The Node Beginner Book

A Node.js tutorial by Manuel Kiessling

About

The aim of this document is to get you started with developing applications with Node.js, teaching you everything you need to know about "advanced" JavaScript along the way. It goes way beyond your typical "Hello World" tutorial.

Status

You are reading the final version of this book, i.e., updates are only done to correct errors or to reflect changes in new versions of Node.js. It was last updated on January 3, 2015.

The code samples in this book are tested to work with Node.js version 0.10.35.

This site allows you to read pages the first 20 pages of this book for free. The complete text is available as a DRM-free eBook (PDF, ePub and Kindle format). More info is available at the end of the free part.

Intended audience

This document will probably fit best for readers that have a background similar to my own: experienced with at least one object-oriented language like Ruby, Python, PHP or Java, only little experience with JavaScript, and completely new to Node.js.

Aiming at developers that already have experience with other programming languages means that this document won't cover really basic stuff like data types, variables, control structures and the likes. You already need to know about these to understand this document.

However, because functions and objects in JavaScript are different from their counterparts in most other languages, these will be explained in more detail.

Structure of this document

Upon finishing this document, you will have created a complete web application

which allows the users of this application to view web pages and upload files.

Which, of course, is not exactly world-changing, but we will go some extra miles and not only create the code that is "just enough" to make these use cases possible, but create a simple, yet complete framework to cleanly separate the different aspects of our application. You will see what I mean in a minute.

We will start with looking at how JavaScript development in Node.js is different from JavaScript development in a browser.

Next, we will stay with the good old tradition of writing a "Hello World" application, which is a most basic Node.js application that "does" something.

Then, we will discuss what kind of "real" application we want to build, dissect the different parts which need to be implemented to assemble this application, and start working on each of these parts step-by-step.

As promised, along the way we will learn about some of the more advanced concepts of JavaScript, how to make use of them, and look at why it makes sense to use these concepts instead of those we know from other programming languages.

The source code of the finished application is available through the NodeBeginnerBook Github repository.

Table of contents

Chapters available in the full book:

JavaScript and Node.js

JavaScript and You

Before we talk about all the technical stuff, let's take a moment and talk about you and your relationship with JavaScript. This chapter is here to allow you to estimate if reading this document any further makes sense for you.

If you are like me, you started with HTML "development" long ago, by writing HTML documents. You came along this funny thing called JavaScript, but you only used it in a very basic way, adding interactivity to your web pages every now and then.

What you really wanted was "the real thing", you wanted to know how to build complex web sites - you learned a programming language like PHP, Ruby, Java, and started writing "backend" code.

Nevertheless, you kept an eye on JavaScript, you saw that with the introduction of jQuery, Prototype and the likes, things got more advanced in JavaScript land, and that this language really was about more than *window.open()*.

However, this was all still frontend stuff, and although it was nice to have jQuery at your disposal whenever you felt like spicing up a web page, at the end of the day you were, at best, a JavaScript *user*, but not a JavaScript *developer*.

And then came Node.js. JavaScript on the server, how cool is that?

You decided that it's about time to check out the old, new JavaScript. But wait, writing Node.js applications is the one thing; understanding why they need to be written the way they are written means - understanding JavaScript. And this time for real.

Here is the problem: Because JavaScript really lives two, maybe even three lives (the funny little DHTML helper from the mid-90's, the more serious frontend stuff like jQuery and the likes, and now server-side), it's not that easy to find information that helps you to learn JavaScript the "right" way, in order to write Node.js applications in a fashion that makes you feel you are not just using JavaScript, you are actually developing it.

Because that's the catch: you already are an experienced developer, you don't want to learn a new technique by just hacking around and mis-using it; you want to be sure that you are approaching it from the right angle.

There is, of course, excellent documentation out there. But documentation alone sometimes isn't enough. What is needed is guidance.

My goal is to provide a guide for you.

A word of warning

There are some really excellent JavaScript people out there. I'm not one of them.

I'm really just the guy I talked about in the previous paragraph. I know a thing or two about developing backend web applications, but I'm still new to "real" JavaScript and still new to Node.js. I learned some of the more advanced aspects of JavaScript just recently. I'm not experienced.

Which is why this is no "from novice to expert" book. It's more like "from novice to advanced novice".

If I don't fail, then this will be the kind of document I wish I had when starting with Node.js.

Server-side JavaScript

The first incarnations of JavaScript lived in browsers. But this is just the context. It defines what you can do with the language, but it doesn't say much about what the language itself can do. JavaScript is a "complete" language: you can use it in many contexts and achieve everything with it you can achieve with any other "complete" language.

