of all we make a POST request to `/webhooks.json?url="myURL"` (Figure 30). According to Twitter, when a webhook is created they will issue a GET request with the CRC code and the app must respond with the requirements on Figure 31. The code that manages this GET request can be seen on Figure 32 and the code that creates the hash code on Figure 33.

```
// FUNCTIONS
//createWebhook('https://unimitwitterbot.herokuapp.com/webhook/twitter');
function createWebhook(url) {
    urls = { url: url };
    twitter.post('account_activity/all/accountactivity/webhooks', urls, (err, body, res) => {
        if (err) console.log("Err" + JSON.stringify(err));
        if (res) {
            console.log("createWebhook " + JSON.stringify(body));
            console.log("response " + JSON.stringify(res));
        }
        //if (res) getWebhook();
    });
}
```

Figure 30: Creating the webhook connection. Note that the commented out code was only executed once. The connection is persistent as long as we do atleast one validation a day.

The CRC request will occur:

- When a webhook URL is registered.
- Approximately *hourly* to validate your webhook URL.
- You can manually trigger a CRC by making a PUT request. As you develop your webhook client, you should plan on manually triggering the CRC as you develop your CRC response.

Response requirements:

- A base64 encoded HMAC SHA-256 hash created from the `crc_token` and your app Consumer Secret
- Valid response_token and JSON format.
- Latency less than 3 seconds.
- 200 HTTP response code.

Figure 31: CRC response requirements

```
* Receives and processes the challenge response check (CRC)
* Twitter does it almost hourly
*/
app.get('/webhook/twitter', (req, res) => {
    var crc_token = req.query.crc_token;
    if (crc_token) {
        console.log("crc token: " + crc_token);
        var hash = security.get_challenge_response(crc_token, twitt.twitter.options.consumer_secret);
        console.log("HASH " + hash);
        res.status(200); // STATUS OK
        res.send(
            {
                response_token: 'sha256=' + hash
            }
        );
        console.log("Sent response token back to twitter");
    } else {
        res.status(400); // STATUS BAD REQUEST
        console.log("Error: crc_token missing from request.")
        res.send('Error: crc_token missing from request.');
    }
});
```

Figure 32: CRC response (code).

```
const crypto = require('crypto');

/**
 * Creates a HMAC SHA-256 hash created from the app TOKEN and
 * your app Consumer Secret.
 * @param  token  the token provided by the incoming GET request
 * @return string
 */
module.exports.get_challenge_response = function(crc_token, consumer_secret) {

  //Compare created hash with the base64 encoded x-twitter-webhooks-signature value.
  //Use a method like compare_digest to reduce the vulnerability to timing attacks.

  hmac = crypto.createHmac('sha256', consumer_secret.toString()).update(crc_token).digest('base64');
  return hmac;
}

```

Figure 33: Generating the hash code.

The next step is registering a subscription for a certain user; in this case we register our own account (Figure 34). It's important to note that we need user-supplied access tokens to do this so, for security reasons, we can't do this for any random account.

```
//addWebhookSubscription();
function addWebhookSubscription() {
    twitter.post("account_activity/all/accountactivity/subscriptions", (err, body, res) => {
        if (err) console.log(err);
        if (res.statusCode === 204) console.log("Added webhook subscription");
        else console.log("Something happened with the webhook subscription");
    });
}
```

Figure 34: Code to add webhook subscription with our own access token key. Remember that the OAuth is being managed by the twitter module (we created a twitter object with our own API keys

The final step is testing the webhook: receiving events (Figure 29). An example of event is shown on Figure 35 after I favorited one of my own tweets.

```
2018-11-10T12:18:33.652539+00:00 heroku[web.1]: Starting process with command `node app.js`
2018-11-10T12:18:31.000000+00:00 app[api]: Build succeeded
2018-11-10T12:18:36.116655+00:00 app[web.1]: Server listening on port 10947
2018-11-10T12:18:36.118652+00:00 app[web.1]: App started and server is running: first tweet list update
2018-11-10T12:18:36.323092+00:00 app[web.1]: Authentication successful...
2018-11-10T12:18:36.323095+00:00 app[web.1]:
2018-11-10T12:18:37.370224+00:00 heroku[web.1]: State changed from starting to up
2018-11-10T12:18:42.449920+00:00 app[web.1]: { for_user_id: '1050875420881629184',
2018-11-10T12:18:42.449968+00:00 app[web.1]: favorite_events:
2018-11-10T12:18:42.449970+00:00 app[web.1]: [ { id: 'eaa8723b7b0890a1ef9e26d37e87a5e3',
2018-11-10T12:18:42.449974+00:00 app[web.1]: created_at: 'Sat Nov 10 12:18:42 +0000 2018',
2018-11-10T12:18:42.449975+00:00 app[web.1]: timestamp_ms: 1541852322268,
2018-11-10T12:18:42.449976+00:00 app[web.1]: favorited_status: [Object],
2018-11-10T12:18:42.449978+00:00 app[web.1]: user: [Object] } ] }
2018-11-10T12:18:42.450891+00:00 app[web.1]: Updated tweets list via webhook
2018-11-10T12:18:42.454610+00:00 heroku[router]: at=info method=POST path="/webhook/twitter" host=unimitwitterbot.herokuapp.com request
_id=2447dfe0-84c2-4ac3-b942-3372e901ca7c fwd="199.16.157.172" dyno=web.1 connect=1ms service=23ms status=200 bytes=204 protocol=https
```

