er-Side JavaScript with Node.js

0 with 50 comments

- **Technology:** JavaScript
- **Difficulty:** Intermediate-Advanced

Share

Download Source Files

Node.js is all the buzz at the moment, and makes creating high performance, real-time web applications easy. It allows JavaScript to be used end to end, both on the server and on the client. This tutorial will walk you through the installation of Node and your first "Hello World" program, to building a scalable streaming Twitter server.

What is Node.js?

JavaScript has traditionally only run in the web browser, but recently there has been considerable interest in bringing it to the server side as well, thanks to the CommonJS project. Other server-side JavaScript environments include Jaxer and Narwhal. However, Node.js is a bit different from these solutions, because it is event-based rather than thread based. Web servers like Apache that are used to serve PHP and other CGI scripts are thread based because they spawn a system thread for every incoming request. While this is fine for many applications, the thread based model does not scale well with many long-lived connections like you would need in order to serve real-time applications like Friendfeed or Google Wave.

> "Every I/O operation in Node.js is asynchronous…"

Node.js, uses an event loop instead of threads, and is able to scale to millions of concurrent connections. It takes advantage of the fact that servers spend most of their time waiting for I/O operations, like reading a file from a hard drive, accessing an external web service or waiting for a file to finish being uploaded, because these operations are much slower than in memory operations. Every I/O operation in Node.js is asynchronous, meaning that the server can continue to process incoming requests while the I/O operation is taking place. JavaScript is extremely well suited to event-based programming because it has anonymous functions and closures which make defining inline callbacks a cinch, and JavaScript developers already know how to program in this way. This event-based model makes Node.js very fast, and makes scaling real-time applications very easy.