Node.js really is just another context: it allows you to run JavaScript code in the backend, outside a browser.

In order to execute the JavaScript you intend to run in the backend, it needs to be interpreted and, well, executed. This is what Node.js does, by making use of Google's V8 VM, the same runtime environment for JavaScript that Google Chrome uses.

Plus, Node.js ships with a lot of useful modules, so you don't have to write everything from scratch, like for example something that outputs a string on the console.

Thus, Node.js is really two things: a runtime environment and a library.

In order to make use of these, you need to install Node.js. Instead of repeating the process here, I kindly ask you to visit the official installation instructions. Please come back once you are up and running.

"Hello World"

Ok, let's just jump in the cold water and write our first Node.js application: "Hello World".

Open your favorite editor and create a file called *helloworld.js*. We want it to write "Hello World" to STDOUT, and here is the code needed to do that:

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

Save the file, and execute it through Node.js:

```
node helloworld.js
```

This should output *Hello World* on your terminal.

Ok, this stuff is boring, right? Let's write some real stuff.

A full blown web application with Node.js

The use cases

Let's keep it simple, but realistic:

- The user should be able to use our application with a web browser
- The user should see a welcome page when requesting http://*domain*/start which displays a file upload form
- By choosing an image file to upload and submitting the form, this image should then be uploaded to http://*domain*/upload, where it is displayed once the upload is finished

Fair enough. Now, you could achieve this goal by googling and hacking together *something*. But that's not what we want to do here.

Furthermore, we don't want to write only the most basic code to achieve the goal, however elegant and correct this code might be. We will intentionally add more abstraction than necessary in order to get a feeling for building more complex Node.js applications.

The application stack

Let's dissect our application. Which parts need to be implemented in order to fulfill the use cases?

- We want to serve web pages, therefore we need an **HTTP server**
- Our server will need to answer differently to requests, depending on which URL the request was asking for, thus we need some kind of **router** in order to map requests to request handlers
- To fullfill the requests that arrived at the server and have been routed using the router, we need actual **request handlers**
- The router probably should also treat any incoming POST data and give it to the request handlers in a convenient form, thus we need **request data handling**
- We not only want to handle requests for URLs, we also want to display content when these URLs are requested, which means we need some kind of **view logic** the request handlers can use in order to send content to the user's browser
- Last but not least, the user will be able to upload images, so we are going to need some kind of **upload handling** which takes care of the details

Let's think a moment about how we would build this stack with PHP. It's not exactly a secret that the typical setup would be an Apache HTTP server with mod_php5 installed.

Which in turn means that the whole "we need to be able to serve web pages and receive HTTP requests" stuff doesn't happen within PHP itself.

Well, with node, things are a bit different. Because with Node.js, we not only implement our application, we also implement the whole HTTP server. In fact, our web application and its web server are basically the same.

This might sound like a lot of work, but we will see in a moment that with Node.js, it's not.

Let's just start at the beginning and implement the first part of our stack, the HTTP server.

Building the application stack

A basic HTTP server

When I arrived at the point where I wanted to start with my first "real" Node.js application, I wondered not only how to actually code it, but also how to organize my code.
Do I need to have everything in one file? Most tutorials on the web that teach you how to write a basic HTTP server in Node.js have all the logic in one place. What if I want to make sure that my code stays readable the more stuff I implement?

Turns out, it's relatively easy to keep the different concerns of your code separated, by putting them in modules.

This allows you to have a clean main file, which you execute with Node.js, and clean modules that can be used by the main file and among each other.

So, let's create a main file which we use to start our application, and a module file where our HTTP server code lives.

My impression is that it's more or less a standard to name your main file *index.js*. It makes sense to put our server module into a file named *server.js*.

Let's start with the server module. Create the file *server.js* in the root directory of your project, and fill it with the following code:

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

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

```
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

That's it! You just wrote a working HTTP server. Let's prove it by running and testing it. First, execute your script with Node.js:

```
node server.js
```

Now, open your browser and point it at http://localhost:8888/. This should display a web page that says "Hello World".

That's quite interesting, isn't it. How about talking about what's going on here and leaving the question of how to organize our project for later? I promise we'll get back to it.

Analyzing our HTTP server

Well, then, let's analyze what's actually going on here.

The first line *require*s the *http* module that ships with Node.js and makes it accessible through the variable *http*.

We then call one of the functions the http module offers: *createServer*. This function returns an object, and this object has a method named *listen*, and takes a numeric value which indicates the port number our HTTP server is going to listen on.

Please ignore for a second the function definition that follows the opening bracket of *http.createServer*.

We could have written the code that starts our server and makes it listen at port 8888 like this:

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

var server = http.createServer();
server.listen(8888);
```

That would start an HTTP server listening at port 8888 and doing nothing else

(not even answering any incoming requests).

