The V8 JavaScript Engine

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 of the browser in which it's hosted. This key feature enabled the rise of Node.js. V8 was chosen to be the engine that powered 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](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.

and many others exist as well.

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.

**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, we will ignore the implementation details of V8: they can be found on more authoritative sites (e.g. the [V8 official site](https://v8.dev/)), 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 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.

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.

Our applications now can run for hours inside a browser, rather than being just a few form validation rules or simple scripts.

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

The usual way to run a Node.js program is to run the node globally available command (once you install Node.js) and pass the name of the file you want to execute.

If your main Node.js application file is app.js, you can call it by typing:

node app.js

While running the command, make sure you are in the same directory which contains the app.js file.

How to exit from a Node.js program

There are various ways to terminate a Node.js application.

When running a program in the console you can close it with ctrl-C, but what we want to discuss here is programmatically exiting.

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.

If this is fine for you, you can pass an integer that signals the operating system the exit code:

process.exit(1)

By default the exit code is 0, which means success. Different exit codes have different meaning, which you might want to use in your own system to have the program communicate to other programs.

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 will later end, Node.js will return that exit code.

A program will gracefully exit when all the processing is done.

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!')

})

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

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:

*Note: process does not require a "require", it's automatically available.*

const express = require('express')

const app = express()

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

res.send('Hi!')

})

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

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.

You can send this signal from inside the program, in another function:

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

Or from another Node.js running program, or any other app running in your system that knows the PID of the process you want to terminate.