

(/visit/heroku-151208-node.js)

# Node.js Tutorial – Step-by-Step Guide For Getting Started

(/HE



Alexandru Vladutu

Alexandru has worked with Node.js since v 0.4 and is the #2 StackOverflow answerer for NodeJS and #1 for Express.

(/billing)



NEED 1 ON 1 E



TABLE OF CON

1 Introducti

1.1 What's n  
than Rails ar

1.2 Use case

1.3 npm, the  
manager

1.4 Resourc

2 Installing

2.1 Resourc

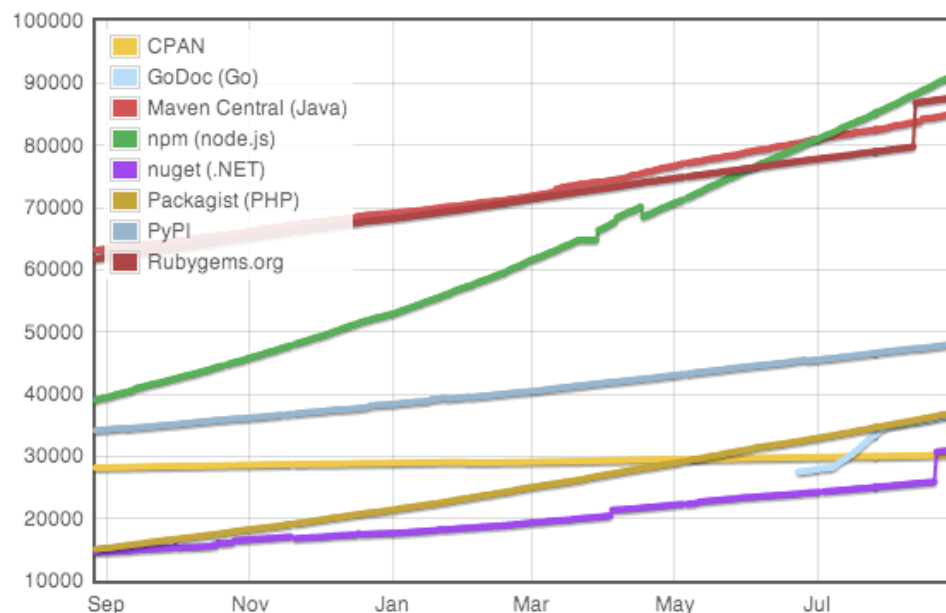


- 2.1 Resource
- 3 Node Fundamentals
  - 3.1 Modules
  - 3.2 Callback
  - 3.3 Events
  - 3.4 Streams
  - 3.5 Resource
- 4 Error Handling
  - 4.1 Error-first
  - 4.2 EventEmitter
  - 4.3 Propagation with the version
  - 4.4 Resource
- 5 Debugging with node-inspector
  - 5.1 Resource
- 6 Creating a web server with Express
  - 6.1 Resource
- 7 Summary

# 1 Introduction

The popularity of JavaScript applications has been skyrocketing in the last few years, with Node.js definitely facilitating this growth. If we look at [modulecounts.com](http://www.modulecounts.com/) (<http://www.modulecounts.com/>) we will see that there are more Node packages in the wild than in the Ruby world. In addition, Node packages are growing faster than Ruby, Python, and Java combined.

# Module Counts



## Include

- ☐ Bower (JS)
- ☐ Clojars (Clojure)
- ☒ CPAN
- ☐ CPAN (search)
- ☐ CRAN (R)
- ☒ GoDoc (Go)
- ☐ Hackage (Haskell)
- ☐ Hex.pm (Elixir/Erlang)
- ☒ Maven Central (Java)
- ☐ MELPA (Emacs)
- ☒ npm (node.js)
- ☒ nuget (.NET)
- ☒ Packagist (PHP)
- ☐ Pear (PHP)
- ☒ PyPI
- ☒ Rubygems.org

