**Exercise02\_02\_01 – Step 1**

This exercise will consist of coding the parts of an OAuth login request in a modular fashion.



1. Create a new folder for the exercise called Exercise02\_02\_01. Copy all of the files into it from the previous Exercise02\_01\_01.
2. First we will install any external third-party modules that we will need with NPM. Since we will be storing client state information in ***cookies***, we will want an external package called ***cookie-parser*** to give us some help with that. It will provide us with some ***middleware*** we can use. Middleware is designed to be executed between an ***endpoint*** request and response, therefore, in the middle.  
     
   We will also be using the ***OAuth*** package for our authentication and authorization. Enter the following commands at a terminal to install the packages and place the dependencies in package.json:  
   ***npm install cookie-parser --save  
   npm install oauth –save***Check **package.json** to make sure the dependencies are correct.
3. Now let’s create a ***config.json*** file that will hold, as its ***properties***, various configuration items, URLs, tokens, and secrets. Some of the properties will be set by us, some will be gotten from our Twitter app that we created in the last exercise, and one we will deal with later. Construct the JSON file as follows:  
   ***{  
    "port": 8080,  
    "request\_token\_url":   
    "https://api.twitter.com/oauth/request\_token",  
    "access\_token\_url":   
    "https://api.twitter.com/oauth/access\_token",  
    "authorize\_url": "https://api.twitter.com/oauth/authorize",  
    "consumer\_key": "f2sHurvY8Asq2bbXYwQDS05vE",  
    "consumer\_secret":   
    "BnsqYkR8sLZahUizwldIXz3qCN6Jso11gFPlSls3XNhTr1N1j6",  
    "oauth\_version": "1.0",  
    "oauth\_signature": "HMAC-SHA1",  
    "oauth\_callback": “http://www.example.com/callback”  
   }***
4. Modify ***index.js*** to give the config file a test. Replace the variable ***port*** declaration with a ***require()*** call, and use config.port in the ***listen()*** method:  
   var express = require('express');  
   ***var config = require('./config.json');***var app = express();  
     
   app.listen(***config.port***, function() {  
    console.log("Server listening on localhost:%s", ***config.port***);   
   });  
   Run your server to make sure that everything is working and the server is listening on the correct port, then terminate it.
5. In ***index.js*** we will need more support with the application. We require() the ***core*** module ***url***, to help us with URL parsing. We also mount ***cookie-parser*** middleware, to use in some of our ***endpoints***, or ***routes***. Look at the syntax of the ***app.use()*** call that mounts it, and also look at the ExpressJS documentation for app.use():  
   ***var url = require('url');***var express = require('express');  
    ***app.use(require('cookie-parser')());***Run your server to make sure that everything is still working, then terminate it.

