« Home / All Guides

Felix's Node.js Beginners Guide

There is lots of information about node.js, but given the rapid pace at which it is developing, it can be difficult for beginners to find good, current information on how to get started. This guide aims to provide exactly that, whilst staying updated with the latest stable version of node.js.

This guide has been updated to reflect the latest changes in node 0.4.x, the currently stable branch of node.js.

Learning JavaScript

This guide assumes that you are already familar with JavaScript. If you are not, you should go ahead and read: Eloquent JavaScript, a free online book by Marijn Haverbeke.

Hello World Tutorial

This tutorial guides you through installing node.js, including the creation of a simple hello world http server.

Installation

First of all: You should run a *nix operating system in order to use node.js at this point. Linux and OSX are recommended, but you should also be able to use FreeBSD and cygwin (on windows). A full windows port of node.js is in the works, but it is not yet ready for public usage.

The most common way to install node.js is to directly compile it from the downloaded source code. There are also various packages available for different package managers, but since their update frequency varies, I recommend installing from source.

You can get the latest source code from nodejs.org. Use the commands below to download and install v0.4.4:

```
$ wget http://nodejs.org/dist/node-v0.4.4.tar.gz
$ tar -xzf node-v0.4.4.tar.gz
$ cd node-v0.4.4
$ ./configure
$ sudo make install
```

Node.js itself has no external dependencies except common build tools as well as pythons for the build system itself. On OSX you must install XCode for this to work, and on Ubuntu you probably have to run:

```
$ apt-get -y install build-essential
```

The interactive node.js shell

If everything worked, you should be able to invoke the interactive node.js shell like this:

```
$ node
> console.log('Hello World');
Hello World
```

The interactive shell (also called REPL) is a great place to test simple one liners, and can also be directly embedded into your node.js applications. In order to get out of it, simply press Ctrl + C.

The REPL also comes with many other great features, most importantly tab auto-completion.

Your first program

Writing a node.js program is as simple as creating a new file with a '.js' extension. For example you could create a simple 'hello_world.js' file with the following content:

```
console.log('Hello World');
```

After you have saved the file, you can execute it from your terminal like so:

```
$ node hello.js
Hello World
```

A hello world http server

Now printing hello world to a terminal isn't all that exciting. Let's take the next step and write a program that responds to hello world via http. We'll call the file 'hello_http.js' and put the following code into it:

```
var http = require('http');

var server = http.createServer(function(req, res) {
  res.writeHead(200);
  res.end('Hello Http');
});
server.listen(8080);
```

Now lets run this program from the terminal by typing:

```
$ node hello_http.js
```

The first thing you'll notice is that this program, unlike our first one, doesn't exit right away. That's because a node program will always run until it's certain that no further events are possible. In this case the open http server is the source of events that will keep things going.

Testing the server is as simple as opening a new browser tab, and navigating to the following url: http://localhost:8080/. As expected, you should see a response that reads: 'Hello Http'.

Alternatively, you could also open up a new terminal and use curl to test your server:

```
$ curl localhost:8080
Hello Http
```

Now let's have a closer look at the steps involved in our little program. In the first line,

we include the http core module and assign it to a variable called http. You will find more information on this in the next section about the module system.

Next we create a variable called server by calling `http.createServer`. The argument passed into this call is a closure that is called whenever an http request comes in.

Finally we call `server.listen(8080)` to tell node.js the port on which we want our server to run. If you want to run on port 80, your program needs to be executed as root.

Now when you point your browser to 'localhost:8080', the connection closure is invoked with a `req` and `res` object. The `req` is a readable stream that emits 'data' events for each incoming piece of data (like a form submission or file upload). The `res` object is a writable stream that is used to send data back to the client. In our case we are simply sending a 200 OK header, as well as the body 'Hello Http'.

The module system

In order to structure your program into different files, node.js provides you with a simple module system.

To illustrate the approach, let's create a new file called 'main.js' with the following content:

```
var hello = require('./hello');
hello.world();
```

As you have probably guessed, the `require('./hello')` is used to import the contents from another JavaScript file. The initial './' indicates that the file is located in the same directory as 'main.js'. Also note that you don't have to provide the file extension, as '.js' is assumed by default.

So let's go ahead and create our 'hello.js' file, with the following content:

```
exports.world = function() {
  console.log('Hello World');
}
```

What you notice here, is that we are assigning a property called 'world' to an object called 'exports'. Such an 'exports' object is available in every module, and it is returned whenever the `require` function is used to include the module. If we now go ahead and run our 'main.js' program, we will see the expected output:

```
$ node main.js
Hello World
```

At this point it should also be mentioned that many node users are overwriting the exports object directly like so:

```
module.exports = function() {
  // ...
}
```

As you might have expected, this will directly cause the `require` function to return the assigned function. This is useful if you're doing object oriented programming, where each file exports the constructor of one class.