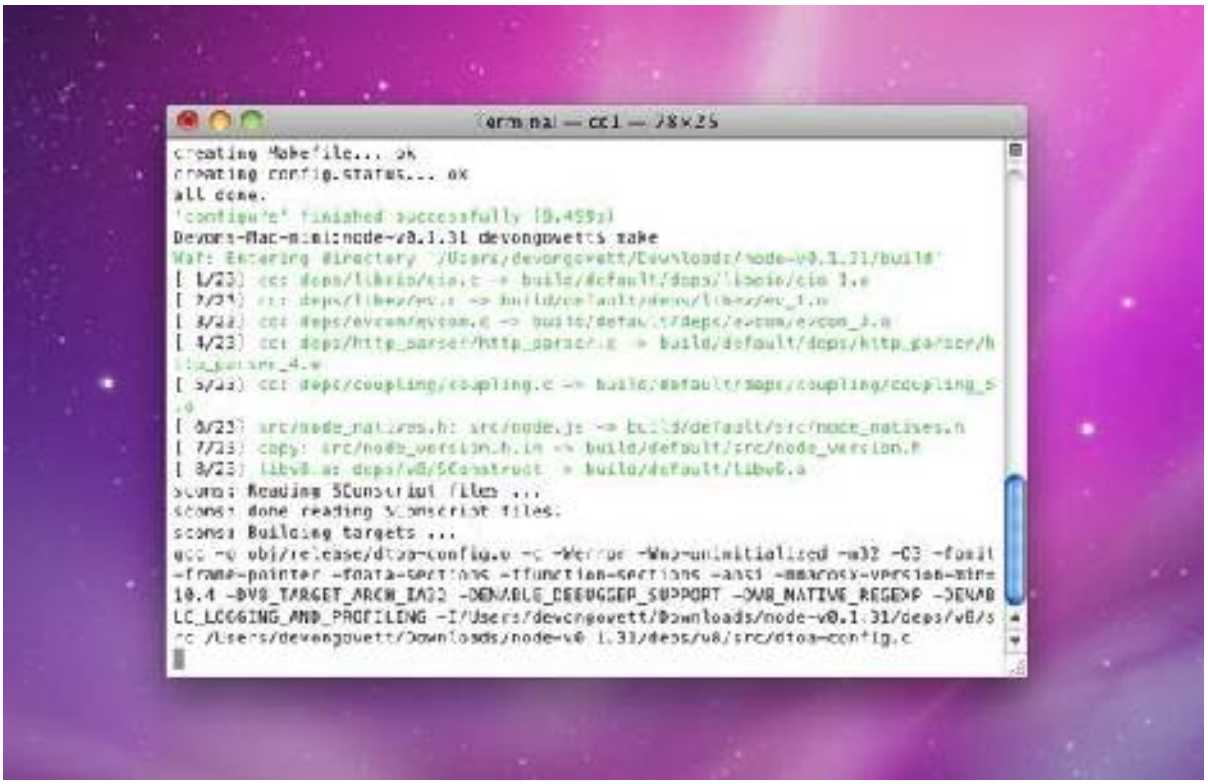
Step 1 Installation

Node.js runs on Unix based systems, such as Mac OS X, Linux, and FreeBSD. Unfortunately, Windows is not yet supported, so if you are a Windows user, you can install it on Ubuntu Linux using Virtualbox. To do so, follow this tutorial. You will need to use the terminal to install and run Node.js.

1. Download the latest release of Node.js from nodejs.org (the latest version at the time of this writing is 0.1.31) and unzip it.
2. Open the terminal, and run the following commands.
    1. cd /path/to/nodejs
    2. make
    3. sudo make install
    4.

```
cd /path/to/nodejs
make
sudo make install
```

A lot of messages will be outputted to the terminal as Node.js is compiled and installed.



Step 2 Hello World!

Every new technology starts with a “Hello World!” tutorial, so we will create a simple HTTP server that serves up that message. First, however, you have to understand the Node.js module system. In Node, functionality is encapsulated in modules which must be loaded in order to be used. There are many modules listed in the Node.js documentation. You load these modules by using the `require` function like so:

view plaincopy to clipboardprint?

1. var sys = require("sys");

```
var sys = require("sys");
```

This loads the sys module, which contains functions for dealing with system level tasks like printing output to the terminal. To use a function in a module, you call it on the variable that you stored the module in, in our case `sys`.

view plaincopy to clipboardprint?

1. sys.puts("Hello World!");

