# Javascript Programming For Twitter API 1.1

By Adam Green

CEO, 140dev.com

# Javascript Programming for Twitter API 1.1

Adam Green

Adam Green Press

Lexington, Massachusetts

## Twitter API 1.1 changes everything

Javascript programmers have to change their basic way of coding when they switch to Twitter API 1.1. With API 1.0 your web page could make an Ajax request directly from Twitter with a callback argument. The response would come back as JSON or XML, which you could parse and then display on the web page.  API 1.1 breaks all of that Javascript code.

With version 1.1 you have to make an OAuth connection to the API before any requests, and I do mean any. That includes getting a user's timeline or even the Twitter home timeline. Twitter search in API 1.1 also requires OAuth. In addition, all RSS feeds from Twitter will be turned off with API 1.1. They too will have to be replaced by code that uses OAuth. Twitter says that version 1.0 will be shut down in March, so it is time to convert your code

Twitter isn't doing this to be mean or evil. Twitter API 1.1 is still free, and it increases many of the rate limits. The problem has been that API 1.0 didn't give Twitter any control over who was grabbing huge quantities of data. The most Twitter could do was block an entire IP address from any access, which is crude and often punishes the innocent on shared servers. The OAuth requirement in API 1.1 allows Twitter to monitor and control who is using the API down to an individual user. That seems fair in return for free access to so much useful data.

## OAuth is not practical with Javascript

It may sound like you can just add OAuth code to your Javascript to solve the problem, but that won't work. OAuth requires the use of special tokens that identify an app and a user who has authorized that app. You have to pass those tokens to the API in every script that calls API 1.1. You can technically do that in Javascript, but it requires you to put the tokens into your web page where they can be read by anyone who looks at the page source. Once someone has the OAuth tokens for an account, they can change the account, make it send tweets or follows, and connect it to their own scripts that may violate the Twitter terms of service.

As a result, making a direct connection from Javascript  to the Twitter API is effectively turned off as a viable programming technique. Kind of sad, but things change and we have to move on.

## Client–Server architecture solves the OAuth issue

Luckily, the solution is pretty straight forward. Instead of calling the Twitter API, your Javascript code can call your own server with Ajax. Your server can call the Twitter API with OAuth tokens from a script  that can't be seen by the public. When the result comes back, your server can return it to your Javascript code. This is a standard model called client-server programming, where the web browser running Javascript is the client.

Javascript programming with the Twitter API has always used a client-server model, except the server was Twitter's. The only change with API 1.1 is adding your own web server to the path. If your website is on a server that allows you to run scripts, you have what you need. If not, getting a web server that can handle this style of programming can be as inexpensive as $5 a month. The basic requirement is that the server needs to let you run code in some programming language, such as PHP, that can communicate with the Twitter API using OAuth.

## Single-user OAuth spoken here

There are two forms of OAuth for use in Twitter API programming: single-user and multi-user. Single-user OAuth involves a Twitter application that has itself as the only authorized user. Twitter gives you a complete set of OAuth tokens for that application, which can then be used to change only the account that owns the application. This is a great solution for code that needs to change a single account, such as tweeting when someone performs an action on your website, or following a user who gives you their Twitter screen name on your website. The good news is that single-user OAuth also lets you use the API to get information about any Twitter account that isn't protected. If you want to get any user's timeline or call Twitter search, single-user OAuth will do the job.

Multi-user OAuth allows you to connect to any Twitter account to make changes, but you have to set up a user interface on your website that allows users to log into Twitter. You also have to keep track of which user has already authorized your Twitter app, and get their OAuth tokens for each API request. These programming techniques are beyond the scope of this short ebook. We'll limit ourselves to single-user OAuth here.

The particular style of single-user OAuth coding used here is documented in a separate free ebook available on my 140dev.com website. It uses the tmhOAuth library, which is also free, and the PHP language, and that's free too. Isn't the Internet wonderful?

## But what about rate limits?

Twitter API 1.1 also makes dramatic changes to the rate limit model for Javascript. With API 1.0 the rate limit in Javascript was applied to the user's IP address, and ranged between 200 to 350 calls per hour depending on the part of the API used. Every IP address had their own rate limit available to them when running Javascript in API 1.0.

With single-user OAuth in API 1.1, all visitors to your website will be sharing the rate limit available to your Twitter app. The good thing is that rate limits for queries, such as user timelines and search are increased to 720 requests per hour in API 1.1. This is certainly enough for personal and small commercial websites. If your website gets more than 720 page views per hour, you will have to code with multi-user OAuth to make Javascript work.

### What will you learn here?

The goal of this ebook is to teach you the basic techniques needed to program with Javascript and single-user OAuth in Twitter API 1.1. I use the jQuery library for Javascript programming, so we'll start with a brief tutorial on jQuery and Ajax. On the server side I use PHP and the tmhOAuth library to connect to the Twitter API and respond to web page Ajax requests. We'll look at simple examples of gathering tweets from user timelines and Twitter search. Then we'll apply these techniques to a complete client app for searching Twitter and displaying the results.