**Exercise02\_02\_01 – Step 2**



1. Let’s open ***index.js*** and define the routes we will use for the first part of the authentication process. The first route will eventually generate a request token. If successful, it will redirect the user to the Twitter authorization page. Let’s start by scaffolding that function to check our control flow by modifying the ***app.get()*** function as follows:  
   ***app.get("/auth/twitter", function(req, res) {  
    res.send("This route will get a request token and   
    redirect the user to the twitter sign in.");  
   });***  
   Run the server. then bring up a browser and test the endpoint with a ***localhost:port/auth/twitter*** URL.
2. Let’s start scaffold another function that will eventually handle Oauth as a callback:  
   ***app.get("/auth/callback", function(req, res) {  
    res.send("This route will handle OAuth as a callback.");***  
   ***});***Run the server. then bring up a browser and test the endpoint with a ***localhost:port/auth/callback*** URL.
3. We are going to ***modularize*** this application, so we are going to want all of the authorization functionality to reside in another module. Let’s start to modify our code for this by naming our first ***anonymous*** callbacks as follows:  
   app.get("/auth/twitter", ***function redirectToTwitterLoginPage(req,   
    res)***   
   Run the server and test the ***/auth/twitter*** endpoint again in the browser.
4. Now let’s start building our authentication module, by creating a file called ***authenticator.js***. First let’s get access to some of the things that we are going to need, like ***Oauth*** and our ***config.json*** file, with some ***require()*** statements. Notice that we are just getting a single Oauth ***class*** from the Oauth package:  
   ***var OAuth = require('oauth').OAuth;  
   var config = require('./config.json');***
5. We are going to need this module in ***index.js***, so let’s open it and ***require()*** in the authenticator right now. Then we will be set up to do some testing when we run our server:  
   var express = require('express');  
   ***var authenticator = require('./authenticator.js');***Run the server and test the ***/auth/twitter*** endpoint again in the browser.
6. We now go back to ***authenticator.js***. We are going to ***construct*** our Oauth object, and initialize it with properties from our ***config*** file:  
   ***var oauth = new OAuth(  
    config.request\_token\_url,  
    config.access\_token\_url,  
    config.consumer\_key,  
    config.consumer\_secret,  
    config.oauth\_version,  
    config.oauth\_callback,  
    config.oauth\_signature  
   );***Run the server and test the ***/auth/twitter*** endpoint again in the browser. So far, so good.
7. Let’s set up a data structure to hold our Twitter credentials as we get them. We will use a ***JSON object*** for that. build it as follows:  
   ***var twitterCredentials = {  
    oauth\_token: "",  
    oauth\_token\_secret: ""  
   }***
8. We need to set up our ***module.exports***. We will set it equal to a ***JSON*** object that will hold the objects that we want to make available in other modules that ***require()*** the authenticator. Let’s start by setting up our ***/auth/twitter*** callback as the first property to export. Copy the callback function definition from ***index.js*** and set it up with a new test string, JSON style, as follows:  
   ***module.exports = {  
    redirectToTwitterLoginPage: function(req, res) {  
    res.send("Using our authenticator module.");  
    }  
   };***
9. We now need to modify ***index.js*** to make use of our exported callback. Modify the ***/auth/twitter*** route to use the property that defines the callback from ***authenticator.js***:  
   ***app.get("/auth/twitter",   
    authenticator.redirectToTwitterLoginPage);***  
   Give this a test. Very modular, nice and clean.
10. We can start building the functionality into ***authenticator.js***. First we will use Oauth to get us a ***request*** ***token*** and a ***token*** ***secret***. We will make use of a method that comes with the OAuth object we constructed, and overwrite our debug ***res.send()***:  
     redirectToTwitterLoginPage: function(req, res) {  
     ***oauth.getOAuthRequestToken(function(error, oauth\_token,   
     oauth\_token\_secret, results) {  
     if (error) {  
     console.log(error);  
     res.send("Authentication failed!");  
     }  
     else {  
     res.send("Authentication successful!");  
     }  
     });*** }  
    Run the server and test the ***/auth/twitter*** endpoint again in the browser.
11. We should now have our request token and request secret. We will store these for future use in our JSON object. We will override our ***res.send()*** debug in the ***else*** statement:  
     else {  
     ***twitterCredentials.oauth\_token = oauth\_token;  
     twitterCredentials.oauth\_token\_secret =   
     oauth\_token\_secret;  
     res.send("Credentials stored");***Run the server and test the ***/auth/twitter*** endpoint again in the browser.
12. Now let’s issue our response, which will be a redirection to Twitter so that the user can log in. We will override our ***res.send()*** debug again:  
     twitterCredentials.oauth\_token\_secret =   
     oauth\_token\_secret;  
     ***res.redirect(config.authorize\_url + '?oauth\_token=' +   
     oauth\_token);***Give this a test and we should get a login page. However, if we try to authorize the page, we have a problem. We have given Twitter a bogus callback. This is because we have no place for Twitter to call us back at, we are not hosted on a real Web domain.

