Getting started with Node.js

> This article was originally published at https://gist.github.com/joepie91/95ed77b71790442b7e61. Some of the links in it still point to Gists that I have written; these will be moved over and relinked in due time.

> Some of the suggestions on this page have become outdated, and better alternatives are available nowadays. However, the suggestions listed here *should still work today* as they did when this article was originally written. You do not *need* to update things to new approaches, and sometimes the newer approaches actually aren't better either, they can even be worse!

"*How do I get started with Node?*" is a commonly heard question in #Node.js. This gist is an attempt to compile some of the answers to that question. It's a perpetual work-in-progress.

And if this list didn't quite answer your questions, I'm available for tutoring and code review! A donation is also welcome :)

Setting expectations

Before you get started learning about JavaScript and Node.js, there's one very important article you need to read: Teach Yourself Programming in Ten Years.

Understand that **it's going to take time** to learn Node.js, just like it would take time to learn any other specialized topic - and that you're not going to learn effectively just by reading things, or following tutorials or courses. **Get out there and build things!** Experience is by far the most important part of learning, and shortcuts to this simply *do not exist*.

Avoid "bootcamps", courses, extensive books, and basically anything else that claims to teach you programming (or Node.js) in a single run. They all lie, and what they promise you simply isn't possible. That's also the reason this post is a *list of resources*, rather than a single book - they're references for when you need to learn about a certain topic at a certain point in time. Nothing more, nothing less.

There's also no such thing as a "definitive guide to Node.js", or a "perfect stack". Every project is going to have different requirements, that are best solved by different tools. There's no point in trying to learn everything upfront, because *you can't know what you need to learn, until you actually need it*.

In conclusion, the best way to get started with Node.js is to simply **decide on a project you want to build, and start working on it**. Start with the simplest possible implementation of it, and over time add bits and pieces to it, learning about those bits and pieces as you go. The links in this post will help you with that.

> You'll find a table of contents for this page on your left.

Javascript refresher

Especially if you normally use a different language, or you only use Javascript occasionally, it's easy to misunderstand some of the aspects of the language.

These links will help you refresh your knowledge of JS, and make sure that you understand the OOP model correctly.

- **A whirlwind tour of the language:** http://learnxinyminutes.com/docs/javascript/
- **Javascript is asynchronous, through using an 'event loop'.** This video explains what an event loop *is*, and this video goes into more detail about how it works and how to deal with corner cases. If you're not familiar with the event loop yet, you should watch both.
- **Javascript does automatic typecasting ("type conversion") in some cases.** This shows how various values cast to a boolean, and this shows how `null` and `undefined` relate to each other.
- **In Javascript, braces are optional for single-line statements - however, you should *always* use them.** This gist demonstrates why.
- **Asynchronous execution in Javascript is normally implemented using CPS.** This stands for "continuation-passing style", and this shows an example of how that works.
- **However, in practice, you shouldn't use that, and you should use Promises instead.** Whereas it is very easy to mess up CPS code, that is not an issue with Promises - error handling is much more reliable, for example. This guide should give you a decent introduction.
- **A callback should be either consistently synchronous, or consistently asynchronous.** You don't really have to worry about this when you're using Promises (as they ensure that this is consistent), but this article still has a good explanation of the reasons for this. A simpler example can be found here.
- **Javascript does not have classes, and constructor functions are a bad idea.** This short article will help you understand the prototypical OOP model that Javascript uses. This gist shows a brief example of what the `this` variable refers to. Often you don't need inheritance at all - this gist shows an example of creating an object in the simplest possible way.

- **In Javascript, closures are everywhere, by default.** This gist shows an example.

The Node.js platform

Node.js is not a language. Rather, it's a "runtime" that lets you run Javascript without a browser. It comes with some basic additions such as a TCP library - or rather, in Node.js-speak, a "TCP module" - that you need to write server applications.