time period ☐ all time ☒ last year ☐ last 90 days ☐ last 30 days ☐ last 7 days

	Aug 20	Aug 21	Aug 22	Aug 23	Aug 24	Aug 25	Aug 26	Avg Growth
<u>Bower (JS)</u>								50/day
<u>Clojars (Clojure)</u>	10015	10020	10027	10037	10044	10066	10078	10/day
<u>CPAN</u>	30179	30185	30189	30198	30205	30208	30216	6/day
<u>CPAN (search)</u>	30179	30185	30189	30198	30205	30208	30216	6/day
<u>CRAN (R)</u>	5815	5816	5817	5822	5826	5826	5832	1/day
<u>GoDoc (Go)</u>	36099	36206	36256	36355	36371	36434	36498	60/day
<u>Hackage (Haskell)</u>	6805	6809	6815	6818	6822	6830	6833	4/day
<u>Hex.pm (Elixir/Erlang)</u>	171	172	172	174	174	177	177	1/day
<u>Maven Central (Java)</u>	84529	84628	84671	84772	84809	84879	84998	74/day
<u>MELPA (Emacs)</u>	1948	1950	1950	1950	1951	1951	1954	1/day
<u>npm (node.js)</u>	89963	90176	90336	90504	90644	90817	91006	181/day
<u>nuget (.NET)</u>	30707	30758	30800	30834	30861	30893	30948	724/day
<u>Packagist (PHP)</u>	36447	36566	36656	36702	36757	36815	36916	77/day
<u>Pear (PHP)</u>	598	598	598	598	598	598	598	0/day
<u>PyPI</u>	47677	47726	47773	47802	47839	47885	47931	44/day
<u>Rubygems.org</u>	87316	87367	87408	87461	87493	87532	87586	47/day

Data from modulecounts.com (collected by scraping the relevant websites once a day)

In this article we are going to take a look at the most important aspects of Node so you can get on track with it and start building applications right away.

## 1.1 What's making Node more popular than Rails and other alternatives?

Node markets itself as an asynchronous, event-driven framework built on top of Chrome's JavaScript engine and designed for creating scalable network applications. It's basically JavaScript plus a bunch of C/C++ under the hood for things like interacting with the filesystem, starting up HTTP or TCP servers and so on.

Node is single-threaded and uses a concurrency model based on an event loop. It is non-blocking, so it doesn't make the program wait, but instead it registers a callback and lets the program continue. This means it can handle concurrent operations without multiple threads of execution, so it can scale pretty well.

In a sequential language such as PHP, in order to get the HTML content of a page you would do the following:

```
1 | $response = file_get_contents("http://example.com");  
2 | print_r($response);
```

javascript

In Node, you register some callbacks instead:

```
1 | var http = require('http');                                javascript
2 |
3 | http.request({ hostname: 'example.com' }, function(res) {
4 |     res.setEncoding('utf8');
5 |     res.on('data', function(chunk) {
6 |         console.log(chunk);
7 |     });
8 | }).end();
```

There are two big differences between the two implementations:

- Node allows you to perform other tasks while waiting to be notified when the response is available.
- The Node application is not buffering data into memory, but instead it's outputting it chunk-by-chunk.

While other event loop systems exist (such as the EventMachine library in Ruby or Twisted in Python), there is a significant difference between them and Node.

In Node, all of the libraries have been designed from the ground up to be non-blocking, but the same cannot be said for the others.

## 1.2 Use cases

Node is ideal for I/O bound applications (or those that wait on user events), but not so great for CPU-heavy applications. Good examples include data-intensive realtime applications (DIRT), single page applications, JSON APIs, and data-streaming applications.

## 1.3 npm, the official Node package manager

Node owes a big part of its success to npm, the package manager that comes bundled with it. There are a lot of great things about npm:

- It installs application dependencies locally, not globally.
- It handles multiple versions of the same module at the same time.
- You can specify tarballs or git repositories as dependencies.
- It's really easy to publish your own module to the npm registry.
- It's useful for creating CLI utilities that others can install (with `npm`) and use right away.

## 1.4 Resources

For more details on why you would use Node, check out this article (<http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js>).

## 2 Installing Node.js and NPM

There are native installers for Node on Windows and OS X, as well as the possibility of installing it via a package manager. However sometimes you will want to test your code with different Node versions, and that's where NVM (Node version manager) comes in.

