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 June 5, 2017.

The code samples in this book are tested to work with both the Long Term Support version 6.10.3 as well as the most current 8.0.0 version of Node.js.

This site allows you to read the first 19 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 across 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 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 page. 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 fulfill 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_php 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! 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.

Hi there! Sorry to interrupt you.

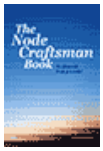
My name is Manuel Kiessling, I'm the author of this book.

If you have read that far, I would like you to consider buying the eBook version of this book. It's a beautifully crafted package including a PDF, an ePub, and a MOBI file, which means you can read it on all kinds of eReaders out there like the Amazon Kindle, the iPad, or the Sony Reader, and of course on any PC or Mac.

But the best thing is that you can buy it bundled together with another great Node.js book at a very attractive discounted price:

The full version of *The Node Beginner Book*, giving you access to all 54 pages of this tutorial, where I talk about blocking and non-blocking operations, handling POST requests and file uploads, and how to finalize the example application into a working whole.

+



The Node Craftsman Book is the offical follow up to The Node Beginner Book. On 170 pages, it covers the following topics:

- Working with NPM and Packages
- Object-oriented JavaScript
- Test-Driven Node.js Development
- Synchronous and Asynchronous operations explained
- Using and creating Event Emitters
- Node.js and MySQL
- Node.js and MongoDB
- Writing fast and efficient code
- Writing a REST webservice application
- Combining Node.js and AngularJS
- Setting up a continuous deployment workflow

Both books together would cost a total of $29.99, but for a limited time, we are offering them as a bundle for only **$9**. You can download them immediately, they are completely DRM-free, and you will receive any future updates to both books for free.

> The object-oriented chapter of #nodecraftsman by @manuelkiessling makes OOP #JavaScript click more than any other explanation I've read.

Nick Strayer
via Twitter

> Thanks to @manuelkiessling for writing the #nodebeginner and #nodecraftsman books. Great intros to a new language for this PHP dev.

Erorus
via Twitter

> "Test-Driven development with Node.js", best chapter in the history of chapters!

Benjamin Todts
via Twitter

> Just starting to read the #nodecraftsman book to learn about node.js! I read #nodebeginner by the same author yesterday, and it was awesome!

Heidi Newton
via eMail
