Assignment Two: JavaScript

Due: February 15, 2013 @ 11:59pm

In this assignment, you'll catch a fever: Bieber fever. You will be implementing the BieberFeed, a tool for displaying a streaming list of Tweets posted about Justin Bieber.

Important Note: this project uses Ajax that violates the same-origin policy, and as such, it requires a web browser that supports XMLHttpRequest 2 and CORS. This means you **must** be using a semi-recent web browser: Firefox 4+, Chrome 7+, Safari 5+, or IE 10+.

The Problem

Every second, there are nearly 50 new Tweets published about Justin Bieber. It can be quite harrowing to keep track of all of them on Twitter itself, so you are being tasked with finding a way to separate the proverbial wheat from the chaff.

Requirements

We have provided a web server that will provide you with the 25 most recent Tweets about Justin Bieber every time you ask for them. You will build a website that takes those Tweets and presents them to the user in some kind of a streaming list.

You will be responsible for periodically querying the server for new Tweets, and using the DOM to append them to a new part of the page. You will also be responsible for filtering out duplicates, as some Tweets may overlap with the last time you requested them.

Getting Started

Create a new directory to hold your project's files, and then run:

```
cs132_install javascript
```

You can also download the assignment from GitHub: https://github.com/brown-cs132-s13/javascript/archive/master.zip.

This will give you a blank `README.md` file (to contain known bugs and any features you want to highlight), a very barebones shell of an HTML file, and a picture of Justin Bieber's face that you may use and modify as you see fit in the design of your project. There is also a "no photo" placeholder image you might want to use for Tweets that don't have pictures associated with them.

You should implement all of your HTML in `bieber.html`, and all of your JavaScript in `bieber.js`.

— The BieberFeed Server —

You will be interacting with a server we've provided that will give you Bieber-related Tweets. The server is located at `http://bieber.mattpatenaude.com` and responds to the following requests:

```
/feed/:login
```

Make a `GET` request to `/feed/<your CS username>` to obtain a new block of 25 Bieber-related Tweets. The response will be returned as a JSON array. Take note that some of these Tweets may be duplicates of Tweets that you've already seen, and it is **your responsibility** to filter those out. Every Tweet has a unique `id` field that you can use for this purpose.

The available properties are those returned by the Twitter API for Tweets. Note that Tweet entities are supported, so you should consider using them to create a more engaging user experience.

```
/feed/stats/:login
```

Make a `GET` request to `/feed/stats/<your CS username>` to obtain statistical information on the Tweets the server is providing, including the total number of Tweets available, the ID of the last-known Tweet, and an array of the search terms the server is using to find new Tweets. For example:

```
{
  "count" : 24,  // number of tweets
  "last" : "", // id of the last tweet
  "terms" : [ "bieber" , "justin bieber", "@justinbieber" ] // the search terms used to query the
Twitter streaming API
}
```

— Using Ajax —

Since the idea of the BieberFeed is that it's a live stream of Bieber-oriented Tweets, having the page refresh every second with new content would be a bit annoying. For that reason, we'll be using a technique called Ajax, which historically stands for Asynchronous JavaScript and XML, but actually applies to any kind of data (text, JSON, and more).

Ajax allows you to make an HTTP request to a website (usually your own, see below) in the background, and then be notified of the results after the content has been fetched. This all happens without the need to refresh the page, so you can then act on the new information immediately from JavaScript.

The standard template for an Ajax request looks something like this:

```
// create a request object
var request = new XMLHttpRequest();

// specify the HTTP method, URL, and asynchronous flag
request.open('GET', 'http://www.example.com/content.json', true);

// add an event handler
request.addEventListener('load', function(e){
    if (request.status == 200) {
        // do something with the loaded content
        var content = request.responseText;
    } else {
        // something went wrong, check the request status
        // hint: 403 means Forbidden, maybe you forgot your username?
    }
}, false);

// start the request, optionally with a request body for POST requests
request.send(null);
```

You might want to encapsulate this into a simple function to make your life easier - think about how you could write a function that works like `request(theURL, callback)`.

