Quickstart

# Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app is run in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

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

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

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

# A Vast Number of Libraries

npm with its simple structure helped the ecosystem of Node.js proliferate, and now the npm registry hosts over 1,000,000 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 hostname = '127.0.0.1'  
const port = process.env.PORT

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}/`)  
})

This code first includes the Node.js http module.  
Node.js has a fantastic standard library, including 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 host name. 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 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. In order to make things easy and exciting for developers, thousands of libraries were built upon Node.js by the community.

Many of those established over time as popular options. Here is a non-comprehensive list of the ones worth learning:

* [**AdonisJs**](https://adonisjs.com/): A full-stack framework highly focused on developer ergonomics, stability, and confidence. Adonis is one of the fastest Node.js web frameworks.
* [**Express**](https://expressjs.com/): It provides 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.
* [**Fastify**](https://fastify.io/): A web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture. Fastify is one of the fastest Node.js web frameworks.
* [**Gatsby**](https://www.gatsbyjs.com/): A [React](https://reactjs.org/)-based, [GraphQL](https://graphql.org/) powered, static site generator with a very rich ecosystem of plugins and starters.
* [**hapi**](https://hapijs.com/): A rich framework for building applications and services that enables developers to focus on writing reusable application logic instead of spending time building infrastructure.
* [**koa**](http://koajs.com/): It is 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.
* [**Loopback.io**](https://loopback.io/): Makes it easy to build modern applications that require complex integrations.
* [**Meteor**](https://meteor.com/): 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](https://reactjs.org/), [Vue](https://vuejs.org/), and [Angular](https://angular.io/). Can be used to create mobile apps as well.
* [**Micro**](https://github.com/zeit/micro): It provides a very lightweight server to create asynchronous HTTP microservices.
* [**NestJS**](https://nestjs.com/): A TypeScript based progressive Node.js framework for building enterprise-grade efficient, reliable and scalable server-side applications.
* [**Next.js**](https://nextjs.org/): A framework to render server-side rendered [React](https://reactjs.org/) applications.
* [**Nx**](https://nx.dev/): A toolkit for full-stack monorepo development using NestJS, Express, [React](https://reactjs.org/), [Angular](https://angular.io/), and more! Nx helps scale your development from one team building one application to many teams collaborating on multiple applications!
* [**Sapper**](https://sapper.svelte.dev/): Sapper is a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing. Offers SSR and more!
* [**Socket.io**](https://socket.io/): A real-time communication engine to build network applications.
* [**Strapi**](https://strapi.io/): Strapi is a flexible, open-source Headless CMS that gives developers the freedom to choose their favorite tools and frameworks while also allowing editors to easily manage and distribute their content. By making the admin panel and API extensible through a plugin system, Strapi enables the world's largest companies to accelerate content delivery while building beautiful digital experiences.

A brief history of Node.js

Believe it or not, Node.js is only 11 years old.

In comparison, JavaScript is 24 years old and the Web is 31 years old.

Eleven years isn't a very long time in tech, but Node.js seems to have been around forever.

I've had the pleasure to work with Node.js since the early days when it was only 2 years old, and despite the limited information in the wild, you could already feel that it was going to be a huge thing.

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

# 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 that could create dynamic pages using server-side JavaScript. Unfortunately, Netscape LiveWire wasn't very successful and server-side JavaScript wasn't popularized until recently, by the introduction of Node.js.

One key factor that led to the rise of Node.js was the timing. Just a few years earlier, JavaScript had started to be considered as a more serious language, thanks to "Web 2.0" applications (such as Flickr, Gmail, etc.) that showed the world what a modern experience on the web could be like.

JavaScript engines also became considerably better as many browsers competed to offer users the best performance. Development teams behind major browsers worked hard to offer better support for JavaScript and find ways to make JavaScript run faster. The engine that Node.js uses under the hood, V8 (also known as Chrome V8 for being the open-source JavaScript engine of The Chromium Project), improved significantly due to this competition.

Node.js happened to be built in the right place and right time, but luck isn't the only reason why it is popular today. It introduces a lot of innovative thinking and approaches for JavaScript server-side development that has already helped many developers.

## 2009

* Node.js is born
* The first form of [npm](https://www.npmjs.com/) is created

## 2010

* [Express](https://expressjs.com/) is born
* [Socket.io](https://socket.io/) is born

## 2011

* npm hits version 1.0
* Larger companies start adopting Node.js: LinkedIn, Uber, etc.
* [hapi](https://hapijs.com/) is born

## 2012

* Adoption continues very rapidly

## 2013

* First big blogging platform using Node.js: [Ghost](https://ghost.org/)
* [Koa](https://koajs.com/) is born

## 2014

* The Big Fork: [io.js](https://iojs.org/) is a major fork of Node.js, with the goal of introducing ES6 support and moving faster

## 2015

* The [Node.js Foundation](https://foundation.nodejs.org/) is born
* IO.js is merged back into Node.js
* npm introduces private modules
* Node.js 4 (versions 1, 2 and 3 never previously released)

## 2016

* The [leftpad incident](https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm)
* [Yarn](https://yarnpkg.com/en/) is born
* Node.js 6

## 2017

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

## 2018

* Node.js 10
* [ES modules](https://nodejs.org/api/esm.html) .mjs experimental support
* Node.js 11

## 2019

* Node.js 12
* Node.js 13

## 2020

* Node.js 14
* Node.js 15

# How to install Node.js

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](https://brew.sh/) 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.js. It allows you to easily switch the Node.js 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.js 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.js 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.js?

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
* Scopes
* 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

# Differences between Node.js and the Browser

Both the browser and Node.js 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.

From the perspective of a frontend developer who extensively uses JavaScript, Node.js apps bring with them a huge advantage: the comfort of programming everything - the frontend and the backend - in a single language.

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.js, 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.js 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.js 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 with using older JavaScript / ECMAScript releases.

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

Another difference is that Node.js 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.js and import in the browser.