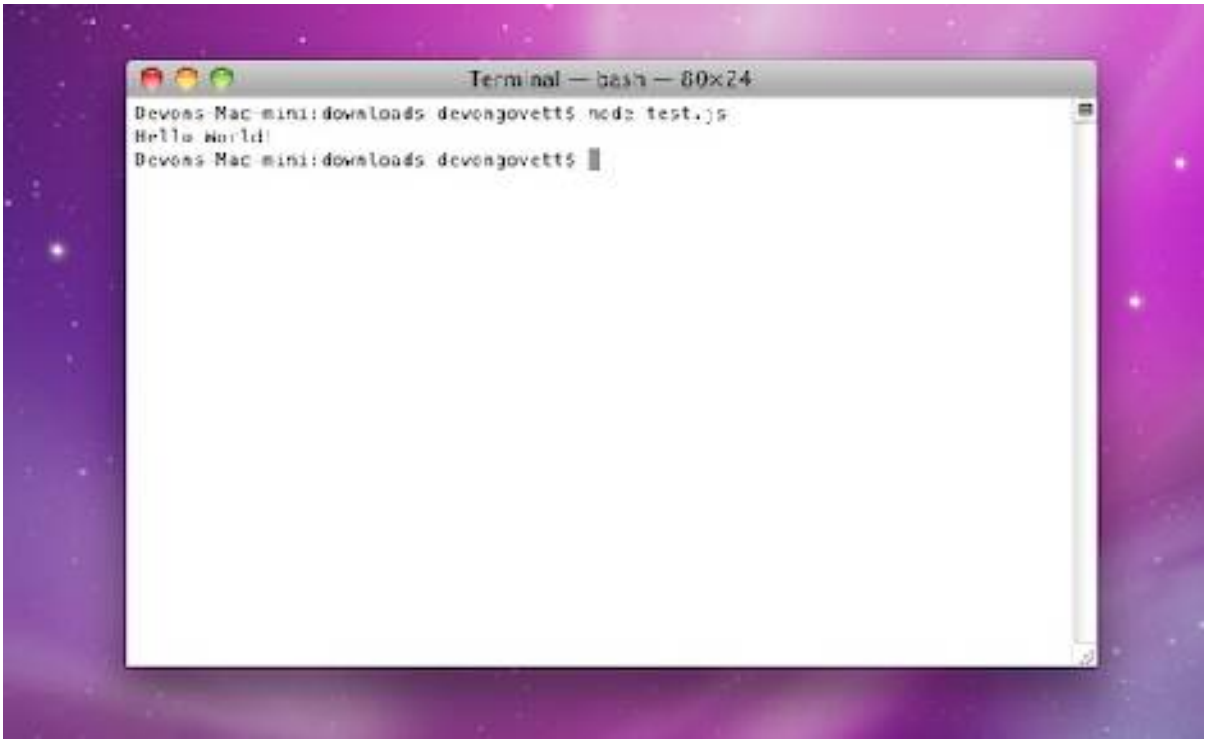
```
sys.puts("Hello World!");
```

Running these two lines is as simple as running the `node` command with the filename of the javascript file as an argument.

1. node test.js

```
node test.js
```

This will output “Hello World!” to the command line when run.



To create an HTTP server, you must `require` the `http` module.

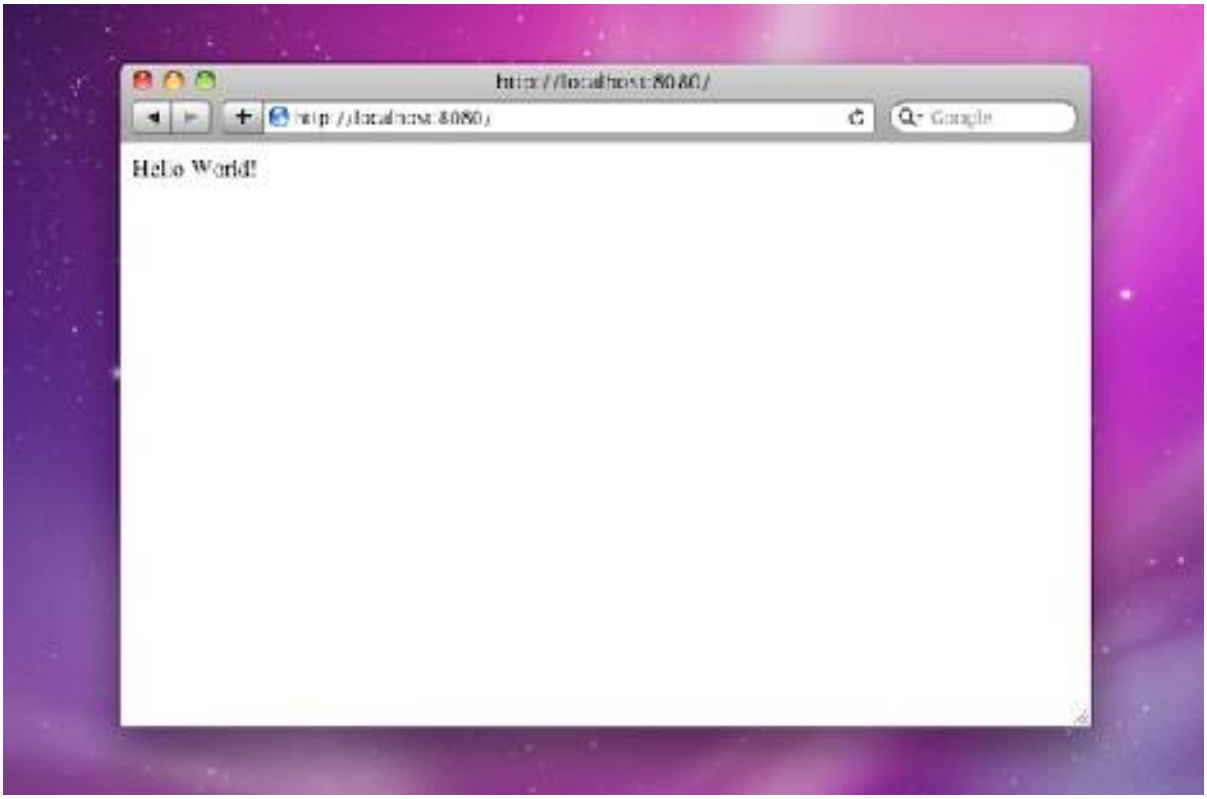
view plaincopy to clipboardprint?

