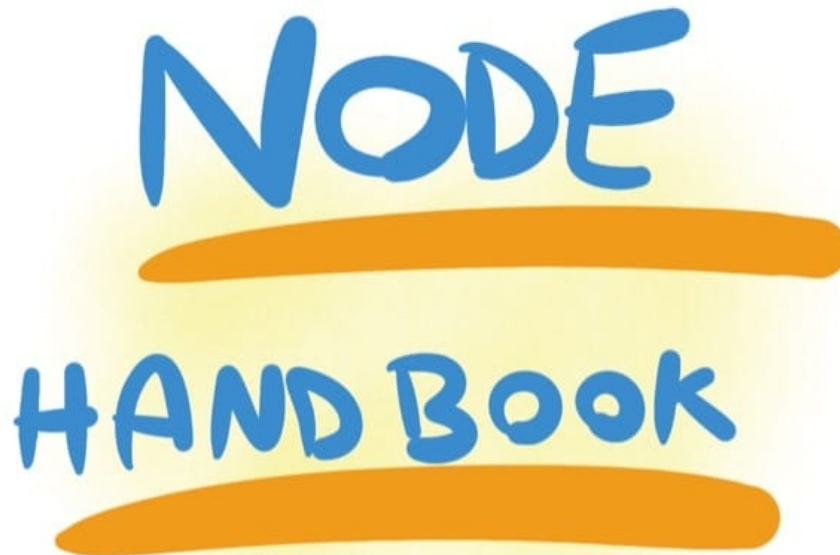


The Node.js Handbook



The Node Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

I find this approach gives a well-rounded overview. This book does not try to cover everything under the sun related to Node. If you think some specific topic should be included, tell me.

You can reach me on Twitter [@flaviocopes](#).

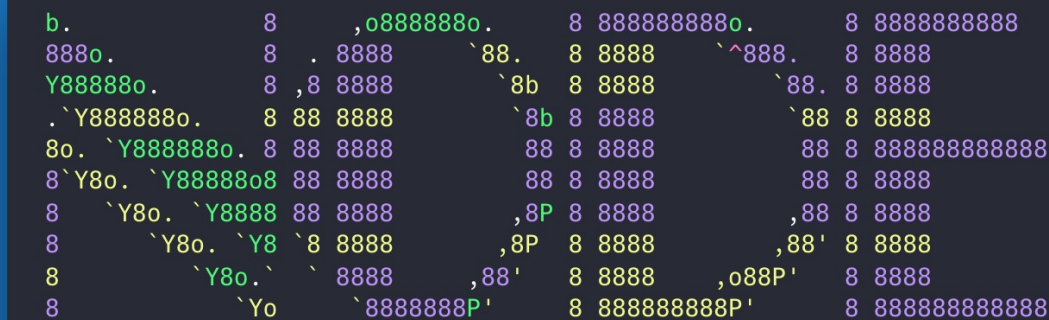
I hope the contents of this book will help you achieve what you want: **learn the basics Node.js**.

This book is written by Flavio. I **publish web development tutorials** every day on my website [flaviocopes.com](#).

Enjoy!

Introduction to Node

This post is a getting started guide to Node.js, the server-side JavaScript runtime environment. Node.js is built on top of the Google Chrome V8 JavaScript engine, and it's mainly used to create web servers - but it's not limited to that



```
b.      8      ,o888888o.      8 8888888888o.      8 8888888888
888o.    8      . 8888      `88.    8 8888      `^888.    8 8888
Y888888o. 8 ,8 8888      `8b 8 8888      `88. 8 8888
. `Y888888o. 8 88 8888      `8b 8 8888      `88 8 8888
8o. `Y888888o. 8 88 8888      88 8 8888      88 8 888888888888
8 `Y8o. `Y888888o8 88 8888      88 8 8888      88 8 8888
8 `Y8o. `Y8888 88 8888      ,8P 8 8888      ,88 8 8888
8 `Y8o. `Y8 `8 8888      ,8P 8 8888      ,88' 8 8888
8 `Y8o. ` 8888      ,88' 8 8888      ,o88P' 8 8888
8 `Yo      `8888888P' 8 8888888888P' 8 888888888888
```

- [Overview](#)
- [The best features of Node.js](#)
 - [Fast](#)
 - [Simple](#)
 - [JavaScript](#)
 - [V8](#)
 - [Asynchronous platform](#)
 - [A huge number of libraries](#)
- [An example Node.js application](#)
- [Node.js frameworks and tools](#)