### Programming tools and skills required

To use the example code shown here all you need is a web server that can deliver HTML pages, and run PHP code. You can run all the examples through any web browser. Of course, you'll need some form of FTP to place code onto your server.

### Source code is available for download

All of the source code shown in this ebook can be downloaded from the 140dev.com website at the URL: http://140dev.com/member. You will find detailed install instructions within the downloadable zip file, but basically all you have to do is place the downloaded code within a web accessible directory on any server. The source code zip also includes the library files needed for jQuery and Twitter OAuth programming. If you are not familiar with single-user OAuth, I recommend downloading my ebook on this subject along with its source code, and reading that first.

## Getting Started with jQuery

jQuery is a free Javascript library that dramatically simplifies the tasks of manipulating web content and running Ajax communication with servers. There are other Javascript libraries you can use in your Twitter API programming, but I find jQuery to be the fastest for performance and the easiest to program. There is a wealth of free jQuery plugins, especially for user interface programming.

You can download the latest copy of this library from http://jquery.com/download/. The source code for this ebook contains a copy of the compressed version of jQuery in the form of **jquery.min.js**.

Before we get started with actual Twitter programming, let's go through a brief tutorial on the jQuery techniques you'll need most often, including Ajax requests from servers. This section assumes that you are familiar with Javascript, but haven't used jQuery before.

## Common jQuery actions

### *Load the library*

```
<script src="jquery.min.js"></script>
```

You must load the jQuery library in your HTML file before running any jQuery commands. It can be loaded from your local server or a remote URL. There is a lot of controversy as to the best location of this load command for performance, with some advocating that it be placed in the <head> section before the page <body>, and others in favor of the end of the <body>. This tutorial will put the jQuery at the start of the pages for simplicity. In later examples it will be placed in the <head>. The one rule to follow is to put this statement before any other jQuery.

### *Run once the page loads*

```
jQuery(document).ready(function($) {
```

jQuery processing code begins with this line. It guarantees that the entire page has loaded and all HTML elements were created before the jQuery code begins executing.

### *Work with HTML elements*

```
<span id='test_span'>Testing jQuery</span>
```

jQuery works with HTML elements, which can be a , , or any other element. You tell jQuery to read from or write to elements by type, or by name. If you assign either an id or class to an element, you can use that name to identify it. If an element has an id of **test_span**, you identify it to jQuery with **$('# test_span')**. If it is a class, you use **$('. test_span')**.

### *Attach click event to HTML elements*

```
$('# test_span').click(function(){
```

One of the coolest parts of jQuery is the way it lets you attach UI events to HTML elements, like reacting to a click. This can be done with any elements, so you are no longer limited to form elements, such as a button. The event creates an unnamed function that can run any other Javascript or jQuery. The result is a highly responsive page.

### *Read and write HTML element values*

```
var element_text = $('#test_span').html();
$('#test_span').html(element_text);
```

The **.html()** function can be used to read or write any HTML element's value. When **.html()** is given a new value, it appears on the page immediately. In effect, you can write to any portion of the page.

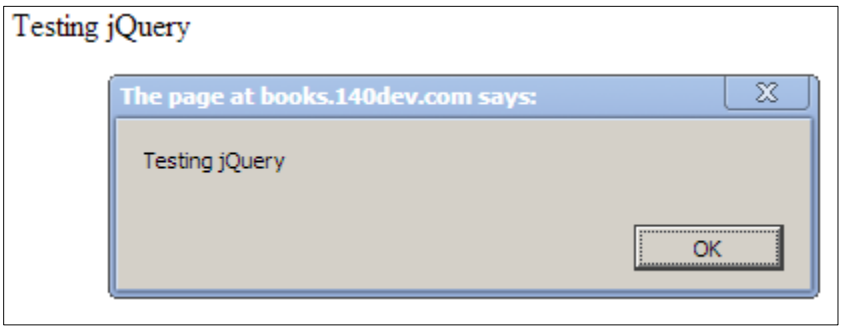
## Sample jQuery code

Putting this all together, we get a simple first example of jQuery in action.

*jquery_test.html*

```
<script src="jquery.min.js"></script>
<script>
  // Run once the page is loaded
  jQuery(document).ready(function($) {

      // Attach a click event to the span element
      $('#test_span').click(function(){

          // Read the element's current value
          var element_text = $('#test_span').html();

          // Display the current value of this span
          alert(element_text);

          // Convert the value to upper case
          element_text = element_text.toUpperCase();

          // Change the span to this new value
          $('#test_span').html(element_text);
      });
  });
</script>

<span id='test_span'>Testing jQuery</span>
```

The resulting page displays **Testing jQuery**. If you click this text, it appears in an alert window. When you close the window, the text changes to **TESTING JQUERY**. Admittedly a trivial example, but pretty amazing in its simplicity and implied power.



## Ajax must call a server within the same domain

A major restriction for Ajax programming in Javascript is that both the web page and the server must be in the same domain. So if your website is at http://mydomain.com, you can only communicate with a server URL at http://mydomain.com.

Different subdirectories within a domain are allowed. A page at http://mydomain.com/site can call server code at http://mydomain.com/code. Different subdomains will not work. A page at http://mydomain.com can't call server code at http://code.mydomain.com.