1. var sys = require("sys"),
2. http = require("http");
3.
4. http.createServer(function(request, response) {
5. response.sendHeader(200, {"Content-Type": "text/html"});

```
6.      response.write("Hello World!");
7.      response.close();
8.  }).listen(8080);
9.
10.sys.puts("Server running at http://localhost:8080/");
```

```
var sys = require("sys"),
    http = require("http");

http.createServer(function(request, response) {
    response.sendHeader(200, {"Content-Type": "text/html"});
    response.write("Hello World!");
    response.close();
}).listen(8080);

sys.puts("Server running at http://localhost:8080/");
```

This script imports the `sys` and `http` modules, and creates an HTTP server. The anonymous function that is passed into `http.createServer` will be called whenever a request comes in to the server. Once the server is created, it is told to listen on port 8080. When a request to the server comes in, we first send HTTP headers with the content type and status code of 200 (successful). Then we send “Hello World!” and close the connection. You might notice that we have to explicitly close the connection. This will make it very easy to stream data to the client without closing the connection. If you run this script and go to `http://localhost:8080/` in your browser, you will see “Hello World!”



Step 3 A Simple Static File Server

OK, so we have built an HTTP server, but it doesn't send anything except for "Hello World," no matter what URL you go to. Any HTTP server must be able to send static files such as HTML files, images and other files. The following code does just that:

view plaincopy to clipboardprint?

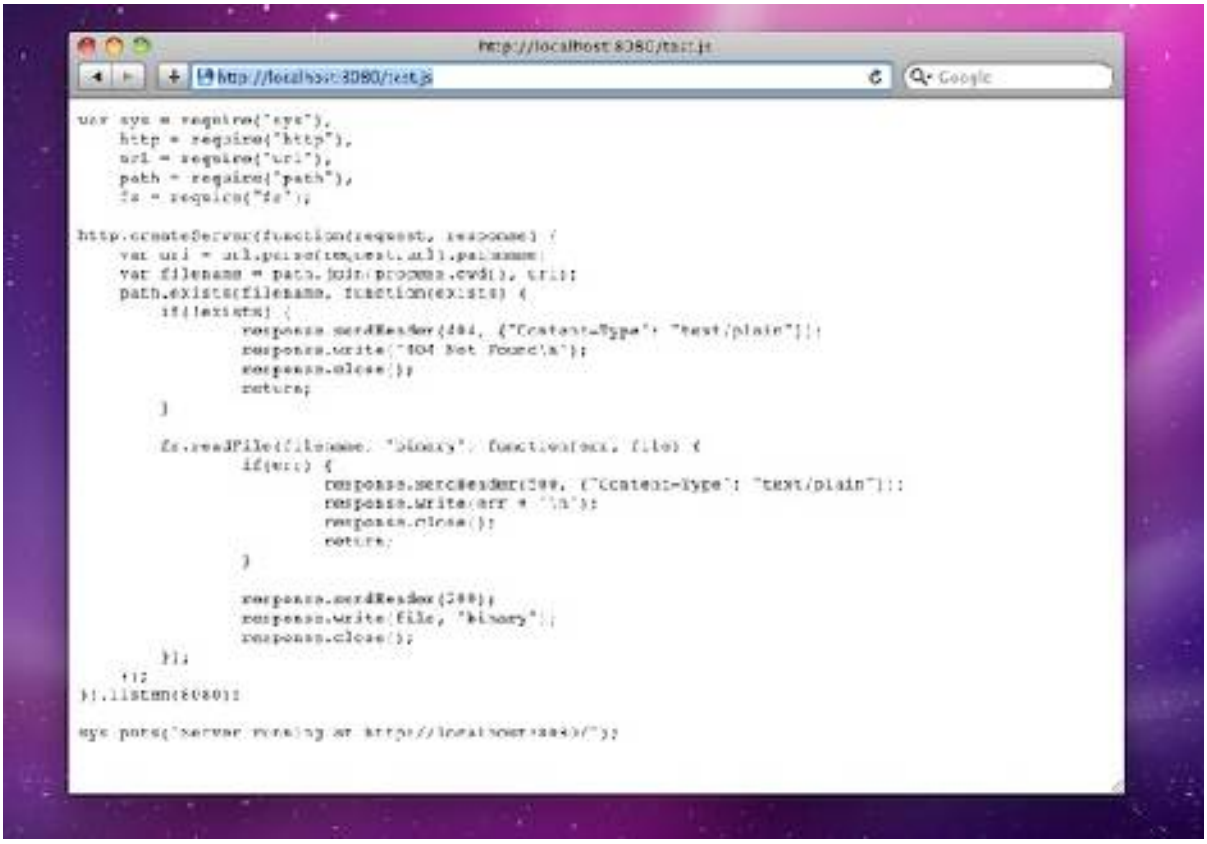
```
1.  var sys = require("sys"),
2.      http = require("http"),
3.      url = require("url"),
4.      path = require("path"),
5.      fs = require("fs");
6.
7.  http.createServer(function(request, response) {
8.      var uri = url.parse(request.url).pathname;
9.      var filename = path.join(process.cwd(), uri);
10.     path.exists(filename, function(exists) {
11.         if(!exists) {
12.             response.sendHeader(404, {"Content-Type": "text/plain"});
13.             response.write("404 Not Found\n");
14.             response.close();
15.             return;
16.         }
17.
18.         fs.readFile(filename, "binary", function(err, file) {
19.             if(err) {
20.                 response.sendHeader(500, {"Content-Type": "text/plain"});
21.                 response.write(err + "\n");
22.                 response.close();
23.                 return;
24.             }
25.
26.             response.sendHeader(200);
27.             response.write(file, "binary");
28.             response.close();
29.         });
30.     });
31.}).listen(8080);
32.
33.sys.puts("Server running at http://localhost:8080/");
```

```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
```

```
        if(!exists) {
                response.sendHeader(404, {"Content-Type": "text/plain"});
                response.write("404 Not Found\n");
                response.close();
                return;
        }

        fs.readFile(filename, "binary", function(err, file) {
                if(err) {
                        response.sendHeader(500, {"Content-Type": "text/plain"});
                        response.write(err + "\n");
                        response.close();
                        return;
                }

                response.sendHeader(200);
                response.write(file, "binary");
                response.close();
        });
    });
}).listen(8080);

sys.puts("Server running at http://localhost:8080/");
```