**Exercise02\_02\_01 – Step 3**



1. We will use various applications to expose our local machine over a secure ***tunnel*** to the public internet. The reason for multiple choices is that at any time one or more of the applications may go unstable.
   1. The first, and most stable application is called ***Serveo***. This application is run from a terminal, using ssh. Run the following command, using the port configuration that follows, where ***xxx*** is your initials:  
      ***ssh –o ServerAliveInterval=3600   
       –R xxxtunnel:80:localhost:8080 serveo.net***  
      It should run in the background, and provide an HTTPS forwarding address. Shut it down with a ^C.
   2. The second application is ***Local Tunnel***. It is not quite as stable as Serveo, it needs to be rerun at times, but it will work fine for us if the need arises. This package must be installed with NPM as follows:  
      ***npm install –g localtunnel***
   3. Now let’s customize our own tunnel so it will give us the same callback address every time, where ***xxx*** is your initials. From a terminal, use the following command:  
      ***lt -p 8080 -h "https://localtunnel.me" -s "xxxtunnel"***  
      For testing on local machines this process will have to be followed every time we start a new server session.
   4. The third application is ***Ngrok***. It is very stable, but sometimes our firewalls block it. It should be pre-installed on your computer. From a terminal, use the following command:  
      ***ngrok http 8080***
2. Copy the HTTPS forwarding address, for example:  
   ***https://mjbtunnel.serveo.net  
   https://mjbtunnel.localtunnel.me***
3. We need to place our callback into our Twitter app in developer.twitter.com. Edit the application details and copy the forwarding address with the ***request\_token*** resource as follows into ***Callback URL***:  
   ***https://mjbtunnel.localtunnel.me/request\_token***
4. Open your ***config.json*** file and paste that address into the oauth\_callback property string, adding the last ***/request\_token*** element as follows:  
    ***"oauth\_callback":   
    https://mjbtunnel.localtunne  
   or  
    https://mjbtunnel.serveo.net l.me/request\_token***Run the server and test the ***/auth/twitter*** endpoint again in the browser. This time we should get a callback. It will give us a ***Cannot GET /request\_token*** error message because we have not yet built the authentication part of the code.
5. We will need to work on the second route to ***authenticate*** the user. It will have to use the Twitter ***callback url*** as the path for its endpoint. To set the stage for this, let’s use our ***url*** ***module*** to help us with the path, and to give us confirmation that we are using the current ngrok callback in ***config.json***. Modify the ***app.listen()*** code to show us the call back:  
   app.listen(config.port, function() {  
    console.log("Server listening on localhost:%s", config.port);  
    ***console.log("OAuth callback: " +   
    url.parse(config.oauth\_callback).hostname +   
    url.parse(config.oauth\_callback).path);***});  
   Compare the ***console.log()*** output with the forwarding path in the tunnel terminal. They should always match and should be checked at every coding session.
6. Now we can modify the second app.get() route to use the Oauth callback resource:  
   app.get(***url.parse(config.oauth\_callback).path***, function(req, res) {  
    res.send("This route will handle OAuth as a callback.");  
   });  
   Run the server and test the ***/auth/twitter*** endpoint again in the browser. Click the ***Authorize app*** button. This time the authentication endpoint should be picked up and we should get our debug message on the Web page.
7. We need to build out the **callback** from the route. Let’s build it so that it calls a function that we will build into our ***authenticator.js*** module. We will overwrite our debug ***res.send()*** with that function, which will be named ***authenticate()***.We will pass into it the endpoint ***request*** and ***response***. We will also give it a ***callback*** to execute when the function finishes:  
   app.get(url.parse(config.oauth\_callback).path, function(req, res) {  
    ***authenticator.authenticate(req, res, function(err) {  
    if (err) {  
    console.log(err);  
    res.sendStatus(401);  
    }  
    else {  
    res.send('Authentication successful!');  
    }  
    });***});  
   We can test this for syntax, but it will crash when we authorize the app. This is because we have not yet built the ***authenticate()*** function.
8. Let’s go over to ***authenticator.js*** and build our ***authenticate*** function into our ***module.exports***. we will start by scaffolding it so that we can test our error and success code back in ***index.js***. Let’s build it as follows to simulate and test for the error condition:  
    }***,  
    authenticate: function(req, res, callback) {  
    return callback("Testing authenticate failure.");  
    }***  
   Give this a test and authorize the app. We should get back our console.log() error and our 401 Unauthorized in the browser.
9. Now we will modify it to simulate and test for the success condition:  
    ***return callback();***Give this a test and authorize the app. We should get back our ***Authorization successful*** in the browser.
10. Let’s now start to build our ***authenticate()*** function for real. First we will test to see if we have our ***oauth\_token*** and ***oauth secret***. We previously stored these in a JSON object, so we will check them. We also need to see if we got an ***oauth\_verifier***. That holds our temporary credentials, which we should have gotten this back from Oauth when we redirected to the authorization URL. If we don’t have all three, we will return an ***error***. Let’s overwrite our return debug as follows:  
     authenticate: function(req, res, callback) {  
     ***if (!(twitterCredentials.oauth\_token &&   
     twitterCredentials.oauth\_token\_secret &&   
     req.query.oauth\_verifier)) {  
     return callback("Request does not have all required keys.");  
     }  
     twitterCredentials.oauth\_token = "";  
     twitterCredentials.oauth\_token\_secret = "";  
     return callback();*** }  
    Give this a test and authorize the app. We should get back our ***Authorization successful*** in the browser, and we are done with this phase. In the next phase. We will exchange our temporary credentials for an access token.