Figure 35: Example of a favorite_event payload

Note that we use the webhooks in order to update our internal list of tweets (this way everytime an event happens, like a tweet or favorite event our

list is updated). There's another, much simpler, way to do this but less efficient and it's based on updating the list every second. This means that we may make useless calls because the list won't change every second as opposed to update it every time something happens. This is how the app was implemented in the earlier stages and I still have the code but it's been commented out (Figure 36).

```
// INTERVALS
// const getTweetsTimer = 1000; // update our tweet list every second

/*
* Update tweets manually. This is another way to do it.
*
setInterval( () => {
    getTweets(200, (myTweets) => {
        console.log("Updated the tweet list");
    });
}, getTweetsTimer);
*/
```

Figure 36: Earlier implementation of the list update

3 Additional attachments

3.1 First Draft

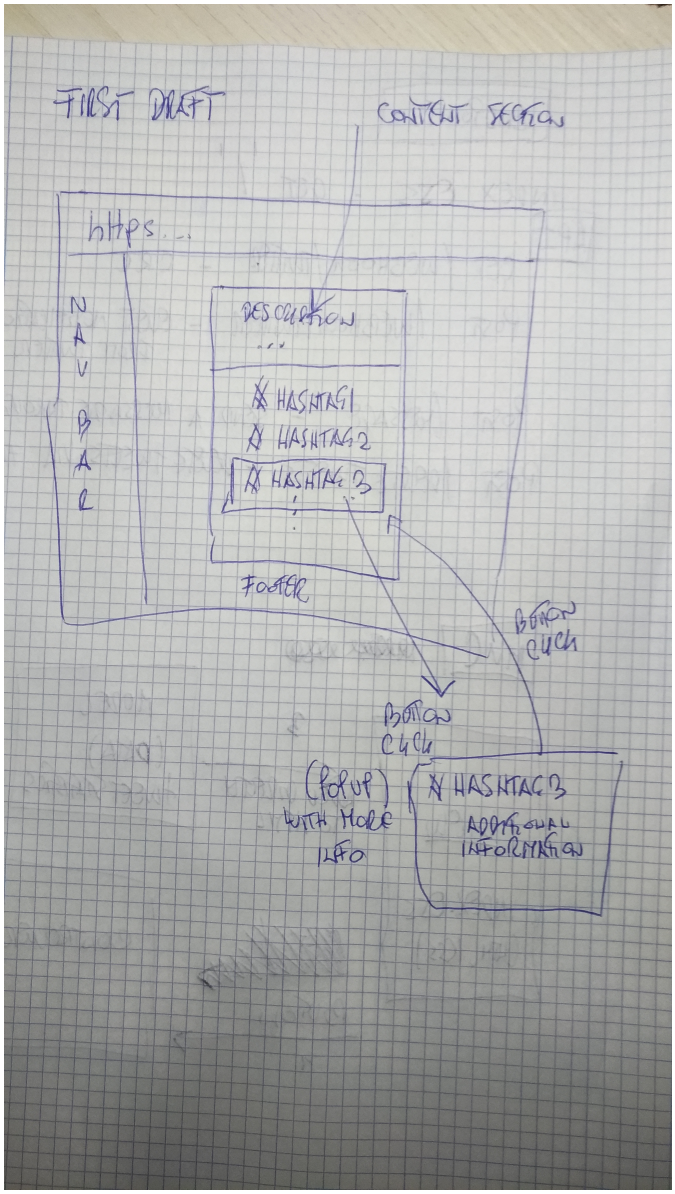

Figure 37: First draft of website

3.2 Application API Endpoints

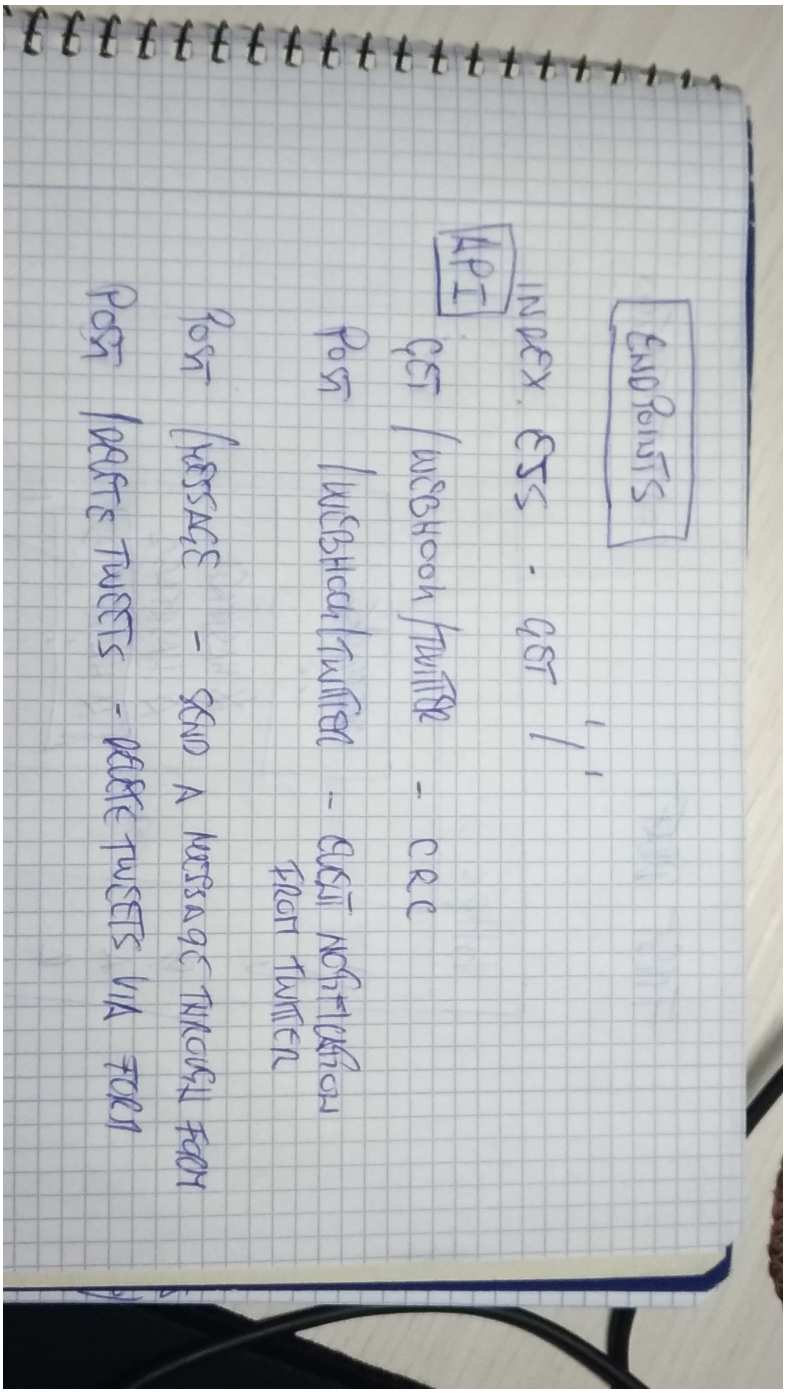

Figure 38: API endpoints for application

3.3 MVC example

Figure 39: MVC example for how the button click updates our view

4 Conclusion

The purpose of this project was to get accustomed to some of the most important web development technologies such as HTML5, CSS (front-end), Javascript (front-end and back-end) and NodeJS (back-end). The app itself is not very useful to customers but it's not a commercial product and I only chose to do this project in order to learn web development. It's taken me abouth a month to develop this app (including the project report which was written in LaTeX and was quite time consuming). All in all it was a good experience for me especially because I didn't know anything about web development beforehand and I quite like it and might pursue it further after graduating.