With NVM you can have multiple versions of Node installed on your system and switch between them easily. In the next few lines we are going to see how to install NVM on an Ubuntu system.

First, we have to make sure our system has a C++ compiler:

```
1 | $ sudo apt-get update                                     bash
2 | $ sudo apt-get install build-essential libssl-dev
```

After that we can copy-paste the one-line installer for NVM into the terminal:

```
1 | $ curl https://raw.githubusercontent.com/creationix/nvm/v0.13.1/install.sh | bash    bash
```

At this moment NVM should be properly installed, so we will logout and login to verify that:

```
1 | $ nvm                                                     bash
```

If there's no error when typing in the `nvm` command, that means that everything's alright. Now we can move on to actually installing Node and npm.

```
1 | $ nvm install v0.10.31                                     bash
```

The output should look like this:

```
1 | $ nvm install v0.10.31                                     bash
2 |
3 | ##### 100.0%
4 |
5 | Now using node v0.10.31
```

Both `node` and `npm` should be available in the terminal now:

```
1 | $ node -v && npm -v          bash
2 | v0.10.31
3 | 1.4.23
```

There is one last thing we need to do so we can always use this version of Node when we login the next time: making this version the default one.

```
1 | $ nvm alias default 0.10.31    bash
```

We can install other Node versions just like we did before and switch between them with the `nvm use` command:

```
1 | $ nvm install v0.8.10          bash
2 | $ nvm use v0.8.10
```

In case you are not sure what versions you have installed on your system just type `nvm list`. That will show you the full list of versions and also the current and default versions, such as the following:

```
$ nvm list
```

```
v0.6.3 v0.6.12 v0.6.14 v0.6.19 v0.7.7 v0.7.8 v0.7.9 v0.8.6 v0.8.11 v0.10.3 v0.10.12
v0.10.15 v0.10.21 v0.10.24 v0.11.9 current: v0.10.24 default -> v0.10.24
```



## 2.1 Resources

For more details on how to install Node with NVM, check out this article (<https://www.digitalocean.com/community/tutorials/how-to-install-node-js-with-nvm-node-version-manager-on-a-vps>).

## 3 Node Fundamentals

We are going to look at the main Node.js concepts next:

- How to include external libraries by requiring modules
- The role of callbacks
- The EventEmitter pattern
- Streams data one chunk at a time

### 3.1 Modules

Java or Python use the `import` function to load other libraries, while PHP and Ruby use `require`. Node implements the CommonJS interface for modules. In Node you can also load other dependencies using the `require` keyword.

For example, we can require some native modules:

```
1 | var http = require('http');  
2 | var dns = require('dns');
```

javascript

We can also require relative files:

```
1 | var myFile = require('./myFile'); // loads myFile.js
```

javascript

To install modules from npm, either search for them on the website (<https://npmjs.org/>) or on Github. The syntax for installing an npm module locally is pretty straightforward:

```
1 | # where express === module name
2 | $ npm install express
```

bash

You can require module install from npm as you would do with the native ones, no need to specify the absolute or relative path:

```
1 | var express = require('express');
```

javascript

The nice thing about requiring Node modules is that they aren't automatically injected into the global scope, but instead you just assigned them to a variable of your choice. That means that you don't have to care about two or more modules that have functions with the same name.

When creating your own modules, all you have to do is take care when exporting something (whether it's a function, an object, a number or so on). The first approach would be to export a single object:

```
1 | var person = { name: 'John', age: 20 };
2 |
3 | module.exports = person;
```

javascript

The second approach requires adding properties to the `exports` object:

```
1 | exports.name = 'John';
2 | exports.age = 20;
```

javascript

A thing to note about modules is that they don't share scope, so if you want to share a variable between different modules, you must include it into a separate module that is then required by the other modules. Another interesting thing you should remember is that modules are only loaded once, and after that they are cached by Node.

Unlike the browser, Node does not have a `window` global object, but instead has two others: `globals` and `process`. However, you should seriously avoid adding properties on the two.

