[**Introduction to Node.js**](https://nodejs.dev/learn/introduction-to-nodejs)

# Node.js is an open-source and cross-platform JavaScript runtime environment.

# Node.js runs the V8 JavaScript engine, the core of Google Chrome,

# A Node.js app runs in a single process, without creating a new thread for every request.

# 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,

# I/O operation, like reading from the network, accessing a database or the file system,

# In case of I/O operation 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.

# 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 = process.env.PORT;

const server = http.createServer((req, res) => {

  res.statusCode = 200;

  res.setHeader("Content-Type", "text/plain"); // set the Content-Type header

  res.end("Hello World!\n");// and we close the response, adding the content as an argument to end():

});

server.listen(port, hostname, () => {

  console.log(`Server running at http://${hostname}:${port}/`);

});

# Whenever a new request is received, the [request event](https://nodejs.org/api/http.html#http_event_request) is called, providing two objects: a request (an [http.IncomingMessage](https://nodejs.org/api/http.html#http_class_http_incomingmessage) object) and a response (an [http.ServerResponse](https://nodejs.org/api/http.html#http_class_http_serverresponse) object).

# the first provides the request details. You could access the request headers and request data. The second is used to return data to the caller.

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

[**Node.js Frameworks and Tools**](https://nodejs.dev/learn/introduction-to-nodejs#nodejs-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.

[**A brief history of Node.js**](https://nodejs.dev/learn/a-brief-history-of-nodejs)

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

In comparison, JavaScript is [24 years old](https://en.wikipedia.org/wiki/JavaScript#Beginnings_at_Netscape) and the Web is [31 years old](https://howoldistheinter.net/).

[**A little bit of history**](https://nodejs.dev/learn/a-brief-history-of-nodejs#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](https://en.wikipedia.org/wiki/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.

# **2009**

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

……..

[**How much JavaScript do you need to know to use Node.js?**](https://nodejs.dev/learn/how-much-javascript-do-you-need-to-know-to-use-nodejs)

# 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

# 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

[**Node.js and the Browser**](https://nodejs.dev/learn/differences-between-nodejs-and-the-browser)

# 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.

# 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.

# 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 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.

# this means that for the time being you use require() in Node.js and import in the browser.

[**The V8 JavaScript Engine**](https://nodejs.dev/learn/the-v8-javascript-engine)

# V8 is the name of the JavaScript engine that powers Google Chrome.

# V8 provides the runtime environment in which JavaScript executes.

# The DOM, and the other Web Platform APIs are provided by the browser.

[**Other JS engines**](https://nodejs.dev/learn/the-v8-javascript-engine#other-js-engines)

# Other browsers have their own JavaScript engine:

* Firefox has [**SpiderMonkey**](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey)
* Safari has [**JavaScriptCore**](https://developer.apple.com/documentation/javascriptcore) (also called Nitro)
* Edge was originally based on [**Chakra**](https://github.com/Microsoft/ChakraCore) but has more recently been [rebuilt using Chromium](https://support.microsoft.com/en-us/help/4501095/download-the-new-microsoft-edge-based-on-chromium) and the V8 engine.

# All those engines implement the [ECMA ES-262 standard](https://www.ecma-international.org/publications/standards/Ecma-262.htm), also called ECMAScript, the standard used by JavaScript.

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

# the [V8 official site](https://v8.dev/)

[**Compilation**](https://nodejs.dev/learn/the-v8-javascript-engine#compilation)

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

# This has been happening 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.

# In this *new world*, compiling JavaScript makes perfect sense because while it might take a little bit more to have the JavaScript *ready*, once done it's going to be much more performant than purely interpreted code.

[**Run Node.js scripts from the command line**](https://nodejs.dev/learn/run-nodejs-scripts-from-the-command-line)

# usual way to run a Node.js program is to run the **node**

node app.js

# Above, you are explicitly telling the shell to run your script with **node**.

# You can also embed this information into your JavaScript file with a "**shebang**" line. The "**shebang**" is the first line in the file, and tells the OS which interpreter to use for running the script. **Below is the first line of JavaScript:**

**#!/usr/bin/node**

# Above, we are explicitly giving the absolute path of interpreter. Not all operating systems have node in the bin folder, but all should have env. You can tell the OS to run env with node as parameter:

**#!/usr/bin/env node**

**// your code**

# To use a shebang, your file should have executable permission. You can give app.js the executable permission by running:

chmod u+x app.js

[**Restart the application automatically**](https://nodejs.dev/learn/run-nodejs-scripts-from-the-command-line#restart-the-application-automatically)

# The **node** command has to be re-executed in bash whenever there is a change in the application, to restart the application automatically, **nodemon** module is used.

[**How to exit from a Node.js program**](https://nodejs.dev/learn/how-to-exit-from-a-nodejs-program)

various ways to terminate a Node.js application.

# in the console you can stop prog. with **ctrl-C**,

Let's start with the most drastic one, and see why *you're better off not using it.*

The **process** core module provides a handy method that allows you to programmatically exit from a Node.js program: **process.exit().**

When Node.js runs this line, the process is immediately forced to terminate. This means that any callback that's pending, any network request still being sent, any filesystem access, or processes writing to stdout or stderr - all is going to be **ungracefully terminated right away**.

process.exit(1)

By default the exit code is 0, which means success. You can read more on exit codes at <https://nodejs.org/api/process.html#process_exit_codes>

# You can also set the process.exitCode property:

process.exitCode = 1

and when the program ends, Node.js will return that exit code.

# Many times with Node.js we start servers, like this HTTP server:

const express = require('express')

const app = express()

app.get('/', (req, res) => {

  res.send('Hi!')

})

const server = app.listen(3000, () => console.log('Server ready'))

// Express is a framework that uses the http module under the hood, app.listen() returns an instance of http. You would use https.createServer if you needed to serve your app using HTTPS, as app.listen only uses the http module.

This program is never going to end. If you call process.exit(), any currently pending or running request is going to be aborted. This is *not nice*. In this case you need to send the command a **SIGTERM** signal, and handle that with the process signal handler:

process.on('SIGTERM', () => {

  server.close(() => {

    console.log('Process terminated')

  })

})

*What are* ***signals****? Signals are a* ***POSIX intercommunication*** *system: a notification sent to a process in order to notify it of an event that occurred.*

***SIGKILL*** *is the signal that tells a process to immediately terminate, and would ideally act like process.exit().*

***SIGTERM*** *is the signal that tells a process to gracefully terminate. It is the signal that's sent from process managers like* ***upstart*** *or* ***supervisord*** *and many others.*

process.kill(process.pid, 'SIGTERM')

[**How to read environment variables from Node.js**](https://nodejs.dev/learn/how-to-read-environment-variables-from-nodejs)

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#