With Twitter API 1.0 you could make an Ajax request directly to the Twitter server with a technique called JSONP, which allowed you to communicate with a server from any domain. Setting up JSONP on your own server is beyond the scope of this ebook, since it requires additional security precautions. For the purposes of the sample code here, you should assume that you can only call the server based scripts from a web page on the same domain. This protects you from hacking attempts from computers outside your domain.

## Ajax request from a web page

If you've ever tried Ajax programming with basic Javascript, you will be amazed at jQuery's simplicity. Using the **ajax()** function, you simply specify the URL you want to communicate with. You then specify a **success** parameter that accepts the response in a variable that is named **data** by convention. When the Ajax call is executed and gets the result from the server, the **success** parameter runs an unnamed function with this **data** variable. You can also create an **error** parameter to handle a failed response. The **ajax_request.html** page below puts it all together. Remember, you MUST change the URL in this code to point to the directory on YOUR server containing a copy of the ebook's source code.

*ajax_request.html*

```
<script src='jquery.min.js'></script>

<script>

  jQuery(document).ready(function($) {
      // Make an Ajax request
      $.ajax({

          // Define the URL being called by Ajax
          // This URL is for illustration only
          // You MUST change it to call your server
          url: 'http://books.140dev.com/ebook_js/code/ajax_response.php',

          // Put the results into the display element
          success: function(data){
```

```
            $('#ajax_results').html(data);
        },

        // Display an error message if the request fails
        error: function(data) {
            $('#ajax_results').html('Ajax request failed');
        }
    })
});
</script>

Testing Ajax: <span id='ajax_results'></span>
```

### Response from a server

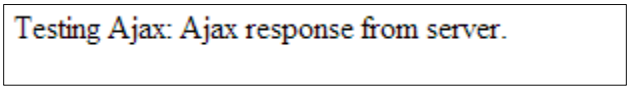
All the server script has to do when called by an Ajax request is print the results. The printed data will be returned to the requesting page where it can be read and displayed, or processed in some other way. All the examples in this ebook use PHP for the server code, but one of the beauties of Ajax is that it can communicate with server scripts written in any programming language. All of the examples return simple text data or HTML, but for more complex data you might choose to return JSON formatted results. Either way, you still print the results and the Ajax request picks it up.

The combination of **ajax_request.html** and **ajax_response.php** is very basic, but it shows just how easy Ajax can be with jQuery.

*ajax_response.php*

```
<?php

// Return results to the Ajax request
print "Ajax response from server.";

?>
```

Loading **ajax_request.html** in a web browser shows the server response a few seconds later.




## Using a server as a Twitter API proxy

A proxy server sounds technically complex, but all it really means is a server that sits between the browser client and the final server that is the target of a request. In this case we need to create a server script that can handle the OAuth portion of a Twitter API request. The complete

solution is easy. Just call your own server with Ajax from your Javascript code, and have it communicate with Twitter using OAuth. Then have your server return the results to your web page Javascript code.

## Getting a user timeline

Let's try a simple server proxy for getting a user's timeline and displaying it on a web page. This is a common use for Javascript with API 1.0, and there are hundreds of thousands of web pages with Twitter timelines that will break unless they use this new proxy model. We start with a web page that calls our server code and displays the result.

*timeline_request.html*

```
<script src='jquery.min.js'></script>

<script>

  jQuery(document).ready(function($) {
      // Make an Ajax request
      $.ajax({

          // Define the URL being called by Ajax
          // This URL is for illustration only
          // You MUST change it to call your server
          url: 'http://books.140dev.com/ebook_js/code/timeline_response.php',

          // Put the results into the display element
          success: function(data){
              $('#timeline_results').html(data);
          },

          // Display an error message if the request fails
          error: function(data) {
              $('#timeline_results').html('Ajax request failed');
          }
      })
  });
</script>

<strong>@Justin Bieber's Tweets:</strong><br/><br/>
<span id='timeline_results'></span>
```

The server code in **timeline_response.php** called by the Ajax request uses the single-user OAuth coding techniques described in my OAuth ebook. After getting the user timeline for the @justinbieber account, it extracts the tweets, assembles them into a single string of HTML, and prints the results. It also checks for the rate limit error code of 429, or any other error, and prints an appropriate message. For simplicity it only prints the text of each tweet.