**Exercise02\_02\_01 – Step 4**



1. Our next task will be to exchange our temporary credential for an ***access*** ***token***. We will go to ***authenticator.js*** and build more into our ***authenticate*** function. We will start by calling a ***method*** of our Oauth object to do the exchange. Let’s build it as follows, to simulate and test for the error condition, by leaving our temporary credentials empty. Build the following code directly after we set them to empty, and overwriting the ***callback()*** debug call:  
    twitterCredentials.oauth\_token = "";  
    twitterCredentials.oauth\_token\_secret = "";  
    ***oauth.getOAuthAccessToken(twitterCredentials.oauth\_token,   
    twitterCredentials.oauth\_token\_secret,   
    req.query.oauth\_verifier,   
    function(error, oauth\_access\_token,   
    oauth\_access\_token\_secret, results) {  
    if (error) {  
    return callback(error);  
    }  
    });***Give this a test and authorize the app. We should get back our ***401 Unauthorized*** in the browser. Now ***remove*** the code that empties twitterCredentials and re-test.
2. We will now verify the user’s identity. To do that we will use one of Twitter’s REST API endpoints. Google ***Twitter REST APIs*** and go to the Docs API Reference Index. There is a list of Twitter’s API endpoints. Click on ***GET account/verify credentials***. This supplies us with information such as the ***endpoint*** ***URL***, request ***parameters***, and an example ***response***. In this case, we can see that the response is going to be a fair amount of JSON, let’s examine that.
3. Now let’s extend our ***authenticate()*** function to verify the user’s credentials. And to obtain a ***Twitter*** ***ID***. Let’s use an ***oauth.get()*** to the API, passing our new ***credentials***, and setting up a ***callback***. We will handle the ***error*** case first. If we are successful, we will ***parse*** our JSON response and console.log() it to the node terminal for debug and examination:  
    if (error) {  
    return callback(error);  
    }  
    ***oauth.get(  
    'https://api.twitter.com/1.1/account/verify\_credentials.json',   
    oauth\_access\_token, oauth\_access\_token\_secret,   
    function(error, data) {  
    if (error) {  
    console.log(error);  
    return callback(error);  
    }  
    data = JSON.parse(data);  
    console.log(data);  
    callback();  
    });***Give this a test and let’s take a look at our JSON data.
4. Let’s extend our the JSON object that holds our credentials to have some new properties for our new credentials and twitter ID:  
   var twitterCredentials = {  
    oauth\_token: "",  
    oauth\_token\_secret: "",  
    ***access\_token: "",  
    access\_token\_secret: "",  
    twitter\_id: ""***
5. We can now store our information for later use:  
    data = JSON.parse(data);  
    ***twitterCredentials.access\_token = oauth\_access\_token;  
    twitterCredentials.access\_token\_secret =   
    oauth\_access\_token\_secret;  
    twitterCredentials.twitter\_id = data.id\_str;  
    console.log(twitterCredentials);*** callback();  
   Let’s give this a test and examine our saved JSON credentials. If everything is okay, we can remove. The console.log() debug.