Overview

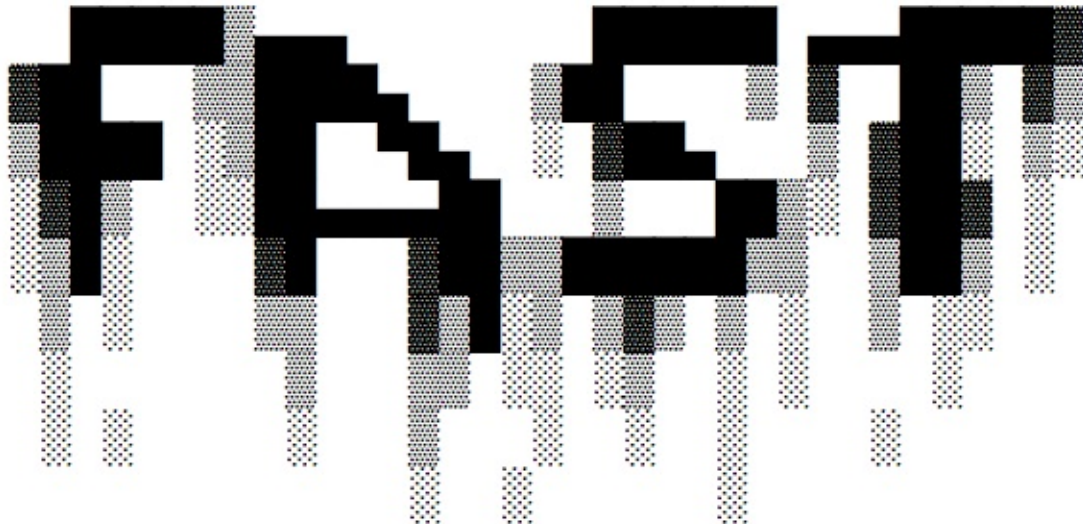
Node.js is a **runtime environment** for **JavaScript** that runs on the **server**.

Node.js is open source, cross-platform, and since its introduction in 2009, it got hugely popular and now plays a significant role in the web development scene. If GitHub stars are one popularity indication factor, having 46000+ stars means being very popular.

Node.js is built on top of the Google Chrome V8 JavaScript engine, and it's mainly used to create web servers - but it's not limited to that.

The best features of Node.js

Fast



One of the main selling points of Node.js is **speed**. JavaScript code running on Node.js (depending on the benchmark) can be twice as fast than compiled languages like C or Java, and orders of magnitude faster than interpreted languages like Python or Ruby, because of its non-blocking paradigm.

Simple

Node.js is simple. Extremely simple, actually.

JavaScript

Node.js runs JavaScript code. This means that millions of frontend developers that already use JavaScript in the browser are able to run the server-side code and frontend-side code using the same programming language without the need to learn a completely different tool.

The paradigms are all the same, and in Node.js the new [ECMAScript](#) standards can be used first, as you don't have to wait for all your users to update their browsers - you decide which ECMAScript version to use by changing the Node.js version.

V8

Running on the [Google V8 JavaScript engine](#), which is Open Source, Node.js is able to leverage the work of thousands of engineers that made (and will continue to make) the Chrome JavaScript runtime blazing fast.

Asynchronous platform



In traditional programming languages (C, Java, Python, PHP) all instructions are blocking by default unless you explicitly "opt in" to perform asynchronous operations. If you perform a network request to read some JSON, the execution of that particular thread is blocked until the response is ready.

JavaScript allows to create asynchronous and non-blocking code in a very simple way, by using a **single thread**, **callback functions** and **event-driven programming**. Every time an expensive operation occurs, we pass a callback function that will be called once we can continue with the processing. We're not waiting for that to finish before going on with the rest of the program.

Such mechanism derives from the browser. We can't wait until something loads from an AJAX request before being able to intercept click events on the page. **It all must happen in real time** to provide a good experience to the user.

If you've created an onclick handler for a web page you've already used asynchronous programming techniques with event listeners.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing threads concurrency, which would be a major source of bugs.

Node provides non-blocking I/O primitives, and generally, libraries in Node.js are written using non-blocking paradigms, making a blocking behavior an exception rather than the normal.

When Node.js needs to perform an I/O operation, like reading from the network, access a database or the filesystem, instead of blocking the thread Node.js will simply resume the operations when the response comes back, instead of wasting CPU cycles waiting.

A huge number of libraries

`npm` with its simple structure helped the ecosystem of node.js proliferate and now the npm registry hosts almost 500.000 open source packages you can freely use.