## 3.2 Callbacks

In asynchronous programming we do not return values when our functions are done, but instead we use the continuation-passing style (CPS). You can read more on CPS here ([http://en.wikipedia.org/wiki/Continuation-passing\\_style](http://en.wikipedia.org/wiki/Continuation-passing_style)).

With this style, an asynchronous function invokes a callback (a function usually passed as the last argument) to continue the program once the it has finished.

Below is an example that looks up IPv4 addresses for a domain:

```
1 | var dns = require('dns');                                javascript
2 |
3 | dns.resolve4('www.google.com', function (err, addresses) {
4 |     if (err) throw err;
5 |
6 |     console.log('addresses: ' + JSON.stringify(addresses));
7 | });
```

We have passed a callback (the inline anonymous function) as the second argument to the `dns.resolve4` asynchronous function. Once the async function has the response ready for us it will invoke the callback, thus continuing the program execution. This is how we make use of CPS.

### 3.3 Events

The standard callback pattern works well for the use cases where we want to be notified when the async function finishes. However, there are situations that require being notified of different events that do not occur at the same time.

Let us look at an example involving an IRC client:

```
1 var irc = require('irc');
2 var client = new irc.Client('irc.freenode.net', 'myIrcBot', {
3   channels: ['#sample-channel']
4 });
5
6 client.on('error', function(message) {
7   console.error('error: ', message);
8 });
9
10 client.on('connect', function() {
11   console.log('connected to the irc server');
12 });
13
14 client.on('message', function (from, to, message) {
15   console.log(from + ' => ' + to + ': ' + message);
16 });
17
18 client.on('pm', function (from, message) {
19   console.log(from + ' => ME: ' + message);
```

javascript

We are listening to different types of events in the above example:

- The `connect` event is emitted when the client has successfully connected to the IRC server.
- The `error` event is triggered in case an error occurs.
- The `message` and `pm` events are emitted for incoming messages.

The events mentioned above make this situation ideal for using the `EventEmitter` pattern.

The `EventEmitter` pattern allows implementors to emit an event to which the consumers can subscribe if they are interested. This pattern may be familiar to you from the browser, where it is used for attaching DOM event handlers.

Node has an `EventEmitter`

([http://nodejs.org/api/events.html#events\\_class\\_events\\_eventemitter](http://nodejs.org/api/events.html#events_class_events_eventemitter)) class in core which we can use to make our own `EventEmitter` objects. Let's create a `MemoryWatcher` class that inherits from `EventEmitter` and emits two types of events:

- A data event at a regular interval, representing the memory usage in bytes
- An error event, in case the memory exceeds a certain limit imposed

The `MemoryWatcher` class will look like the following:

```
1 var EventEmitter = require('events').EventEmitter;
2 var util = require('util');
3
4 function MemoryWatcher(opts) {
5   if (!(this instanceof MemoryWatcher)) {
6     return new MemoryWatcher();
7   }
8
9   opts = opts || {
10     frequency: 30000 // 30 seconds
11   };
12
13   EventEmitter.call(this);
14
15   var that = this;
16
17   setInterval(function() {
18     var bytes = process.memoryUsage().rss;
19
```

javascript

An easier way to create `EventEmitter` objects is to make new instances from the raw `EventEmitter` class:

```
1 | var EventEmitter = require('events').EventEmitter;
2 | var emitter = new EventEmitter();
3 | setInterval(function() {
4 |     console.log(process.memoryUsage().rss);
5 | }, 30000);
```

javascript

## 3.4 Streams

Streams (<http://nodejs.org/api/stream.html>) represent an abstract interface for asynchronously manipulating a continuous flow of data. They are similar to Unix pipes and can be classified into five types: readable, writable, transform, duplex and "classic".

As with Unix pipes, Node streams implement a composition operator called `.pipe()`. The main benefits of using streams are that you don't have to buffer the whole data into memory and they're easily composable.