**Exercise02\_02\_01 – Step 5**



1. As we have seen from documentation, Twitter’s RESTful APIs only make use of the ***GET*** and ***POST*** HTTP verbs. Let’s first build a generic call for each verb that we can use for any of the endpoints that we want to build. Go to ***authenticator.js*** and build a ***get()*** and ***post()*** function for export as follows:  
   module.exports = {  
    ***get: function(url, access\_token, oauth\_access\_token\_secret,   
    callback) {  
    oauth.get.call(oauth, url, access\_token,   
    oauth\_access\_token\_secret, callback);  
    },  
    post: function(url, access\_token, oauth\_access\_token\_secret,   
    body, callback) {  
    oauth.post.call(oauth, url, access\_token,   
    oauth\_access\_token\_secret, body, callback);  
    },***Give this a test and authorize the app. We should not have any problems.
2. We are going to need our Twitter ***credentials*** for all of our Twitter API calls. Let’s export one more function from authenticator.js to give them to us on demand. We can build this right after our post function:  
    ***getCredentials: function() {  
    return twitterCredentials;  
    },***
3. For our first task, let’s use the RESTful APIs to send ***Tweets***. This functionality is implemented with the ***POST statuses/update*** URL. Scroll to that in the ***API Reference*** and let’s look at the documentation. We can copy off the URL from here when we need it. We also see that there is a required parameter named ***status***, which can hold up to 140 characters. This forms the text of our Tweets.
4. Let’s go to ***index.js*** and add a new ***route*** for Tweets, which we will give a URL of ***/tweet***. First we will get our credentials to see if we can proceed:  
   ***app.get('/tweet', function(req, res) {  
    credentials = authenticator.getCredentials();  
    if (!credentials.access\_token ||   
    !credentials.access\_token\_secret) {  
    return res.sendStatus(401);  
    }  
    res.sendStatus(200);  
   });***Let’s run the server and then do a couple of tests. First let’s go directly to the ***/tweet*** route in the browser. Because we have not done an authorization with Twitter, we should get a status ***401 Unauthorized*** message. Now let’s go to our ***/auth/twitter*** route and authorize. Then go back to ***/tweet*** and we should get a status ***200 OK*** message.
5. Let’s build out the actual Tweet code. We will need to do an ***authenticator.post()*** call to use the correct URL. We can copy that ***URL*** from the Twitter API documentation and store it in a ***var***. We will build a simple ***status*** that will be the body of our post() call. That will become the text of our tweet. We can build as follows, overwriting our debug ***res.sendStatus(200)*** call:  
    ***var url = 'https://api.twitter.com/1.1/statuses/update.json';  
    authenticator.post(url, credentials.access\_token,   
    credentials.access\_token\_secret,   
    {  
    status: "Hello there Twitter RESTful API"  
    },  
    function (error, data) {  
    if (error) {  
    return res.status(400).send(error);  
    }  
    res.send('Tweet successful!');  
    });***  
   Run the server, send the browser to ***/auth/twitter***, then to **/tweet**. Open up your Twitter account and see if the Tweet got there.

**Exercise02\_02\_01 – Step 6**



1. Now let’s use the RESTful APIs to search for ***Tweets***. This functionality is implemented with the ***GET search/tweets*** URL. Scroll to that in the ***API Reference*** and let’s look at the documentation. We have access to the URL, we see that there is a required ***q*** ***parameter***, and we can see that our response is going to be JSON.
2. The required ***q parameter*** documentation tells us that we have to build a search query to use with this API call. Let’s get the support we need for this, and we will learn how we use it when we build our route:  
   var express = require('express');  
   ***var querystring = require('querystring');***
3. Let’s go to ***index.js*** and add a new route to search for Tweets, which we will give a URL of ***/search***. Let’s copy the ***/tweet*** route and paste it below itself. First we will still need to get our credentials to see if we can proceed. Next let’s copy the Resource URL from the Twitter documentation into the url variable:  
   ***app.get('/search', function (req, res) {  
    var credentials = authenticator.getCredentials();  
    if (!credentials.access\_token || !credentials.access\_token\_secret)   
   {  
    return res.sendStatus(401);  
    }  
    var url = 'https://api.twitter.com/1.1/search/tweets.json';***
4. In this case we are going to do a ***GET*** call to the API. We will need to modify our code to make that call. GET requests do not take a ***body*** parameter, so we will remove the ***JSON*** holding the ***status*** property. We will also change our ***res.send()*** to display the ***data*** instead of a message as follows:  
    ***authenticator.get(url, credentials.access\_token,   
    credentials.access\_token\_secret,   
    function (error, data) {  
    if (error) {  
    return res.status(400).send(error);  
    }  
    res.send(data);  
    });***
5. Now let’s build our ***query*** ***string***. When we have that, we will use it by adding it to the end of our ***url*** as a name/value pair as follows:  
    var url = 'https://api.twitter.com/1.1/search/tweets.json';  
    ***var query = querystring.stringify({ q: ‘BMW' });  
    url += "?" + query;***Run the server, send the browser to ***/auth/twitter***, then to **/search**. There should be a tremendous amount of results. Try switching around values for the query string.