An example Node.js application

The most common example Hello World of Node.js is a web server:

```
const http = require('http')

const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

This code first includes the Node.js `http` module.

Node.js has an amazing [standard library](#), including a first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the `request event` is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with

```
res.statusCode = 200
```

we set the `statusCode` property to 200, to indicate a successful response.

We set the Content-Type header:

```
res.setHeader('Content-Type', 'text/plain')
```

and we end close the response, adding the content as an argument to `end()` :

```
res.end('Hello World\n')
```

Node.js frameworks and tools

Node.js is a low-level platform, and to make things easier and more interesting for developers thousands of libraries were built upon Node.js.

Many of those established over time as popular options. Here is a non-comprehensive list to the ones I consider very relevant and worth learning:

- **Express**, one of the most simple yet powerful ways to create a web server. Its minimalist approach, unopinionated, focused on the core features of a server, is key to its success.
- **Meteor**, an incredibly powerful full-stack framework, powering you with an isomorphic approach to building apps with JavaScript, sharing code on the client and the server. Once an off-the-shelf tool that provided everything, now integrates with frontend libs React, **Vue** and Angular. Can be used to create mobile apps as well.
- **koa**, built by the same team behind Express, aims to be even simpler and smaller, building on top of years of knowledge. The new project born out of the need to create incompatible changes without disrupting the existing community.
- **Next.js**, a framework to render server-side rendered **React** applications.
- **Micro**, a very lightweight server to create asynchronous HTTP microservices.
- **Socket.io**, a real-time communication engine to build network applications.

A brief history of Node

A look back on the history of Node.js from 2009 to today

Believe it or not, Node.js is just 9 years old.

In comparison, JavaScript is 23 years old and the web as we know it (after the introduction of Mosaic) is 25 years old.

9 years is such a little amount of time for a technology, but Node.js seems to have been around forever.

I've had the pleasure to work with Node since the early days when it was just 2 years old, and despite the little information available, you could already feel it was a huge thing.

In this post, I want to draw the big picture of Node in its history, to put things in perspective.

- [A little bit of history](#)
- [2009](#)
- [2010](#)
- [2011](#)
- [2012](#)
- [2013](#)
- [2014](#)
- [2015](#)
- [2016](#)
- [2017](#)
- [2018](#)

A little bit of history

JavaScript is a programming language that was created at Netscape as a scripting tool to manipulate web pages inside their browser, [Netscape Navigator](#).

Part of the business model of Netscape was to sell Web Servers, which included an environment called *Netscape LiveWire*, which could create dynamic pages using server-side JavaScript. So the idea of server-side JavaScript was not introduced by Node.js, but it's old just like JavaScript - but at the time it was not successful.

One key factor that led to the rise of Node.js was timing. JavaScript since a few years was starting being considered a serious language, thanks for the "Web 2.0" applications that showed the world what a modern experience on the web could be like (think Google Maps or

GMail).

The JavaScript engines performance bar raised considerably thanks to the browser competition battle, which is still going strong. Development teams behind each major browser work hard every day to give us better performance, which is a huge win for JavaScript as a platform. V8, the engine that Node.js uses under the hood, is one of those and in particular it's the Chrome JS engine.

But of course, Node.js is not popular just because of pure luck or timing. It introduced much innovative thinking on how to program in JavaScript on the server.

2009

Node.js is born The first form of [npm](#) is created

2010

[Express](#) is born [Socket.io](#) is born

2011

npm hits 1.0 Big companies start adopting Node: LinkedIn, Uber [Hapi](#) is born

2012

Adoption continues very rapidly

2013

First big blogging platform using Node: Ghost [Koa](#) is born

2014

Big drama: [IO.js](#) is a major fork of Node.js, with the goal of introducing ES6 support and move faster

2015

The [Node.js Foundation](#) is born IO.js is merged back into Node.js npm introduces private modules Node 4 (no 1, 2, 3 versions were previously released)

2016

The [leftpad incident](#) [Yarn](#) is born Node 6

2017

npm focuses more on security Node 8 HTTP/2 [V8](#) introduces Node in its testing suite, officially making Node a target for the JS engine, in addition to Chrome 3 billion npm downloads every week

2018

Node 10 [ES modules](#) .mjs experimental support

How to install Node

How you can install Node.js on your system: a package manager, the official website installer or nvm

Node.js can be installed in different ways. This post highlights the most common and convenient ones.

Official packages for all the major platforms are available at <https://nodejs.org/en/download/>.

One very convenient way to install Node.js is through a package manager. In this case, every operating system has its own.

On macOS, [Homebrew](#) is the de-facto standard, and - once installed - allows to install Node.js very easily, by running this command in the CLI:

```
brew install node
```

Other package managers for Linux and Windows are listed in <https://nodejs.org/en/download/package-manager/>

`nvm` is a popular way to run Node. It allows you to easily switch the Node version, and install new versions to try and easily rollback if something breaks, for example.

It is also very useful to test your code with old Node versions.

See <https://github.com/creationix/nvm> for more information about this option.

My suggestion is to use the official installer if you are just starting out and you don't use Homebrew already, otherwise, Homebrew is my favorite solution.

In any case, when Node is installed you'll have access to the `node` executable program in the command line.

How much JavaScript do you need to know to use Node?

If you are just starting out with JavaScript, how much deeply do you need to know the language?

As a beginner, it's hard to get to a point where you are confident enough in your programming abilities.

While learning to code, you might also be confused at where does JavaScript end, and where Node.js begins, and vice versa.

I would recommend you to have a good grasp of the main JavaScript concepts before diving into Node.js:

- Lexical Structure
- Expressions
- Types
- Variables
- Functions
- this
- Arrow Functions
- Loops
- Loops and Scope
- Arrays
- Template Literals
- Semicolons
- Strict Mode
- ECMAScript 6, 2016, 2017

With those concepts in mind, you are well on your road to become a proficient JavaScript developer, in both the browser and in Node.js.

The following concepts are also key to understand asynchronous programming, which is one fundamental part of Node.js:

- Asynchronous programming and callbacks
- Timers
- Promises
- Async and Await
- Closures
- The Event Loop

Luckily I wrote a free ebook that explains all those topics, and it's called **JavaScript Fundamentals**. It's the most compact resource you'll find to learn all of this.

You can find the ebook at the bottom of this page: <https://flaviocopes.com/javascript/>.

Differences between Node and the Browser

How writing JavaScript application in Node.js differs from programming for the Web inside the browser

Both the browser and Node use JavaScript as their programming language.

Building apps that run in the browser is a completely different thing than building a Node.js application.

Despite the fact that it's always JavaScript, there are some key differences that make the experience radically different.

As a frontend developer that writes Node apps have a huge advantage - the language is still the same.

You have a huge opportunity because we know how hard it is to fully, deeply learn a programming language, and by using the same language to perform all your work on the web - both on the client and on the server, you're in a unique position of advantage.

What changes is the ecosystem.

In the browser, most of the time what you are doing is interacting with the [DOM](#), or other [Web Platform APIs](#) like Cookies. Those do not exist in Node, of course. You don't have the `document` , `window` and all the other objects that are provided by the browser.

And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.

Another big difference is that in Node.js you control the environment. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node you will run the application on. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.

This means that you can write all the modern [ES6-7-8-9](#) JavaScript that your Node version supports.

Since JavaScript moves so fast, but browsers can be a bit slow and users a bit slow to upgrade, sometimes on the web, you are stuck to use older JavaScript / ECMAScript releases.

YYou can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node, you won't need that.

Another difference is that Node uses the [CommonJS module system](#), while in the browser we are starting to see the [ES Modules](#) standard being implemented.

In practice, this means that for the time being you use `require()` in Node and `import` in the browser.

v8

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome. V8 provides the runtime environment in which JavaScript executes. The DOM and the other Web Platform APIs are provided by the browser.



V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome.

V8 provides the runtime environment in which JavaScript executes. The [DOM](#), and the other [Web Platform APIs](#) are provided by the browser.

The cool thing is that the JavaScript engine is independent by the browser in which it's hosted. This key feature enabled the rise of [Node.js](#). V8 was chosen for being the engine chosen by Node.js back in 2009, and as the popularity of Node.js exploded, V8 became the engine that now powers an incredible amount of server-side code written in JavaScript.

The Node.js ecosystem is huge and thanks to it V8 also powers desktop apps, with projects like Electron.

Other JS engines

Other browsers have their own JavaScript engine:

- Firefox has [Spidermonkey](#)
- Safari has **JavaScriptCore** (also called Nitro)
- Edge has **Chakra**

and many others exist as well.

All those engines implement the ECMA ES-262 standard, also called [ECMAScript](#), the standard used by JavaScript.

The quest for performance

V8 is written in C++, and it's continuously improved. It is portable and runs on Mac, Windows, Linux and several other systems.

In this V8 introduction, I will ignore the implementation details of V8: they can be found on more authoritative sites (e.g. the V8 official site), and they change over time, often radically.

V8 is always evolving, just like the other JavaScript engines around, to speed up the Web and the Node.js ecosystem.

On the web, there is a race for performance that's been going on for years, and we (as users and developers) benefit a lot from this competition because we get faster and more optimized machines year after year.

Compilation

JavaScript is generally considered an interpreted language, but modern JavaScript engines no longer just interpret JavaScript, they compile it.

This happens since 2009 when the SpiderMonkey JavaScript compiler was added to Firefox 3.5, and everyone followed this idea.

JavaScript is internally compiled by V8 with **just-in-time (JIT) compilation** to speed up the execution.

This might seem counter-intuitive, but since the introduction of Google Maps in 2004, JavaScript has evolved from a language that was generally executing a few dozens of lines of code to complete applications with thousands to hundreds of thousands of lines running in the browser.