- **The easiest way to install Node.js on Linux and OS X, is to use `nvm`.** The instructions for that can be found here. Make sure you create a `default` alias (as explained in the documentation), if you want it to work like a 'normal' installation.
- **If you are using Windows:** You can download an installer from the Node.js website. You should consider using a different operating system, though - Windows is generally rather poorly suited for software development outside of .NET. Things will be a lot easier if you use Linux or OS X.
- **The package manager you'll use for Node.js, is called NPM.** While it's very simple to use, it's not particularly well-documented. This article will give you an introduction to it.
- **Don't hesitate to add dependencies, even small ones!** Node.js and NPM are specifically designed to make this possible without running into issues, and you will get big benefits from doing so. This post explains more about that.
- **The module system is very simple.** The Node.js documentation explains this further.
- **MongoDB is commonly recommended and used with Node.js. It is, however, extremely poorly designed - and you shouldn't use it.** This article goes into more detail about *why* you shouldn't use it. If you're not sure what to use, use PostgreSQL.
- The rest of the documentation for all the modules included with Node.js, can be found here.

Setting up your environment

- **To be able to install "native addons" (compiled C++ modules), you need to take some additional steps.** If you are on Linux or OS X, you likely already have everything you need - however, on Windows you'll have to install a few additional pieces of software. The instructions for all of these platforms can be found here. **Do not skip this step.** Installing pure-Javascript modules is not always a viable solution, especially where it concerns cryptography-related modules such as `scrypt` or `bcrypt`.
- **If you're running into issues on Windows,** try these instructions from Microsoft.
- **There are a lot of build tools for helping you manage your code.** It can get a bit confusing, though - there are a lot of articles that just tell you to combine a pile of different tools, without ever explaining what they're for. This is a hype-free overview of different kinds of build tools, and what they may be useful for.

Functional programming

Javascript has part of its roots in functional programming languages, which means that you can use some of those concepts in your own projects. They can be greatly beneficial to the readability and maintainability of your code.

- This article gives an introduction to `map`, `filter` and `reduce` - three functional programming operations that help a *lot* in writing maintainable and predictable code.
- This gist shows an example of using those with Bluebird, the Promises library that I recommended in the Promises Reading Guide.
- This slide deck demonstrates currying in Javascript, another functional programming technique - think of them as "partially executed functions".

Module patterns

To build "configurable" modules, you can use a pattern known as "parametric modules". This gist shows an example of that. This is another example.

A commonly used pattern is the `EventEmitter` - this is exactly what it sounds like; an object that emits events. It's a very simple abstraction, but helps greatly in writing loosely coupled code. This gist illustrates the object, and the full documentation can be found here.

Code architecture

The 'design' of your codebase matters a lot. Certain approaches for solving a problem work better than other approaches, and each approach has its own set of benefits and drawbacks. Picking the right approach is important - it will save you hours (or days!) of time down the line, when you are maintaining your code.

I'm still in the process of writing more about this, but so far, I've already written an article that explains the difference between monolithic and modular code and why it matters. You can read it here.

Express

If you want to build a website or web application, you'll probably find Express to be a good framework to start with. As a framework, it is *very* small. It only provides you with the basic necessities - everything else is a plugin.

If this sounds complicated, don't worry - things almost always work "out of the box". Simply follow the `README` for whichever "middleware" (Express plugin) you want to add.

To get started with Express, simply follow the below articles. Whatever you do, don't use the Express Generator - it generates confusing and bloated code. Just start from scratch and follow the guides!

- Installing Express (some of this was already covered in the NPM guide above)
- A Hello World example
- Routing
- Using template engines
- Writing Middleware
- Using Middleware
- Static File Handling (this is middleware, too!)
- Error Handling
- Debugging

To get a better handle on how to render pages server-side with Express:

- Rendering pages server-side with Express (and Pug) (a step-by-step walkthrough, work in progress)

Some more odds and ends regarding about Express:

- Some FAQs (don't use MVC, however - this is why.)
- Express Behind Proxies
- The full Express API documentation

Some examples:

- Making something "globally available" in an Express application
- Writing configurable middleware (using the same technique as the parametric modules I showed earlier)

Combining Express and Promises:

- A short article explaining how to use `express-promise-router`
- An example, also explaining what would happen if you didn't handle errors.