To have a better understanding of how streams work we will create an application that reads a file, encrypts it using the AES-256 algorithm and then compresses it using gzip. All of this using streams, which means that for each chunk read it will encrypt and compress it.

```
1 var crypto = require('crypto');
2 var fs = require('fs');
3 var zlib = require('zlib');
4
5 var password = new Buffer(process.env.PASS || 'password');
6 var encryptStream = crypto.createCipher('aes-256-cbc', password);
7
8 var gzip = zlib.createGzip();
9 var readStream = fs.createReadStream(**filename); // current file
10 var writeStream = fs.createWriteStream(**dirname + '/out.gz');
11
12 readStream // reads current file
13   .pipe(encryptStream) // encrypts
14   .pipe(gzip) // compresses
15   .pipe(writeStream) // writes to out file
16   .on('finish', function () { // all done
17     console.log('done');
18   });
```

javascript

Here we take a readable stream, pipe it into an encryption stream, then pipe that into a gzip compression stream and finally pipe it into a write stream (writing the content to disk). The encryption and compression streams are transform streams, which represent duplex streams where the output is in some way computed from the input.

After running that example we should see a file called `out.gz` . Now it's time to implement the reverse, which is decrypting the file and outputting the content to the terminal:



javascript

```
1 var crypto = require('crypto');
2 var fs = require('fs');
3 var zlib = require('zlib');
4
5 var password = new Buffer(process.env.PASS || 'password');
6 var decryptStream = crypto.createDecipher('aes-256-cbc', password);
7
8 var gzip = zlib.createGunzip();
9 var readStream = fs.createReadStream(__dirname + '/out.gz');
10
11 readStream // reads current file
12   .pipe(gzip) // uncompresses
13   .pipe(decryptStream) // decrypts
14   .pipe(process.stdout) // writes to terminal
15   .on('finish', function () { // finished
16     console.log('done');
17   });
```

## 3.5 Resources

For more details on Node fundamentals, read through this overview (<http://webapplog.com/node-js-fundamentals-a-concise-overview-of-the-main-concepts/>). For a more in-depth guide on streams, check out the stream-handbook (<https://github.com/substack/stream-handbook>).

## 4 Error Handling

Error handling is one of the most important topics in Node. If you ignore errors or deal with them improperly, your entire application might crash or be left in an inconsistent state.

## 4.1 Error-first callbacks

The "error-first" callback is a standard protocol for Node callbacks. It originated in Node core, but it has spread into userland as well to become today's standard. This is a very simple convention, with basically one rule: the first argument for the callback function should be the error object.

That means that there are two possible scenarios:

- If the error argument is null, then the operation was successful.
- If the error argument is set, then an error occurred and you need to handle it.

Let's take a look at how we read a file's content with Node:

```
1 | fs.readFile('/foo.txt', function(err, data) {                                javascript
2 |     // ...
3 | });
```

The callback for `fs.readFile` has two arguments: the error and the file content.

Now let's implement a similar function that reads the content of multiple files, passed as an array argument. The signature for the function should look similar, but instead of passing a single file path we will pass in an array this time:

```
1 | readFiles(filesArray, callback);                                           javascript
```

We will respect the error-first pattern and won't handle the error in the `readFiles` function, but will delegate that responsibility to the callback. The `readFiles` function will loop over the file paths and read the content for each. If it encounters an error, it will invoke the callback only once. After it's finished reading the content for the last file in the array it will invoke the array with `null` as the first argument.

```
1 var fs = require('fs');
2
3 function readFiles(files, callback) {
4   var filesLeft = files.length;
5   var contents = {};
6   var error = null;
7
8   var processContent = function(filePath) {
9     return function(err, data) {
10       // an error was previously encountered and the callback was invoked
11       if (error !== null) { return; }
12
13       // an error happen while trying to read the file, so invoke the callback
14       if (err) {
15         error = err;
16         return callback(err);
17       }
18
19       contents[filePath] = data;
```

javascript

## 4.2 EventEmitter errors

We have to be careful when dealing with event emitters (that means streams too), because if there's an unhandled error event it will crash our application. Here is the most simple example of such an event, triggered by ourselves:

```
1 | var EventEmitter = require('events').EventEmitter;                                javascript
2 |
3 | var emitter = new EventEmitter();
4 |
5 | emitter.emit('error', new Error('something bad happened'));
```

Depending on your application this might be a fatal error (unrecoverable) or an error that should not crash your application (like failed sending an email for example).