Combine the above template with something like `setInterval(...)` (see the Mozilla Developer Network's documentation) to periodically load new Tweets. Note that the timeout value passed to `setInterval(...)` is in milliseconds, and we ask that you don't use a value smaller than 3000 (3 seconds) so as not to overload our server.

If you want to learn more, there's a good tutorial on Using XMLHttpRequest on the Mozilla Developer Network.

Aside: the Same-Origin Policy

In order to prevent a particularly malicious class of attacks known as cross-site request forgery, web browsers enforce a policy known as the same-origin policy. Essentially, the same-origin policy places restrictions on what can be done with content that originates from a different *origin* than that of the page you're currently on (an origin being a combination of a domain name, port, and protocol - HTTP or HTTPS).

By default, Ajax requests to websites on different origins are **forbidden** by web browsers. The reason for this is that I could, for instance, make an Ajax request from my website at `evilsite.com` to `facebook.com`, and the browser would send along all of your saved cookies for Facebook, meaning I would have unrestricted access to your (fully logged in!) Facebook session. I could somewhat trivially construct a website that, just by virtue of visiting it, would "like" my company's page on Facebook (or something much more malicious). Thus, the same-origin policy is a "Good Thing".

However, there are times when the same-origin policy makes it very inconvenient to do legitimate things - like on this project. To that end, a new standard called CORS has evolved. By adding a few headers to our responses from the BieberFeed server:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization, Content-Length, X-Requested-With
```

... we are able to inform the browser that "yes, this website will allow other websites to make requests to it." In particular, `Access-Control-Allow-Origin: *` says that our server will respond to requests from *any* origin. In practice, you would want to place a list of domain names here that your website is explicitly allowing requests from (for example, Facebook might not want to allow `evilsite.com` to make Ajax requests to it, but it might allow `instagram.com` to do it).

— Parsing JSON —

The responses from the BieberFeed server will come back as JSON, which is a data storage format based on the object-literal notation in JavaScript. JSON objects are actually valid JavaScript code, but because that leaves the door wide open for nasty things, you don't want to just dump it in an `eval(...)`. Parsing JSON safely is fairly straightforward:

```
// inside your Ajax response handler
var content = request.responseText;
var data = JSON.parse(content);
```

The `data` object now holds a JavaScript representation of the data that was in `content` as a string. So, for example, in response to the `/feed/:login` request, `data` would be an array, each element of which would be an object with the properties of an individual Tweet.

Appending the Tweets

You'll want to come up with some way to insert new Tweets into the page. Usually, you'll do this (for example) with `document.createElement(...)`:

```
// we have an unordered list of Tweets
var ul = document.getElementById('tweets');

// create a new li element for the Tweet, and append it
var li = document.createElement('li');
li.innerHTML = '<strong>' + data[0].user.name + '</strong> ' + data[0].text;
ul.appendChild(li);
```

You might want to use `element.insertBefore(...)` instead of `element.appendChild(...)` depending on how you want to indicate Tweet chronology (newest on top vs. newest on bottom). The Mozilla Developer Network documentation for HTML elements includes a large number of DOM methods you could use to make your life easier.

Other Niceties

Think about how your choice of mechanisms will affect the user experience of the BieberFeed. Do new Tweets appear at the top, or the bottom? Does the page keep growing in length, or do you start removing old Tweets after a certain point? Do Tweets appear several at a time, or spaced out? Can the user start and stop the feed (hint: look up `clearInterval(...)`)? Is the list of Tweets independently scrollable (hint: lookup `overflow-y: auto`), or does the entire page scroll with it?

Add any features you can to try to make the experience of looking at so many Bieber Tweets bearable.

Handing In

Before handing in your project, make sure you've checked to make sure you've filled in your `README.md` file (you can use any format you like for your README, but Markdown is highly recommended).

To hand in your project, from your project directory, run:

```
cs132_handin javascript
```

That's it!