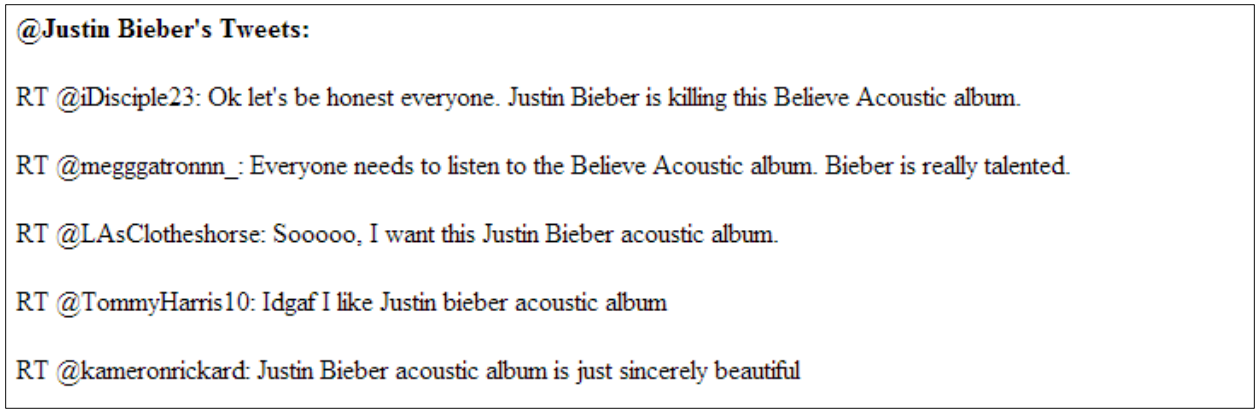
The ebook's sample source code comes with a file called app_tokens.php. This contains the OAuth tokens the code needs to make an OAuth connection with Twitter. Before running the code, you must change this file to use your own OAuth tokens. The details for getting a set of OAuth tokens are documented in my OAuth ebook.

*timeline_response.php*

```
<?php

// You MUST modify app_tokens.php to use your own Oauth tokens
require 'app_tokens.php';

// Create an OAuth connection
require 'tmhOAuth.php';

$connection = new tmhOAuth(array(
 'consumer_key'    => $consumer_key,
 'consumer_secret' => $consumer_secret,
 'user_token'      => $user_token,
 'user_secret'     => $user_secret
));

// Get the timeline with the Twitter API
$http_code = $connection->request('GET',$connection->url('1.1/statuses/user_timeline'),
  array('screen_name' => 'justinbieber',
      'count' => 100));

// Request was successful
if ($http_code == 200) {

  // Extract the tweets from the API response
  $tweet_data = json_decode($connection->response['response'],true);

  // Accumulate tweets from results
  $tweet_stream = '';
  foreach($tweet_data as $tweet) {

      // Add this tweet's text to the results
      $tweet_stream .= $tweet['text'] . '<br/><br/>';
  }

  // Send the tweets back to the Ajax request
  print $tweet_stream;

// Handle errors from API request
} else {
  if ($http_code == 429) {
      print 'Error: Twitter API rate limit reached';
  } else {
```

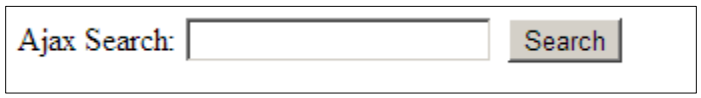
```
        print 'Error: Twitter was not able to process that request';
    }
}
?>
```

Loading the **timeline_request.html** page in a browser displays the latest tweets for this account. Of course, in a real web page you would use CSS to combine this tweet stream with the rest of the page in an attractive design. We'll look at this type of CSS for tweets later in the ebook.




## Getting Twitter search results

The same type of Ajax request and response model can be used to proxy requests from the search API. In **search_request.html** we also need to create a Javascript form to get the search terms that are passed to **search_response.php**.




Since this is an Ajax form, we don't want to reload the page when the user clicks the **Search** button, so we don't actually create an HTML form. Instead we create the input field and button elements, and then use jQuery to attach a click event to the button. When the user clicks the button, the current value of the input field is read and passed to the **search_response.php** script with Ajax. The result is a page that lets the user enter new search words, click **Search**, and see the results repeatedly without the page reloading.

Because the Ajax request uses a URL to call the server, the search terms must be part of a valid URL. A user can enter multiple search words separated by spaces in the input field, and spaces are not valid in a URL. The **search_request.html** script uses the Javascript **encodeURIComponent()** function to convert spaces into entities that are acceptable in a URL.

*Search_request.html*

```
<script src='jquery.min.js'></script>

<script>

  jQuery(document).ready(function($) {

      // Run when Search button is clicked
      $('#search_button').click(function(){

          // This can take a few seconds so display progress text
          $('#search_results').html('Searching Twitter...');

          // Get the value of the input field
          // Encode it for use in a URL
          var search_value = encodeURIComponent($('input[name=search_terms]').val());

          // Send the search terms to the server in an Ajax request
          // This URL is for illustration only
          // You MUST change it to your server
          $.ajax({
              url: 'http://books.140dev.com/ebook_js/code/search_response.php?q=' + search_value,
              success: function(data){

                  // Display the results
                  $('#search_results').html(data);
              }
          })
      })
  });
</script>

Ajax Search:
<input name='search_terms' autofocus='autofocus'/>
<button id='search_button'>Search</button>

<div id='search_results'></div>
```

When **search_response.php** is called with Ajax, it extracts the value of the **q** argument that was sent in the URL, and uses that to call the Twitter search API.