Either way you should attach an error event handler:

```
1 | emitter.on('error', function(err) {                                                javascript
2 |     console.error('something went wrong with the ee:' + err.message);
3 | });
```

## 4.3 Propagating more descriptive errors with the `verror` module

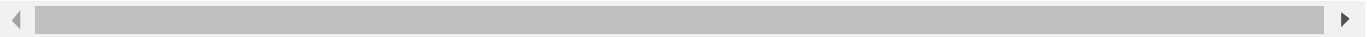
There are a lot of situations where we'll want to delegate the error to the callback and don't deal with the error ourselves. In fact, that's exactly what we did with the `readFiles` function we created earlier. In case there's an error reading the file we will just delegate that to the callback.

Let's try to call the function with a non-existent file and see what happens:

```
1 | readFiles(['non-existing-file'], function(err, contents) {           javascript
2 |     if (err) { throw err; }
3 |
4 |     console.log(contents);
5 | });
```

The output should be something like the following:

```
1 | $ node readFiles.js                                           bash
2 |
3 | /Users/alexandruvladutu/www/airpair-article/examples/readFiles.js:34
4 |     if (err) { throw err; }
5 |                     ^
6 | Error: ENOENT, open '/Users/alexandruvladutu/www/airpair-article/examples/non-existing-file'
```



That's not super helpful, especially because in real-world situations there will probably be a function that calls another function that calls the original function. For example, you might have another function called `readMarkdownFiles` that will only read markdown files using the `readFiles` function.

Also, the output above doesn't even provide a useful stack trace, so you would have to dig deeper to find out where exactly the error came from. Luckily we can do something about that by integrating the `verror` module (<http://npm.im/verror>) into our application.

With `verror`, we can wrap our errors to provide more descriptive messages.

We will have to require the module at the beginning of the file and then wrap the error when invoking the callback:

```
1 | var verror = require('verror');                                javascript
2 |
3 | function readFiles(files, callback) {
4 |     ...
5 |     return callback(new VError(err, 'failed to read file %s', filePath));
6 |     ...
7 | }
```

Now let's try to run the example again:

```
1 | $ node readFiles-verror.js                                     bash
2 |
3 |   /Users/alexandruvladutu/www/airpair-article/examples/readFiles-verror.js:35
4 |     if (err) { throw err; }
5 |                   ^
6 | VError: failed to read file /Users/alexandruvladutu/www/airpair-article/examples/non-existir
7 |   at /Users/alexandruvladutu/www/airpair-article/examples/readFiles-verror.js:17:25
8 |   at fs.js:207:20
9 |   at Object.oncomplete (fs.js:107:15)
```

And there it is! Instead of having to search Google for 'ENOENT' and digging through the code, now we know that there was a problem reading the file and that came from the `readFiles` function.

This is a simple example but it shows the power of `verror`. In production this module will be a lot more useful because the codebase will probably be large and the error will be propagated through more functions than in our basic example.

## 4.4 Resources

For more details check out Joyent's guide on Node error handling (<https://www.joyent.com/developers/node/design/errors>).

## 5 Debugging Node applications with node-inspector

For small bugs you can always use `console.log` to track down one thing or another, but for more complex situations there's node-inspector (<https://github.com/node-inspector/node-inspector>). It has a lot of goodies baked in, but the most important are:

- It's based on the Blink Developer Tools, so it should look and feel familiar to frontend developers.
- It has the ability to setup breakpoints.
- We can step over, step in, step out, resume (continue).
- We can inspect scopes, variables, object properties.
- Besides inspecting, we can also edit variables and object properties.