**Exercise02\_02\_01 – Step 7**



1. Now let’s use the RESTful APIs to search for ***Friends***, or those that the user is following. This functionality is implemented with the ***GET friends/list*** URL. Scroll to that in the ***API Reference*** and let’s look at the documentation. We have access to the URL, we see that it returns a ***cursored*** collection, which means that the resource sends friends in groups of 20 in this case. We can see that our responses are going to be JSON.
2. Let’s go to ***index.js*** and add a new route to search for Friends, which we will give a URL of ***/friends***. Let’s copy the ***/search*** route and paste it below itself. This will work because ***cursors*** are implemented in much the same way as ***query strings***. First we will still need to get our credentials to see if we can proceed. Next let’s copy the Resource URL from the Twitter documentation into the url variable:  
   ***app.get('/friends', function (req, res) {  
    var credentials = authenticator.getCredentials();  
    if (!credentials.access\_token || !credentials.access\_token\_secret) {  
    return res.sendStatus(401);  
    }  
    var url = 'https://api.twitter.com/1.1/friends/list.json';  
    authenticator.get(url, credentials.access\_token,   
    credentials.access\_token\_secret,   
    function (error, data) {  
    if (error) {  
    return res.status(400).send(error);  
    }  
    res.send(data);  
    });***
3. Now let’s build our ***query*** ***string*** for handling our cursor. The cursors are returned with the last JSON data item. If we have that, we will use it by adding it to the end of our ***url*** as a name/value pair. The first time through, it will not be part of our URL as follows:  
    var url = 'https://api.twitter.com/1.1/friends/list.json';  
    ***if (req.query.cursor) {  
    url += "?" + querystring.stringify({ cursor: req.query.cursor });  
    }***Run the server, send the browser to ***/auth/twitter***, then to **/friends**. We should get our first 20 friends.
4. Scroll down to the bottom and as far right as possible on the last line of the results. There should be a name/value pair with name ***next\_cursor\_str***. Copy the value and it to the end or the browser URL as a query parameter, without the quotes, as follows:  
   ***http://localhost:8080/friends?cursor=1553263176318567747***  
   Use enter on that route and we should get the next, or last, page of the friends list.

**Exercise02\_02\_01 – Step 8**



1. Now let’s use the RESTful APIs to search for ***All*** ***Friends***. To do this in a completely automated fashion with cursored collections, we will make use of the ***async.js*** module. Let’s install it and make sure it is a dependency in package.json. Then we will require() it in **index.js**:  
   ***npm install async –save***var querystring = require('querystring');  
   ***var async = require('async');***Make sure to check the package.json file.
2. To search for All Friends, we will use a multiple stage process. The first step will be to get all of the friends Ids with the ***GET friends/ids*** URL. Scroll to that in the ***API Reference*** and let’s look at the documentation. We have access to the URL, we see that it returns a ***cursored*** collection, and this resource sends ids in groups of 5000. If there are more than 5000 ids, we will be hitting this resource more than once.  
     
   The next step will be to use the ***GET users/lookup*** API. This resource takes a list of up to 100 ids and returns an array of users. If there are more than 100 friends from the first API, we will be hitting this resource more than once.
3. Let’s go to ***index.js*** and add a new route to search for All Friends, which we will give a URL of ***/allfriends***. Let’s build this route from scratch, as it is of a very different structure than our previous ones:  
   ***app.get('/allfriends', function(req, res) {  
    var credentials = authenticator.getCredentials();  
   });***
4. Now we can scaffold out our waterfall. It will consist of two tasks, or functions, in its array. The first function will be responsible for getting all of the friends’ Ids. The second will take care of passing the IDs to the lookup resource. Add a sendStatus(200) call for debug:  
   app.get('/allfriends', function(req, res) {  
    ***async.waterfall([  
    // get friends ids  
    function(callback) {  
      