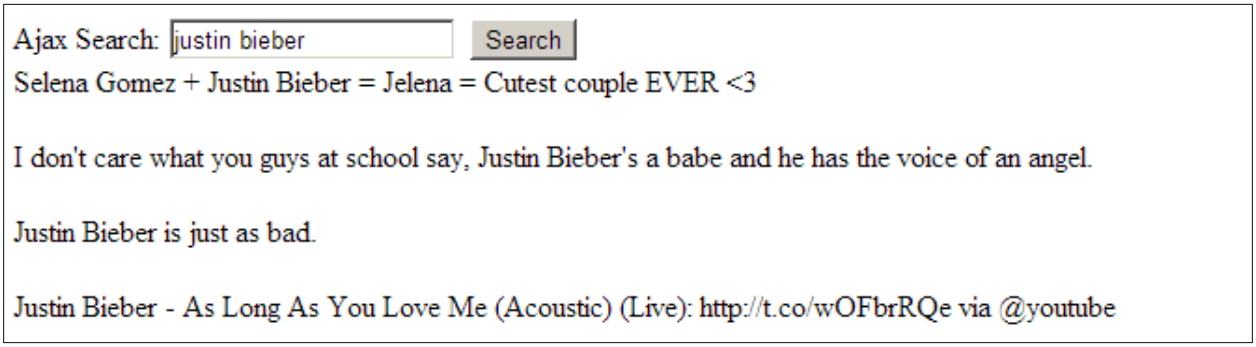
*search_response.php*

```
<?php

// Make sure search terms were sent
if (!empty($_GET['q'])) {
```

```
// Strip any dangerous text out of the search
$search_terms = htmlspecialchars($_GET['q']);

// Create an OAuth connection
require 'app_tokens.php';
require 'tmhOAuth.php';
$connection = new tmhOAuth(array(
  'consumer_key'    => $consumer_key,
  'consumer_secret' => $consumer_secret,
  'user_token'      => $user_token,
  'user_secret'     => $user_secret
));

// Run the search with the Twitter API
$http_code = $connection->request('GET',$connection->url('1.1/search/tweets'),
          array('q' => $search_terms,
          'count' => 100,
          'lang' => 'en',
              'type' => 'recent'));

// Search was successful
if ($http_code == 200) {

    // Extract the tweets from the API response
    $response = json_decode($connection->response['response'],true);
    $tweet_data = $response['statuses'];

    // Accumulate tweets from search results
    $tweet_stream = '';
    foreach($tweet_data as $tweet) {

        // Ignore retweets
        if (isset($tweet['retweeted_status'])) {
            continue;
        }

        // Add this tweet's text to the results
        $tweet_stream .= $tweet['text'] . '<br/><br/>';
    }

    // Send the result tweets back to the Ajax request
    print $tweet_stream;

// Handle errors from API request
} else {
    if ($http_code == 429) {
        print 'Error: Twitter API rate limit reached';
    } else {
        print 'Error: Twitter was not able to process that search';
```

```
        }
    }

} else {
    print 'No search terms found';
}

?>
```

The result is a simple search page that uses Ajax to create a responsive user interface, and a server to proxy the API request with Twitter. This is the foundation for any form based interface that allows users to communicate with the Twitter API. It still needs CSS formatting, and we will want to display a real tweet with all of its associated information. That comes next.




## Model search client application

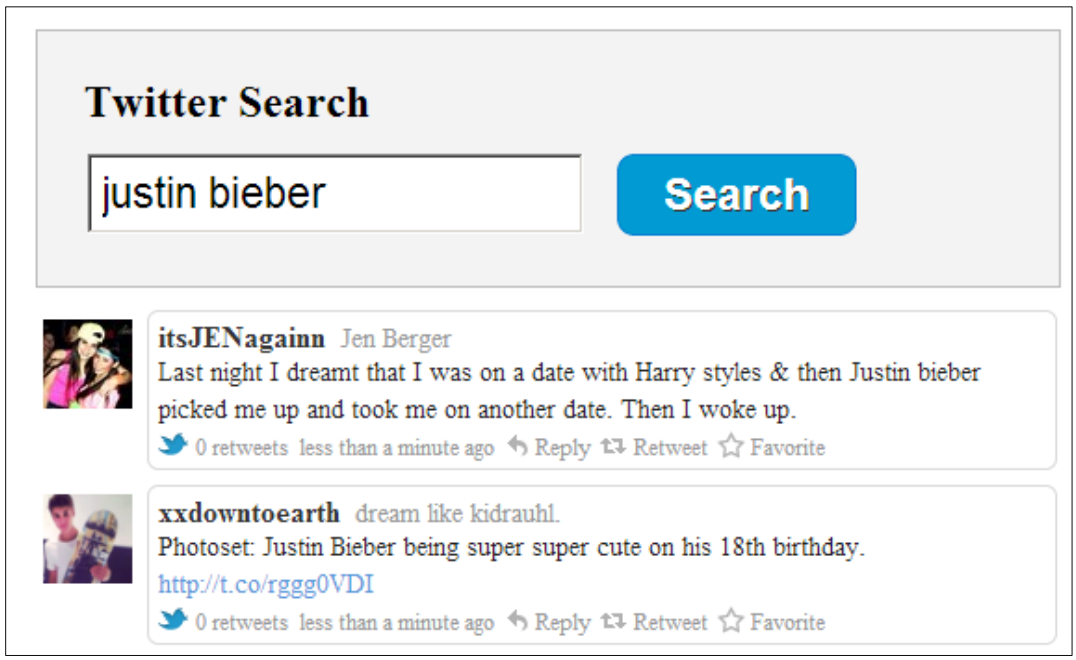
You now have the basic techniques you need to create a useful Twitter app with a Javascript user interface. The next step is fleshing it out to look like an app and display tweets correctly. Twitter has strict rules about displaying tweets on public web pages. Along with making this app look good, you'll also learn how to make it meet these display rules.