The next thing you need to know about the module system is how it deals with `require` calls that don't include a relative hint about the location of the included file. Take for example:

```
var http = require('http');
```

What node.js will do in this case, is to first look if there is a core module named http, and since that's the case, return that directly. But what about non-core modules, such as 'mysql'?

```
var mysql = require('mysql');
```

In this case node.js will walk up the directory tree, moving through each parent directory in turn, checking in each to see if there is a folder called 'node_modules'. If such a folder is found, node.js will look into this folder for a file called 'mysql.js'. If no matching file is found and the directory root '/' is reached, node.js will give up and throw an exception.

At this point node.js also considers an additional, mutable list of alternative include directories which are accessible through the `require.paths` array. However, there is intense debate about removing this feature, so you are probably better off ignoring it.

Last but not least, node.js also lets you create an 'index.js' file, which indicates the main include file for a directory. So if you call `require('./foo')`, both a 'foo.js' file as well as an 'foo/index.js' file will be considered, this goes for non-relative includes as well.

Using EventEmitters

Node.js implements the observer pattern using a class called EventEmitter. Whenever there is an object that represents the source of several kinds of events, node.js usually makes the underlaying class inherit from EventEmitter.

Using EventEmitter's is pretty straight-forward. You can listen to a specific event by calling the 'on()' function on your object, providing the name of the event, as well as a callback closure as the parameters. For example:

```
var data = '';
req
  .on('data', function(chunk) {
    data += chunk;
  })
  .on('end', function() {
    console.log('POST data: %s', data);
  })
```

As you can see, the `on()` function also returns a reference to the object it belongs to, allowing you to chain several of such event listeners.

If you're only interested in the first occurrence of an event, you can use the `once()` function instead.

Finally, you can remove event listeners by using the `removeListener` function. Please note that the argument to this function is a reference to the callback you are trying to remove, not the name of the event:

```
var onData = function(chunk) {
  console.log(chunk);
  req.removeListener(onData);
}

req.on('data', onData);
```

The example above is essentially identical to the `once()` function.

Next Steps

Now that you know your node.js basics, you're probably best off by writing a few little programs yourself. The best place to start out is node's api documentation, using it as a source of inspiration for something you want to play with.

Debugging node.js apps

There are many ways to debug your node.js based applications. Personally I prefer to do as little debugging as possible, so I strictly follow the advice of the test driven development guide.

However, if you find yourself in a situation where you need to locate a tricky bug in an existing applications, here are a few approaches that can help.

Using console.log()

The easiest way to understand a problem is by inspecting objects using console.log(). You can either directly pass in objects as parameters:

```
var foo = {bar: 'foobar'};
console.log(foo);
```

Or you can use its sprintf()-like capabilities to format your debug output:

```
var foo = {bar: 'foobar'};
console.log('Hello %s, this is my object: %j', 'World', foo);
```

Using the node debugger

If console.log() isn't your thing, or you think your problem can be better analyzed using breakpoints, the node's built-in debugger is a great choice. You can invoke the debugger by simply calling:

```
$ node debug my_file.js
```

Work in progress, please come back later ...

Using the WebKit Inspector

Work in progress, please come back later ...

Frameworks

If you're new to node.js, you might not want to re-invent the wheel when it comes to parsing POST requests, routing urls or rendering views. In this case, you probably want to use one of the popular web frameworks. This section gives you a quick overview over the popular choices, and my opinionated take on them.

Express

At this point express is probably the go-to framework for most node.js developers. It's relatively mature, and includes the connect (think rack) middleware layer. Included in the package are routing, configuration, a template engine, POST parsing and many other features.

While express is a solid framework, it's currently addressing a much smaller scope than fullstack frameworks like Rails, CakePHP or Django. It's more comparable to Sinatra, and unfortunately doesn't really make a big effort to differentiate itself from its Ruby roots into something that feels natural in JavaScript. Anyhow, short of writing your own framework, it's certainly a great choice at this point.

fab.js

You think you know JavaScript? Think again. Originally inspired by jQuery's chaining, fab.js has taken a very unconventional approach of twisting JavaScript beyond most peoples brain capacity. Each function returns another function, eliminating the need for method names altogether, while giving the resulting code a lisp-esque look & feel.

At this point I don't consider fab.js production-ready, but if you're still exploring the world of node.js, you should absolutely try it out at least once. If nothing else, fab.js shows the world that JavaScript doesn't have to copy Ruby, Python or PHP when it comes to web frameworks, and can go its own unique ways.

Hosting & Deployment

Quick & Dirty Deployment

If you have just written your first node.js application, and you want to get it running as fast as possible, this is how to do it:

1. Copy your program to the server you want to run it on. If you're using git, this probably just means to clone the repository from another server or service like GitHub.

2. Assuming your project contains a 'server.js' file, navigate to the directory containing it, then type:

```
$ screen
$ node server.js
```

This invokes your 'server.js' program inside a so called screen session. Screen is a tool

that provides you with a shell that remains its state, even when you close the terminal app you used to login to your server.

So you can now safely close your terminal app, and your 'server.js' will continue running. If you want to monitor it, you can log into your server again, and type:

```
$ screen -r
```

This will reconnect you to the backgrounded shell running your program.

However, this approach is only recommended for experimental deployments. If your node applications crashes at some point, screen will not try to restart it, so don't use this method for production applications.

Joyent no.de

Work in progress, please come back later ...