We start by requiring all of the modules that we will need in our code. This includes the `sys`, `http`, `url`, `path`, and `fs` or filesystem modules. Next we create an HTTP server like we did before. This time, we will use the `url` module to parse the incoming URL of the request and find the pathname of the file being accessed. We find the actual filename on the server's hard drive by using `path.join`, which joins `process.cwd()`, or the current working directory, with the path to the file being requested. Next, we check if the file exists, which is an asynchronous operation and thus requires a callback. If the file does not exist, a 404 Not Found message is sent to the user and the function returns. Otherwise, we read the file using the `fs` module using the "binary" encoding, and send the file to the user. If there is an error reading the file, we present the error message to the user, and close the connection. Because all of this is asynchronous, the server is able to serve other requests while reading the file from the disk no matter how large it is.

If you run this example, and navigate to `http://localhost:8080/path/to/file`, that file will be shown in your browser.



Step 4 A Real Time Tweet Streamer

Building on our static file server, we will build a server in Node.js that streams tweets to a client that is served through our static file server. To start, we will need one extra module in this example: the `events` module. Node has a concept called an `EventEmitter`, which is used all over to handle event listeners for asynchronous tasks. Much like in jQuery or another client side JavaScript framework where you bind event listeners to things like mouse clicks, and AJAX requests, Node allows you to bind event listeners to many things, some of which we have already used. These include every I/O operation, such as reading a file, writing a file, checking if a file exists, waiting for HTTP requests, etc. The `EventEmitter` abstracts the logic of binding, unbinding, and triggering such event listeners. We will be using an `EventEmitter` to notify listeners when new tweets are loaded. The first few lines of our tweet streamer imports all of the required modules, and defines a function for handling static files, which was taken from our previous example.