I've broken the app into separate files to make them easier to modify and reuse in your own future code. Here is a listing of all the pieces:

- search_client.html
- search_client.css
- search_client.js
- tweet_template.html
- tweet.css
- display_lib.php
- search_server.php

## Client HTML page

The complete app is in the ebook source code and starts with **search_client.html**. Running that page in a browser and entering a search term of **justin bieber** gives us a much prettier result than the last simple example.




*search_client.html*

```
<head>
  <title>Twitter Search</title>
  <link href="search_client.css" type="text/css" rel="stylesheet" />
  <link href="tweet.css" type="text/css" rel="stylesheet" />
  <script src="jquery.min.js"></script>
  <script src="search_client.js"></script>
</head>
<body>
  <div id="search_box">
      <h1>Twitter Search</h1>
      <input name="search_terms" autofocus="autofocus"/>
      <button id="search_button">Search</button>
  </div>
  <div id="search_results"></div>
</body>
```

## CSS files

There are two CSS files used by this app: **search_client.css** and **tweet.css**. Documenting all the CSS used to produce the app would take more space than I have in this ebook. Hopefully the CSS is self-explanatory. **Tweet.css** is used to format an individual tweet. I split these in two so

you can use tweet.css with any app that displays tweets. **Search_client.css** formats the main web page and input form.

*search_client.css*

```
body {
width:900px;
margin:24px auto;
}

p,ul,li,h1,h2 {
margin:0;
padding:0;
}

#search_box {
background:#f3f3f3;
padding:25px;
width:500px;
border:1px solid #bbb;
}

h1 {
font-size:24px;
margin-bottom:12px;
}

input[name=search_terms] {
font-size:23px;
padding:6px;
float:left;
margin-right:16px;
}

#search_button {
background-color:#019AD2;
-webkit-border-radius:9px;
border-radius:9px;
border:1px solid #057ED0;
color:#fff;
font-size:24px;
font-weight:700;
padding:6px 24px;
text-shadow:1px 1px 0 #333;
cursor:pointer;
}

.search_button:hover {
background-color:#378de5;
```

```
}

.search_button:active {
position:relative;
top:1px;
}

#search_results{
   display: inline-block;
  width:600px;
  margin-top:12px;
}
```

## Search Javascript file

All of the previous Javascript code was placed inside the HTML files. Another approach is to create a separate .js file and call this from the HTML file with the <script> tag. I prefer this method, because it makes the HTML easier to read and edit. **Search_client.js** assumes that the jQuery library has already been loaded by the HTML file.

*search_client.js*

```
// jQuery script for search request with server
jQuery(document).ready(function($) {

  // Run when Search button is clicked
  $('#search_button').click(function(){

      // Display a progress indicator
      $('#search_results').html('<img src="ajax_loader.gif"> Searching Twitter...');

      // Get the value of the input field
      // Encode it for use in a URL
      var search_value = encodeURIComponent($('input[name=search_terms]').val());

      // Send the search terms to the server in an Ajax request
      // This URL is for illustration only
      // You MUST change it to your server
      $.ajax({
          url: 'http://books.140dev.com/ebook_js/code/search_server.php?q=' + search_value,

          success: function(data){
              // Display the results
              $('#search_results').html(data);
          }
      })
  })
});
```

## Tweet HTML template

**Tweet_template.html** uses the only programming technique that we haven't seen before in this ebook. It is used to create a complete HTML version of the data received from the API for each tweet. Instead of assembling the HTML for a tweet with pieces of PHP code, it is easier to create a single block of HTML and then use the PHP **str_replace()** function to insert data where needed. Each piece of data is represented in this HTML file as a macro in the form of **[data].** For example, anywhere a tweet id is needed the macro **[tweet_id]** is used. This can be replaced by the actual id for each tweet. The insertion is done by the **search_server.php** script below.