The really interesting (and, if your background is a more conservative language like PHP, odd looking) part is the function definition right there where you would expect the first parameter of the *createServer()* call.

Turns out, this function definition IS the first (and only) parameter we are giving to the *createServer()* call. Because in JavaScript, functions can be passed around like any other value.

Passing functions around

You can, for example, do something like this:

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

Read this carefully! What we are doing here is, we pass the function *say* as the first parameter to the *execute* function. Not the return value of *say*, but *say* itself!

Thus, *say* becomes the local variable *someFunction* within *execute*, and execute can call the function in this variable by issuing *someFunction()* (adding brackets).

Of course, because *say* takes one parameter, *execute* can pass such a parameter when calling *someFunction*.

We can, as we just did, pass a function as a parameter to another function by its name. But we don't have to take this indirection of first defining, then passing it - we can define and pass a function as a parameter to another function in-place:

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

We define the function we want to pass to *execute* right there at the place where

execute expects its first parameter.

This way, we don't even need to give the function a name, which is why this is called an *anonymous function*.

This is a first glimpse at what I like to call "advanced" JavaScript, but let's take it step by step. For now, let's just accept that in JavaScript, we can pass a function as a parameter when calling another function. We can do this by assigning our function to a variable, which we then pass, or by defining the function to pass in-place.

How function passing makes our HTTP server work

With this knowledge, let's get back to our minimalistic HTTP server:

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

By now it should be clear what we are actually doing here: we pass the *createServer* function an anonymous function.

We could achieve the same by refactoring our code to:

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

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
```

Maybe now is a good moment to ask: Why are we doing it that way?

Event-driven asynchronous callbacks

To understand why Node.js applications have to be written this way, we need to

understand how Node.js executes our code. Node's approach isn't unique, but the underlying execution model is different from runtime environments like Python, Ruby, PHP or Java.

Let's take a very simple piece of code like this:

```
var result = database.query("SELECT * FROM hugetable");
console.log("Hello World");
```

Please ignore for now that we haven't actually talked about connecting to databases before - it's just an example. The first line queries a database for lots of rows, the second line puts "Hello World" to the console.

Let's assume that the database query is really slow, that it has to read an awful lot of rows, which takes several seconds.

The way we have written this code, the JavaScript interpreter of Node.js first has to read the complete result set from the database, and then it can execute the *console.log()* function.

If this piece of code actually was, say, PHP, it would work the same way: read all the results at once, then execute the next line of code. If this code would be part of a web page script, the user would have to wait several seconds for the page to load.

However, in the execution model of PHP, this would not become a "global" problem: the web server starts its own PHP process for every HTTP request it receives. If one of these requests results in the execution of a slow piece of code, it results in a slow page load for this particular user, but other users requesting other pages would not be affected.

The execution model of Node.js is different - there is only one single process. If there is a slow database query somewhere in this process, this affects the whole process - everything comes to a halt until the slow query has finished.

To avoid this, JavaScript, and therefore Node.js, introduces the concept of event-driven, asynchronous callbacks, by utilizing an event loop.

We can understand this concept by analyzing a rewritten version of our problematic code:

```
database.query("SELECT * FROM hugetable", function(rows) {
  var result = rows;
```