References

[1] W3 Schools. Html5 introduction. https://www.w3schools.com/html/html5_intro.asp.

[2] W3 Schools. Css tutorial. https://www.w3schools.com/css/.

[3] Pluralsight. Ready to try javascript? https://www.javascript.com/.

[4] W3 Schools. Ajax introduction. https://www.w3schools.com/js/js_ajax_intro.asp.

[5] NodeJS. Node.js® is a javascript runtime built on chrome's v8 javascript engine. https://nodejs.org/en/.

[6] Heroku. Learn about building, deploying, and managing your apps on heroku. https://devcenter.heroku.com/.

[7] Twitter. Getting started with the twitter apis. https://developer.twitter.com/en/docs/basics/getting-started.

[8] Marco Trombino. You might not need jquery: A 2018 performance case study. May 2018. https://medium.com/@trombino.marco/you-might-not-need-jquery-a-2018-performance-case-study-aa6531d0b0c3.

[9] Last update by: peterschussheim. Sending forms through javascript. Aug 2018. https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Sending_forms_through_JavaScript.

[10] N/A. Node.js - express framework. https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm.

[11] dougwilson. body-parser module. https://www.npmjs.com/package/body-parser.

[12] N/A. path module. https://nodejs.org/docs/latest/api/path.html.

[13] desmondmorris. twitter module. https://www.npmjs.com/package/twitter.

[14] Heroku. Deploying with git. https://devcenter.heroku.com/articles/git.

[15] Ross Elliot. Getting the woeid of a location. http://woeid.rosselliot.co.nz/.

[16] Twitter. Getting started with webhooks. https://developer.twitter.com/en/docs/accounts-and-users/subscribe-account-activity/guides/getting-started-with-webhooks.