view plaincopy to clipboardprint?

```
1.  var sys = require("sys"),
2.      http = require("http"),
3.      url = require("url"),
4.      path = require("path"),
5.      fs = require("fs"),
6.      events = require("events");
7.
8.  function load_static_file(uri, response) {
9.      var filename = path.join(process.cwd(), uri);
```

```
    path.exists(filename, function(exists) {
      if(!exists) {
        response.sendHeader(404, {"Content-Type": "text/plain"});
        response.write("404 Not Found\n");
        response.close();
        return;
      }

      fs.readFile(filename, "binary", function(err, file) {
        if(err) {
          response.sendHeader(500, {"Content-Type": "text/plain"});
          response.write(err + "\n");
          response.close();
          return;
        }

        response.sendHeader(200);
        response.write(file, "binary");
        response.close();
      });
    });
}
```

```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs"),
    events = require("events");

function load_static_file(uri, response) {
        var filename = path.join(process.cwd(), uri);
        path.exists(filename, function(exists) {
                if(!exists) {
                        response.sendHeader(404, {"Content-Type": "text/plain"});
                        response.write("404 Not Found\n");
                        response.close();
                        return;
                }

                fs.readFile(filename, "binary", function(err, file) {
                        if(err) {
                                response.sendHeader(500, {"Content-Type":
"text/plain"});
                                response.write(err + "\n");
                                response.close();
                                return;
                        }

                        response.sendHeader(200);
                        response.write(file, "binary");
                        response.close();
                });
        });
```

```
}
```

We have used the `http` module to create a server before, but it is also possible to create an HTTP client using the module. We will be creating an HTTP client to load tweets from Twitter's public timeline, which is performed by the `get_tweets` function.

view plaincopy to clipboardprint?

```
1. var twitter_client = http.createClient(80, "api.twitter.com");
2.
3. var tweet_emitter = new events.EventEmitter();
4.
5. function get_tweets() {
6.     var request = twitter_client.request("GET", "/1/statuses/public_timeline.json", {"host": "api.t
   witter.com"});
7.
8.     request.addListener("response", function(response) {
9.         var body = "";
10.        response.addListener("data", function(data) {
11.            body += data;
12.        });
13.
14.        response.addListener("end", function() {
15.            var tweets = JSON.parse(body);
16.            if(tweets.length > 0) {
17.                tweet_emitter.emit("tweets", tweets);
18.            }
19.        });
20.    });
21.
22.    request.close();
23.}
24.
25.setInterval(get_tweets, 5000);
```

```
var twitter_client = http.createClient(80, "api.twitter.com");

var tweet_emitter = new events.EventEmitter();

function get_tweets() {
        var request = twitter_client.request("GET",
"/1/statuses/public_timeline.json", {"host": "api.twitter.com"});

        request.addListener("response", function(response) {
                var body = "";
                response.addListener("data", function(data) {
                        body += data;
                });

                response.addListener("end", function() {
                        var tweets = JSON.parse(body);
                        if(tweets.length > 0) {
                                tweet_emitter.emit("tweets", tweets);
```

```
                        }
                });
        });

        request.close();
}

setInterval(get_tweets, 5000);
```

First, we create an HTTP client on port 80 to api.twitter.com, and create a new `EventEmitter`. The `get_tweets` function creates an HTTP "GET" request to Twitter's public timeline, and adds an event listener that will be triggered when Twitter's servers respond. Because Node.js is asynchronous, the data in the body of the response comes in chunks, which are picked up by the response's "data" listener. This listener simply appends the chunk to the `body` variable. Once all of the chunks have come in, the "end" listener is triggered, and we parse the incoming JSON data. If more than one tweet is returned, we `emit` the "tweets" event on our `tweet_emitter`, and pass in the array of new tweets. This will trigger all of the event listeners listening for the "tweets" event, and send the new tweets to each client. We retreive the new tweets every five seconds, by using `setInterval`.

Finally, we need to create the HTTP server to handle requests.