Some common Express middleware that you might want to use:

- **Sessions:** express-session, with connect-session-knex if you are using Knex.
- **Message flashing:** connect-flash
- **Handling request payloads ("form/POST data"):** body-parser
- **Handling uploads and other multipart data:** multer if you want it written to disk like PHP would do, or connect-busboy if you want to interact with the upload stream directly.
- **Access logs:** morgan
- **OAuth/OpenID integration:** Passport

Coming from other languages or platforms

- **If you are used to PHP or similar:** Contrary to PHP, Node.js does *not* use a CGI-like model (ie. "one pageload is one script"). Instead, it is a persistent process - your code *is* the webserver, and it handles many incoming requests at the same time, for as long as the process keeps running. This means you can have persistent state - this gist shows an example of that.
- **If you are used to synchronous platforms:** This gist illustrates the differences between a (synchronous) PHP script and an (asynchronous) Node.js application.

Security

Note that this advice isn't necessarily complete. It answers some of the most common questions, but your project might have special requirements or caveats. When in doubt, you can always ask in the #Node.js channel!

Also, keep in mind the golden rule of security: humans *suck* at repetition, regardless of their level of competence. **If a mistake *can* be made, then it *will* be made.** Design your systems such that they are hard to use incorrectly.

- **Sessions:** Use something that implements session cookies. If you're using Express, express-session will take care of this for you. Whatever you do, **don't use JWT for sessions**, even if many blog posts recommend it - it will cause security problems. This article goes into more detail.
- **Password hashing:** Use `scrypt`. This wrapper module will make it easier to use.
- **CSRF protection:** You need this if you are building a website. Use csurf.
- **XSS:** Every good templater will escape output by default. **Only** use templaters that do this (such as Jade or Nunjucks)! If you need to explicitly escape things, you should consider it insecure - it's too easy to forget to do this, and is practically guaranteed to result in vulnerabilities.

- **SQL injection:** Always use parameterized queries. When using MySQL, use the `node-mysql2` module instead of the `node-mysql` module - the latter doesn't use real parameterized queries. Ideally, use something like Knex, which will also prevent many other issues, and make your queries much more readable and maintainable.
- **Random numbers and values:** Generating unpredictable random numbers is a lot harder than it seems. `Math.random()` will generate numbers that may *seem* random, but are actually quite predictable to an attacker. If you need random values, read this article for recommendations. It also goes into more detail about the types of "randomness" that exist.
- **Cryptography:** Follow the suggestions in this gist. Whatever you do, **do not use the `crypto` module directly**, unless you really have no other choice. Never use pure-Javascript reimplementations - always use bindings to the original implementation, where possible (in the form of native addons).
- **Vulnerability advisories:** The Node Security Project keeps track of known vulnerabilities in Node.js modules. Services like VersionEye will e-mail you, if your project uses a module that is found vulnerable.

Useful modules:

This is an incomplete list, and I'll probably be adding stuff to it in the future.

- **Determining the type of a value:** type-of-is
- **Date/time handling:** Moment.js
- **Making HTTP requests:** bhttp
- **Clean debugging logs:** debug
- **Cleaner stacktraces and errors:** pretty-error
- **Markdown parsing:** marked
- **HTML parsing:** cheerio (has a jQuery-like API)
- **WebSockets:** ws

Deployment

- **Don't run Node.js as root, ever!** If you want to expose your service at a privileged port (eg. port 80), and you probably do, then you can use authbind to accomplish that safely.

Distribution

- **Your project is ready for release!** But... you should still pick a license. This article will give you a very basic introduction to copyright, and the different kind of (common) licenses you can use.