    },  
    // lookup friends data  
    function(ids, callback) {  
      
    }  
    ]);  
    res.sendStatus(200);***});Run the server, send the browser to **/allfriends**. We should get our ***200 OK*** status.
5. Let’s set up a little bit of support and debug in the first waterfall task as follows:  
    async.waterfall([  
    // get friends ids  
    function(callback) {  
    ***var cursor = -1;  
    var ids = [];  
    console.log('ids.length: ' + ids.length);***
6. Now let’s carefully build the first ***waterfall*** function with an ***asynch.whilst()*** method. This function will start with the cursor set to -1, so at the beginning of the ***friends/ids*** collection. The first parameter to the ***whilst()*** is a ***function*** that is the loop condition. It will loop through the next function, 5000 ids at a time through the collection. It concatenates them into the ***ids*** array. Each loop returns with its ***callback*** to check the whilst() condition., until the cursor reaches 0, the end of the collection. It then executes the final function, and if no error, will pass the ids array to the next function in the waterfall. Remove the ***res.sendStatus(200)*** debug. There is some ***console.log(ids)*** debug. Be careful, and do not get confused by callback hell:  
    console.log('ids.length: ' + ids.length);  
    ***async.whilst(function() {  
    return cursor != 0;  
    }, function(callback) {  
    var url = 'https://api.twitter.com/1.1/friends/ids.json';  
    url += "?" + querystring.stringify({ user\_id:   
    credentials.twitter\_id, cursor: cursor});  
    authenticator.get(url, credentials.access\_token,   
    credentials.access\_token\_secret,   
    function(error, data) {  
    if (error) {  
    return res.status(400).send(error);  
    }  
    data = JSON.parse(data);  
    cursor = data.next\_cursor\_str;  
    ids = ids.concat(data.ids);  
    callback();  
    });  
    }, function(error) {  
    if (error) {  
    return res.status(500).send(error);  
    }  
    console.log(ids);  
    callback(null, ids);  
    });***Run the server, send the browser to **/allfriends**. We should get our ids from the ***console.log()***. If everything is okay, remove the console.log debug.
7. Now let’s carefully build the second ***waterfall*** function with an ***asynch.times()*** method. This function will start by defining two functions. The first will return the next hundred ids. The second will compute the number of requests needed top process all of the ids:  
    function(ids, callback) {  
    ***var getHundredIds = function(i) {  
    return ids.slice(100\*i, Math.min(ids.length, 100\*(i+1)));  
    }  
    var requestsNeeded = Math.ceil(ids.length/100);  
    }***
8. We now use the ***asynch.times()*** to call the next function the correct number of times. It makes the API requests and passes the data to the async callback, which is called ***next()***. Once it has run the proper number of times, it then executes the final function. It passes in the results of all of the calls as the second argument. The friends will actually be passed in as an array of arrays. The function reduces this into a single array, then sorts the friends alphabetically. Be careful, and do not get confused by callback hell:  
    var requestsNeeded = Math.ceil(ids.length/100);  
    ***async.times(requestsNeeded, function(n, next) {  
    var url = 'https://api.twitter.com/1.1/users/lookup.json';  
    url += "?" + querystring.stringify({ user\_id:   
    getHundredIds(n).join(',')});  
    authenticator.get(url, credentials.access\_token,   
    credentials.access\_token\_secret,   
    function(error, data) {  
    if (error) {  
    return res.status(400).send(error);  
    }  
    var friends = JSON.parse(data);  
    next(null, friends);  
    });  
    },  
    function(err, friends) {  
    friends = friends.reduce(function(previousValue,   
    currentValue, currentIndex, array) {  
    return previousValue.concat(currentValue);  
    }, []);  
    friends.sort(function(a, b) {  
    return   
    a.name.toLowerCase().  
    localeCompare(b.name.toLowerCase())  
    });  
    res.send(friends);  
    console.log('ids.length: ' + ids.length);  
    });  
    }***Run the server, send the browser to **/allfriends**. We should get our friends on the browser screen We can verify the number of friends from the ***console.log()***.