view plaincopy to clipboardprint?

```
1. http.createServer(function(request, response) {
2.     var uri = url.parse(request.url).pathname;
3.     if(uri === "/stream") {
4.
5.         var listener = tweet_emitter.addListener("tweets", function(tweets) {
6.             response.sendHeader(200, { "Content-Type" : "text/plain" });
7.             response.write(JSON.stringify(tweets));
8.             response.close();
9.
10.            clearTimeout(timeout);
11.        });
12.
13.        var timeout = setTimeout(function() {
14.            response.sendHeader(200, { "Content-Type" : "text/plain" });
15.            response.write(JSON.stringify([]));
16.            response.close();
17.
18.            tweet_emitter.removeListener(listener);
19.        }, 10000);
20.
21.    }
22.    else {
23.        load_static_file(uri, response);
24.    }
25.}).listen(8080);
26.
27.sys.puts("Server running at http://localhost:8080/");
```

```
http.createServer(function(request, response) {
```

```
    var uri = url.parse(request.url).pathname;
    if(uri === "/stream") {

        var listener = tweet_emitter.addListener("tweets", function(tweets) {
                response.sendHeader(200, { "Content-Type" : "text/plain" });
                response.write(JSON.stringify(tweets));
                response.close();

                clearTimeout(timeout);
        });

        var timeout = setTimeout(function() {
                response.sendHeader(200, { "Content-Type" : "text/plain" });
                response.write(JSON.stringify([]));
                response.close();

                tweet_emitter.removeListener(listener);
        }, 10000);

    }
    else {
        load_static_file(uri, response);
    }
}).listen(8080);

sys.puts("Server running at http://localhost:8080/");
```

Just as we did with our static file server, we create an HTTP server that listens on port 8080. We parse the requested URL, and if the URL is equal to `"/stream"`, we will handle it, otherwise we pass the request off to our static file server. Streaming consists of creating a listener to listen for new tweets on our `tweet_emitter`, which will be triggered by our `get_tweets` function. We also create a timer to kill requests tht last over 10 seconds by sending them an empty array. When new tweets come in, we send the tweets as JSON data, and clear the timer. You will see how this works better after seeing the client side code, which is below. Save it as `test.html` in the same directory as the server side JavaScript.

view plaincopy to clipboardprint?

```
1. <!DOCTYPE html>
2. <html>
3.     <head>
4.         <title>Tweet Streamer</title>
5.         <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.
   min.js"></script>
6.     </head>
7.     <body>
8.         <ul id="tweets"></ul>
9.         <script type="text/javascript">
10.        var tweet_list = $("#tweets");
11.
12.        function load_tweets() {
13.            $.getJSON("/stream", function(tweets) {
14.                $.each(tweets, function() {
15.                    $("<li>").html(this.text).prependTo(tweet_list);
```

```
16.             });
17.             load_tweets();
18.         });
19.     }
20.
21.     setTimeout(load_tweets, 1000);
22.     </script>
23. </body>
24.</html>
```

```
<!DOCTYPE html>
<html>
        <head>
                <title>Tweet Streamer</title>
                <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js"></script>
        </head>
        <body>
                <ul id="tweets"></ul>
                <script type="text/javascript">
                var tweet_list = $("#tweets");

                function load_tweets() {
                        $.getJSON("/stream", function(tweets) {
                                $.each(tweets, function() {
                                        $
("<li>").html(this.text).prependTo(tweet_list);
                                });
                                load_tweets();
                        });
                }

                setTimeout(load_tweets, 1000);
                </script>
        </body>
</html>
```

Here, we have a simple HTML page that imports the jQuery library and defines an unordered list to put the tweets in. Our client side JavaScript caches the tweet list, and runs the `load_tweets` function after one second. This gives the browser enough time to finish loading the page before we start the AJAX request to the server. The `load_tweets` function is very simple: It uses jQuery's `getJSON` function to load `/stream`. When a response comes in, we loop through all of the tweets and prepend them to the tweet list. Then, we call `load_tweets` again. This effectively creates a loop that loads new tweets, which times out after ten seconds because of the timeout on the server. Whenever there are new tweets, they are pushed to the client which maintains a continuous connection to the server. This technique is called long-polling.

If you run the server using `node` and go to `http://localhost:8080/test.html`, you will see the Twitter public timeline stream into your browser.

Next Steps

Node.js is a very exciting technology that makes it easy to create high performance real time applications. I hope you can see its benefit, and can use it in some of your own applications. Because of Node's great module system, it is easy to use prewritten code in your application, and there are many third party modules available for just about everything – including database connection layers, templating engines, mail clients, and even entire frameworks connecting all of these things together. You can see a complete list of modules on the Node.js wiki, and more Node tutorials can be found on How To Node. I would also recommend that you watch a video from JSConf, in which Ryan Dahl, the creator of Node, describes the design philosophy behind Node. That is available here.

I hope you have enjoyed this tutorial. If you have any comments, you can leave one here or send me a message on Twitter. Happy noding!