*tweet_template.html*

```
<div class="tweet">
  <div class="tweet_left">
      <div class="tweet_image">
          <a href="https://twitter.com/[screen_name]">
              <img src="[profile_image_url]">
          </a>
      </div>
  </div>
  <div class="tweet_right">
      <div class="tweet_triangle"></div>
      <div class="tweet_row">
          <div class="tweet_username">
              <a href="https://twitter.com/[screen_name]" target="search">[screen_name]</a>
              <span class="tweet_name">[name]</span>
          </div>
      </div>
      <div class="tweet_row">
          <div class="tweet_text">[tweet_text]</div>
      </div>
      <div class="tweet_row">
          <div class="tweet_timestamp">
              <img class="stream_bluebird" src="/oauth/images/bird_16_blue.png">
              <span class="retweets">[retweet_count] retweets</span>
              <a href="http://twitter.com/[screen_name]/status/[tweet_id]"
                  target="search" class="tweet_time_link">[created_at]</a>
              <img src="reply.png" class="imageof_reply">
              <a href="http://twitter.com/intent/tweet?in_reply_to=[tweet_id]"
                  target="search">Reply</a>
              <img src="retweet.png" class="imageof_retweet">
              <a href="http://twitter.com/intent/retweet?tweet_id=[tweet_id]"
                  target="search">Retweet</a>
              <img src="favorite.png" class="imageof_favorite">
              <a href="http://twitter.com/intent/favorite?tweet_id=[tweet_id]"
                  target="search">Favorite</a>
          </div>
      </div>
```

```
  </div>
</div>
```

## Tweet CSS file

The combination of **tweet_template.html** and **tweet.css** can be used to create a nicely formatted tweet that meets all of Twitter's display guidelines. Splitting them into separate files like this makes it easy for a web designer to work with them without having to mess up any of the PHP code. I'm sure a web designer would look at this from the opposite direction and agree as well.

*tweet.css*

```
.tweet {
float:left;
padding-bottom:8px;
position:relative;
width:550px;
}

.tweet_left {
float:left;
width:28px;
margin:2px 4px 4px 2px;
}

.tweet_left img {
border:2px solid #fff;
margin-top:1px;
}

.tweet_left img:hover {
border:2px solid #C40001;
}

.tweet_right {
-moz-border-radius:6px;
-webkit-border-radius:6px;
border-radius:6px;
border:1px solid #DADADA;
float:right;
position:relative;
width:478px;
padding:2px 5px 4px;
}

.tweet_triangle {
background:url(twitter_triangle.gif) no-repeat scroll 100% 0 transparent;
height:13px;
```

```
left:-7px;
position:absolute;
top:10px;
width:7px;
}

.tweet_row {
line-height:15px;
position:relative;
}

.tweet_username {
color:#999;
font-size:14px;
margin-top:4px;
}

.tweet_username a {
color:#333;
font-size:16px;
margin-right:4px;
font-weight:700;
text-decoration:none;
}

.tweet_text {
font-size:14px;
color:#222;
line-height:20px;
word-wrap:break-word;
width:100%;
margin:0 0 3px;
padding:0;
}

.tweet_text a {
color:#5799DB;
text-decoration:none;
}

.tweet_timestamp,.tweet_timestamp a {
color:#999;
font-size:12px;
text-decoration:none;
float:left;
display:inline;
}

.tweet_timestamp {
```

```
margin-right:4px;
}

.tweet_timestamp .retweets {
float:left;
margin-right:6px;
}

.tweet_timestamp img {
float:left;
margin-left:4px;
margin-right:3px;
margin-top:-2px;
}

.tweet_timestamp .stream_bluebird {
margin-left:0;
margin-right:4px;
}

.tweet_timestamp .imageof_reply {
margin-right:2px;
}

.tweet_timestamp .imageof_retweet {
margin-top:-1px;
}

a:hover {
text-decoration:underline;
}
```

## Tweet display library

Two common conversions you need to perform when displaying tweets are linkifying the text and converting database formatted dates into Twitter format. This simple display library provides both of those functions. Don't ask me how the regex for linkifying actually works. I've been collecting linkifying code for years, and this is a compilation of many versions. It does work nicely though, even if I don't actually understand regex. ☺

*display_lib.php*

```
<?php

// Convert tweet text into linkified version
function linkify($text) {

  // Linkify URLs
   $text = ereg_replace("[[:alpha:]]+://[^<>[:space:]]+[[:alnum:]/]",
```

```
        "<a href=\"\\0\">\\0</a>", $text);

    // Linkify @mentions
    $text = preg_replace("/\B@(\w+(?!\/))\b/i",
        '<a href="https://twitter.com/\\1">@\\1</a>', $text);

    // Linkify #hashtags
    $text = preg_replace("/\B(?<![=\/])#([\w]+[a-z]+([0-9]+)?)/i",
        '<a href="https://twitter.com/#!/search/%23\\1">#\\1</a>', $text);

    return $text;
}

// Convert a tweet creation date into Twitter format
function twitter_time($time) {

  // Get the number of seconds elapsed since this date
  $delta = time() - strtotime($time);
  if ($delta < 60) {
    return 'less than a minute ago';
  } else if ($delta < 120) {
    return 'about a minute ago';
  } else if ($delta < (60 * 60)) {
    return floor($delta / 60) . ' minutes ago';
  } else if ($delta < (120 * 60)) {
    return 'about an hour ago';
  } else if ($delta < (24 * 60 * 60)) {
    return floor($delta / 3600) . ' hours ago';
  } else if ($delta < (48 * 60 * 60)) {
    return '1 day ago';
  } else {
    return number_format(floor($delta / 86400)) . ' days ago';
  }
}

?>
```

## Search server script

We've come to the end with **search_server.php**, which is the Twitter API proxy code for this search app. This version of the Ajax response uses the same techniques as **timeline_response.php** and **search_response.php** that we've seen earlier. The major enhancement is that it uses **tweet_template.html** and **display_lib.php** to assemble a complete display for each tweet.