Scalability

Scalability is a result of your application architecture, not the technologies you pick. Be wary of anything that claims to be "scalable" - it's much more important to write loosely coupled code with small components, so that you can split out responsibilities across multiple processes and servers.

Troubleshooting

Is something not working properly? Here are some resources that might help:

- Is `npm install` causing an error? Use this error explaining tool to find out what's wrong.
- `DeprecationWarning: Using Buffer without new will soon stop working.` - the solution for this can be found here.

Optimization

The first rule of optimization is: **do not optimize.**

The correct order of concerns is security first, then maintainability/readability, and *then* performance. Optimizing performance is something that you shouldn't care about, until you have *hard metrics* showing you that it is needed. If you can't show a performance problem in numbers, it doesn't exist; while it is easy to optimize readable code, it's much harder to make optimized core more readable.

There is one exception to this rule: *never* use any methods that end with `Sync` - these are blocking, synchronous methods, and will block your event loop (ie. your entire application) until they have completed. They may look convenient, but they are not worth the performance penalty.

Now let's say that you *are* having performance issues. Here are some articles and videos to learn more about how optimization and profiling works in Node.js / V8 - they are going to be fairly in-depth, so you may want to hold off on reading these until you've gotten some practice with Node.js:

- Common causes of deoptimization
- Monomorphism, and why it is important
- Tuning Node.js
- A tour of V8: object representation

- Node.js in flames
- Realtime Node.js App: A Stress Testing Story (using Socket.IO)
- A bigger list of resources about V8 optimization and internals can be found here.

If you're seeing memory leaks, then these may be helpful articles to read:

- Three kinds of memory leaks

These are some modules that you may find useful for profiling your application:

- **node-inspector:** Based on Chrome Developer Tools, this tool gives you many features, including CPU and heap profiling. Also useful for debugging in general. **Since Node.js v6.3.0, you can also connect directly using Chrome Developer Tools.**
- **heapdump:** On-demand heap dumps, for later analysis. Usable from application code *in production*, so very useful for making a heap dump the moment your application goes over a certain heap size.
- **memwatch-next:** Provides memory leak detection, and heap diffing.

Writing C++ addons

You'll usually want to avoid this - C++ is not a memory-safe language, so it's much safer to just write your code in Javascript. V8 is rather well-optimized, so in most cases, performance isn't a problem either. That said, sometimes - eg. when writing bindings to something else - you just *have* to write a native module.

These are some resources on that:

- The addon documentation
- `nan`, an abstraction layer for making your module work across Node.js versions (you should absolutely use this)
- `node-gyp`, the build tool you will need for this purpose
- V8 API documentation for every supported Node.js version

Writing Rust addons

Neon is a new project that lets you write **memory-safe compiled extensions** for Node.js, using Rust. It's still pretty new, but quite promising - an introduction can be found here.

Odds and ends

Some miscellaneous code snippets and examples, that I haven't written a section or article for yet.

- **Named logging in Gulp:** https://gist.github.com/joepie91/e7d66ffdb17d1ea69c56
- **Cached image:** https://gist.github.com/joepie91/cee42198b6bc6a24ea44
- **Combining Gulp and Electron:**
  https://gist.github.com/joepie91/f81cdbc1b45d52ab4b87

Future additions to this list

There are a few things that I'm currently working on documenting, that will be added to this list in the future. I write new documentation as I find the time to do so.

- **Node.js for PHP developers** (a migration guide) - In progress.
- **A comprehensive guide to Promises** - Planned.
- **A comprehensive guide to streams** - Planned.
- **Error handling mechanisms and strategies** - Planned.
- **Introduction to HTTP** - Planned.
- **Writing a secure authentication system** - Planned.
- **Writing abstractions** - Planned.

Revision #1
Created 2024-12-11 01:33:07 UTC by joepie91
Updated 2024-12-11 16:50:35 UTC by joepie91