node-inspector is installable via npm:

```
1 | $ npm install -g node-inspector
```

bash

Let's say we have the following basic Node example:

```
1 | var http = require('http');
2 | var port = process.env.PORT || 1337;
3 |
4 | http.createServer(function(req, res) {
5 |   res.writeHead(200, { 'Content-Type': 'text/html' });
6 |   res.end(new Date() + '\n');
7 | }).listen(port);
8 |
9 | console.log('Server running on port %s', port);
```

javascript

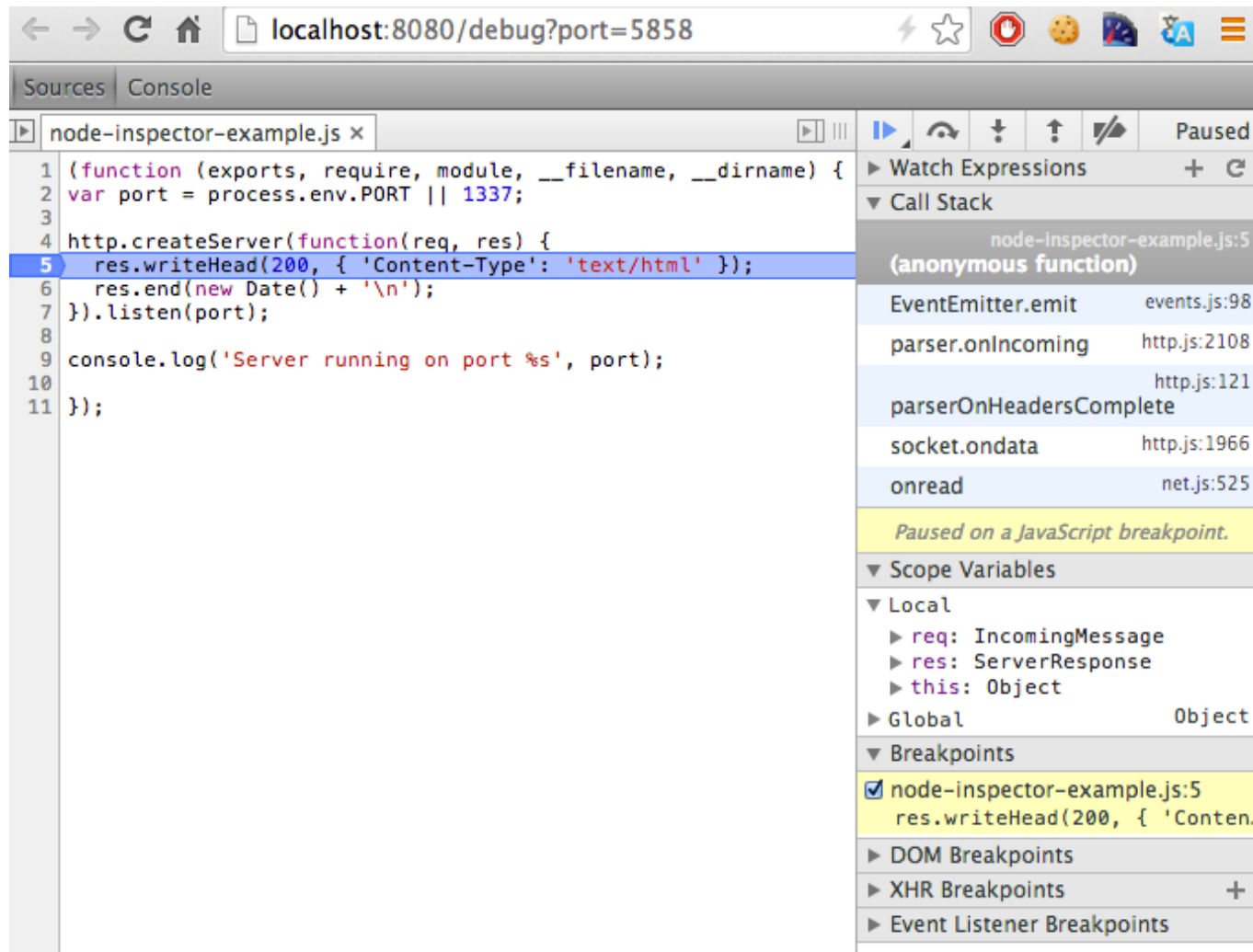
To run our example with node-inspector we just need to type in the following command:

```
1 | # basically `node-debug` instead of `node`
2 | $ node-debug example.js
```

bash

That should start our application and open the node-inspector interface in Chrome. Let's setup a breakpoint in the request handlers (by clicking on the line number for the one containing `res.writeHead`). Now open another tab and visit `http://localhost:1337` (`http://localhost:1337`). The browser should be in a loading stage, but switch to the node-inspector interface.