*search_server.php*

```
<?php

// The search terms are passed in the q parameter
// search_server.php?q=[search terms]
if (!empty($_GET['q'])) {

  // Remove any hack attempts from input data
  $search_terms = htmlspecialchars($_GET['q']);

  // Get the application OAuth tokens
  require 'app_tokens.php';

  // Create an OAuth connection
  require 'tmhOAuth.php';
  $connection = new tmhOAuth(array(
   'consumer_key'    => $consumer_key,
   'consumer_secret' => $consumer_secret,
   'user_token'      => $user_token,
   'user_secret'     => $user_secret
  ));

  // Request the most recent 100 matching tweets
  $http_code = $connection->request('GET',$connection->url('1.1/search/tweets'),
        array('q' => $search_terms,
            'count' => 100,
            'lang' => 'en',
           'type' => 'recent'));

  // Search was successful
  if ($http_code == 200) {

      // Extract the tweets from the API response
      $response = json_decode($connection->response['response'],true);
      $tweet_data = $response['statuses'];

      // Load the template for tweet display
      $tweet_template= file_get_contents('tweet_template.html');

      // Load the library of tweet display functions
      require 'display_lib.php';

      // Create a stream of formatted tweets as HTML
      $tweet_stream = '';
      foreach($tweet_data as $tweet) {

          // Ignore any retweets
          if (isset($tweet['retweeted_status'])) {
              continue;
```

```
            }

            // Get a fresh copy of the tweet template
            $tweet_html = $tweet_template;

            // Insert this tweet into the html
            $tweet_html = str_replace('[screen_name]',
                $tweet['user']['screen_name'],$tweet_html);
            $tweet_html = str_replace('[name]',
                $tweet['user']['name'],$tweet_html);
            $tweet_html = str_replace('[profile_image_url]',
                $tweet['user']['profile_image_url'],$tweet_html);
            $tweet_html = str_replace('[tweet_id]',
                $tweet['id'],$tweet_html);
            $tweet_html = str_replace('[tweet_text]',
                linkify($tweet['text']),$tweet_html);
            $tweet_html = str_replace('[created_at]',
                twitter_time($tweet['created_at']),$tweet_html);
            $tweet_html = str_replace('[retweet_count]',
                $tweet['retweet_count'],$tweet_html);

            // Add the HTML for this tweet to the stream
            $tweet_stream .= $tweet_html;
        }

        // Pass the tweets HTML back to the Ajax request
        print $tweet_stream;

    // Handle errors from API request
    } else {
        if ($http_code == 429) {
            print 'Error: Twitter API rate limit reached';
        } else {
            print 'Error: Twitter was not able to process that search';
        }
    }

} else {
    print 'No search terms found';
}

?>
```

## Troubleshooting the ebook source code

I've helped a lot people get started with Twitter API programming, and the problems they experience fall into predictable patterns. Here are the most common errors and their solutions.

### Use your own OAuth tokens

The sample code includes a file called **app_tokens.php**. It has place holders for all the OAuth tokens. You need to create a Twitter app, generate a set of OAuth tokens, and place them into this file. If you are not familiar with this procedure, it is explained in detail in the Twitter OAuth ebook.

### Change the sample source code to point to your server

The sample code for the ebook has Ajax calls that use URLs for the 140dev.com server. You have to make sure that you have edited the code to use URLs for your server instead.

### Domain for server must be same as web page

The web page that runs your Javascript code must be in the same domain as the server script that response to that request. If the URL for your web page starts with http://mydomain.com, the server code URL must also start with http://mydomain.com.

This error can be very frustrating, because nothing happens when you run the Ajax request. The solution is to include an error parameter in your jQuery **ajax()** function. This can be as simple as an alert box that warns you. It can be removed once your code is debugged and functioning.

### Incorrect identifier for HTML elements

jQuery works with HTML elements through what it calls identifiers. If you have a span with an id of **mytext** (entered as **),** you would refer to it in a jQuery statement with the identifier of **$("#mytext")**. If this code fails, make sure the identifier is written correctly. Ids start with # and classes start with a period.

### Use Firebug

Once you get serious about jQuery programming, you should install Firebug in your browser. It is invaluable for seeing all the HTML elements in a page and watching Ajax requests as they happen.

### Run server code directly

Debugging Ajax code through the requesting page can be frustrating, because you can't really see what is going on without inserting a bunch of alerts into the Javascript. There is a simpler

approach. Since the server response script is run with a URL, you can run that URL within your browser. Just make sure you include the proper URL arguments. For example, **server_response.php** can be called with this URL in a browser. Of course, you need to change the URL to your server:

http://books.140dev.com/ebook_js/code/search_response.php?q=Justin+bieber

## Google Group for this ebook

If all else fails, you can post a question on the Google Group set up for questions related to this ebook.