```
});
console.log("Hello World");
```

Here, instead of expecting *database.query()* to directly return a result to us, we pass it a second parameter, an anonymous function.

In its previous form, our code was synchronous: *first* do the database query, and only when this is done, *then* write to the console.

Now, Node.js can handle the database request asynchronously. Provided that *database.query()* is part of an asynchronous library, this is what Node.js does: just as before, it takes the query and sends it to the database. But instead of waiting for it to be finished, it makes a mental note that says "When at some point in the future the database server is done and sends the result of the query, then I have to execute the anonymous function that was passed to *database.query()*."

Then, it immediately executes *console.log()*, and afterwards, it enters the event loop. Node.js continuously cycles through this loop again and again whenever there is nothing else to do, waiting for events. Events like, e.g., a slow database query finally delivering its results.

This also explains why our HTTP server needs a function it can call upon incoming requests - if Node.js would start the server and then just pause, waiting for the next request, continuing only when it arrives, that would be highly inefficent. If a second user requests the server while it is still serving the first request, that second request could only be answered after the first one is done - as soon as you have more than a handful of HTTP requests per second, this wouldn't work at all.

It's important to note that this asynchronous, single-threaded, event-driven execution model isn't an infinitely scalable performance unicorn with silver bullets attached. It is just one of several models, and it has its limitations, one being that as of now, Node.js is just one single process, and it can run on only one single CPU core. Personally, I find this model quite approachable, because it allows to write applications that have to deal with concurrency in an efficient and relatively straightforward manner.

You might want to take the time to read Felix Geisendörfer's excellent post Understanding node.js for additional background explanation.

Let's play around a bit with this new concept. Can we prove that our code continues after creating the server, even if no HTTP request happened and the

callback function we passed isn't called? Let's try it:

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

function onRequest(request, response) {
  console.log("Request received.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);

console.log("Server has started.");
```

Note that I use *console.log* to output a text whenever the *onRequest* function (our callback) is triggered, and another text right *after* starting the HTTP server.

When we start this (*node server.js*, as always), it will immediately output "Server has started." on the command line. Whenever we request our server (by opening http://localhost:8888/ in our browser), the message "Request received." is printed on the command line.

Event-driven asynchronous server-side JavaScript with callbacks in action :-)

(Note that our server will probably write "Request received." to STDOUT two times upon opening the page in a browser. That's because most browsers will try to load the favicon by requesting http://localhost:8888/favicon.ico whenever you open http://localhost:8888/).

How our server handles requests

Ok, let's quickly analyze the rest of our server code, that is, the body of our callback function *onRequest()*.

When the callback fires and our *onRequest()* function gets triggered, two parameters are passed into it: *request* and *response*.

Those are objects, and you can use their methods to handle the details of the HTTP request that occured and to respond to the request (i.e., to actually send something over the wire back to the browser that requested your server).

And our code does just that: Whenever a request is received, it uses the *response.writeHead()* function to send an HTTP status 200 and content-type in the HTTP response header, and the *response.write()* function to send the text

"Hello World" in the HTTP response body.

At last, we call *response.end()* to actually finish our response.

At this point, we don't care for the details of the request, which is why we don't use the *request* object at all.

Finding a place for our server module

Ok, I promised we will get back to how to organize our application. We have the code for a very basic HTTP server in the file *server.js*, and I mentioned that it's common to have a main file called *index.js* which is used to bootstrap and start our application by making use of the other modules of the application (like the HTTP server module that lives in *server.js*).

Let's talk about how to make server.js a real Node.js module that can be used by our yet-to-be-written *index.js* main file.

As you may have noticed, we already used modules in our code, like this:

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

...

http.createServer(...);
```

Somewhere within Node.js lives a module called "http", and we can make use of it in our own code by requiring it and assigning the result of the require to a local variable.

This makes our local variable an object that carries all the public methods the *http* module provides.

It's common practice to choose the name of the module for the name of the local variable, but we are free to choose whatever we like:

```
var foo = require("http");

...

foo.createServer(...);
```

Fine, it's clear how to make use of internal Node.js modules. How do we create our

own modules, and how do we use them?

Let's find out by turning our *server.js* script into a real module.

Turns out, we don't have to change that much. Making some code a module means we need to *export* those parts of its functionality that we want to provide to scripts that require our module.

For now, the functionality our HTTP server needs to export is simple: scripts requiring our server module simply need to start the server.

To make this possible, we will put our server code into a function named *start*, and we will export this function:

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

function start() {
  function onRequest(request, response) {
    console.log("Request received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

This way, we can now create our main file *index.js*, and start our HTTP there, although the code for the server is still in our *server.js* file.

Create a file *index.js* with the following content:

```
var server = require("./server");

server.start();
```

As you can see, we can use our server module just like any internal module: by requiring its file and assigning it to a variable, its exported functions become available to us.

That's it. We can now start our app via our main script, and it still does exactly the same:

```
node index.js
```

Great, we now can put the different parts of our application into different files and wire them together by making them modules.

We still have only the very first part of our application in place: we can receive HTTP requests. But we need to do something with them - depending on which URL the browser requested from our server, we need to react differently.

For a very simple application, you could do this directly within the callback function *onRequest()*. But as I said, let's add a bit more abstraction in order to make our example application a bit more interesting.

Making different HTTP requests point at different parts of our code is called "routing" - well, then, let's create a module called *router*.

What's needed to "route" requests?

We need to be able to feed the requested URL and possible additional GET and POST parameters into our router, and based on these the router then needs to be able to decide which code to execute (this "code to execute" is the third part of our application: a collection of request handlers that do the actual work when a request is received).

So, we need to look into the HTTP requests and extract the requested URL as well as the GET/POST parameters from them. It could be argued if that should be part of the router or part of the server (or even a module of its own), but let's just agree on making it part of our HTTP server for now.

All the information we need is available through the *request* object which is passed as the first parameter to our callback function *onRequest()*. But to interpret this information, we need some additional Node.js modules, namely *url* and *querystring*.

The *url* module provides methods which allow us to extract the different parts of a URL (like e.g. the requested path and query string), and *querystring* can in turn be used to parse the query string for request parameters:

```
                           url.parse(string).query
                                      |
           url.parse(string).pathname |
                       |              |
```

```
                          |                    |
                        ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                                    ---       -----
                                     |          |
                                     |          |
         querystring.parse(string)["foo"]       |
                                                |
                     querystring.parse(string)["hello"]
```

We can, of course, also use *querystring* to parse the body of a POST request for parameters, as we will see later.

Let's now add to our *onRequest()* function the logic needed to find out which URL path the browser requested:

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Fine. Our application can now distinguish requests based on the URL path requested - this allows us to map requests to our request handlers based on the URL path using our (yet to be written) router.

In the context of our application, it simply means that we will be able to have requests for the */start* and */upload* URLs handled by different parts of our code. We will see how everything fits together soon.

Ok, it's time to actually write our router. Create a new file called *router.js*, with the following content:

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

Of course, this code basically does nothing, but that's ok for now. Let's first see how to wire together this router with our server before putting more logic into the router.

Our HTTP server needs to know about and make use of our router. We could hard-wire this dependency into the server, but because we learned the hard way from our experience with other programming languages, we are going to loosely couple server and router by injecting this dependency (you may want to read Martin Fowlers excellent post on Dependency Injection for background information).

Let's first extend our server's *start()* function in order to enable us to pass the route function to be used by parameter:

```
var http = require("http");
var url = require("url");

function start(route) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

And let's extend our *index.js* accordingly, that is, injecting the route function of our router into the server:

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

Again, we are passing a function, which by now isn't any news for us.

If we start our application now (*node index.js, as always*), and request an URL, you can now see from the application's output that our HTTP server makes use of our router and passes it the requested pathname:

```
bash$ node index.js
Request for /foo received.
About to route a request for /foo
```

(I omitted the rather annoying output for the /favicon.ico request).

Execution in the kingdom of verbs

May I once again stray away for a while and talk about functional programming again?

Passing functions is not only a technical consideration. With regard to software design, it's almost philosophical. Just think about it: in our index file, we could have passed the *router* object into the server, and the server could have called this object's *route* function.

This way, we would have passed a *thing*, and the server would have used this thing to *do* something. Hey, router thing, could you please route this for me?

But the server doesn't need the thing. It only needs to get something *done*, and to get something done, you don't need things at all, you need *actions*. You don't need *nouns*, you need *verbs*.

Understanding the fundamental mind-shift that's at the core of this idea is what made me really understand functional programming.

And I did understand it when reading Steve Yegge's masterpiece Execution in the Kingdom of Nouns. Go read it now, really. It's one of the best writings related to software I ever had the pleasure to encounter.

Routing to real request handlers

Back to business. Our HTTP server and our request router are now best friends and talk to each other as we intended.

Of course, that's not enough. "Routing" means, we want to handle requests to different URLs differently. We would like to have the "business logic" for requests to */start* handled in another function than requests to */upload*.

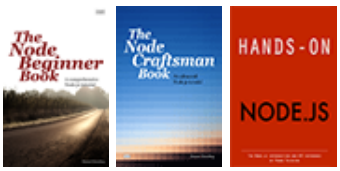
Right now, the routing "ends" in the router, and the router is not the place to actually "do" something with the requests, because that wouldn't scale well once

our application becomes more complex.

Let's call these functions, where requests are routed to, *request handlers*. And let's tackle those next, because unless we have these in place there isn't much sense in doing anything with the router for now.

All three books together would cost a total of $28.97, but we are offering them as a bundle for only **$9.99**. You can download them immediately, they are completely DRM-free, and you will receive any future updates to all three books for free.

Buy this bundle now



291 pages in total
100% DRM-free
Free lifetime updates
PDF, Kindle, ePub
Only $9.99