If you open the console you can inspect the request and response objects, modify them and so on. This is just a basic example to get you started with node-inspector, but in real-world applications you will still benefit from these debugging techniques to track down more complicated issues.

## 5.1 Resource

For more details on debugging, check out this walkthrough (<http://blog.nodeknockout.com/post/34843655876/debugging-with-node-inspector.>).

## 6 Creating a realtime application with Express and Socket.io

Express is the most popular web framework for Node, while Socket.IO is a realtime framework that enables bi-directional communication between web clients and the server. We are going to create a basic tracking pixel application using the two that has a dashboard which reports realtime visits.

Besides Express and Socket.IO we will need to install the `emptygif` module. When the user visits `http://localhost:1337/tpx.gif` (`http://localhost:1337/tpx.gif`), a message will be sent to all the users that are viewing the homepage (`http://localhost:1337/`). The message will contain information related to the clients, mainly their IP address and user agent.

Below is the code for the `server.js` file:

```
1 var emptygif = require('emptygif');
2 var express = require('express');
3 var app = express();
4 var server = require('http').createServer(app);
5 var io = require('socket.io')(server);
6
7 app.get('/tpx.gif', function(req, res, next) {
8   io.emit('visit', {
9     ip: req.ip,
10    ua: req.headers['user-agent']
11  });
12
13  emptygif.sendEmptyGif(req, res, {
14    'Content-Type': 'image/gif',
15    'Content-Length': emptygif.emptyGifBufferLength,
16    'Cache-Control': 'public, max-age=0' // or specify expiry to make sure it will call e
17  });
18 });
19
```

javascript

Now on the frontend all we have to do is listen for that 'visit' event emitted by the server and modify the UI accordingly, like so:

```

1 <!DOCTYPE HTML>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Realtime pixel tracking dashboard</title>
6   <style type="text/css">
7     .visit {
8       margin: 5px 0;
9       border-bottom: 1px dotted #CCC;
10      padding: 5px 0;
11    }
12    .ip {
13      margin: 0 10px;
14      border-left: 1px dotted #CCC;
15      border-right: 1px dotted #CCC;
16      padding: 0 5px;
17    }
18  </style>
19 </head>

```

markup

Now start up the application, open the dashboard in a tab and the tracking pixel URL in different browsers. The dashboard should be similar to this one:

← → ↻ 🏠	localhost:1337	⚡ ☆
<b>Realtime pixel tracking dashboard</b>		
August 24th 2014, 00:26:56	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.143 Safari/537.36
August 24th 2014, 00:27:04	127.0.0.1	curl/7.30.0
August 24th 2014, 00:28:34	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:17.0) Gecko/20100101 Firefox/17.0
August 24th 2014, 00:28:51	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:17.0) Gecko/20100101 Firefox/17.0

## 6.1 Resources

For more details on Express and Socket.io check out this tutorial (<https://www.digitalocean.com/community/tutorials/how-to-install-express-a-node-js-framework-and-set-up-socket-io-on-a-vps>).

## 7 Summary

Node isn't a silver bullet, but hopefully you have more insights on the proper use cases by now. In short, Node is a great option for applications that wait on I/O and have to handle a lot of concurrent connections.

The npm registry is growing on a daily basis, which means there are more and more modules ready to be used. We have not only learned how to setup Node, but also core concepts such as callbacks, events and streams. In the last part of the article we tackled production topics such as error handling, debugging, and creating practical applications.

If you are left wondering if Node has matured yet, you should know that popular companies such as Yahoo, Walmart or PayPal are using it in production. What's stopping you? If you have any problems or further questions, I would be happy to jump on an AirPair and help you work through them.

14 Comments   AirPair

 Login ▾

 Recommend   10    Share

Sort by Best ▾

