A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O. Making blocking behavior the exception rather than the norm.

Another big difference is that in Node.js you control the environment.

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.

**V8 JavaScript Engine**

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.

V8 is written in C++.

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

node app.js

telling the shell to run your script with node.

*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.*

*Dotenv*

*You can also run your js file with node -r dotenv/config index.js command if you don't want to import the package in your code.*

If after some code you type \_, that is going to print the result of the last operation.

A global installation is performed using the -g flag:

npm install -g lodash

The package.json file is kind of a manifest for your project. It can do a lot of things, completely unrelated. It's a central repository of configuration for tools.

The goal of package-lock.json file is to keep track of the exact version of every package that is installed so that a product is 100% reproducible in the same way even if packages are updated by their maintainers.

# Event Loop

The Node.js JavaScript code runs on a single thread. There is just one thing happening at a time.

# process.nextTick()

Every time the event loop takes a full trip, we call it a tick.

When we pass a function to process.nextTick(), we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts:

Calling setTimeout(() => {}, 0) will execute the function at the end of next tick, much later than when using nextTick() which prioritizes the call and executes it just before the beginning of the next tick.

Use nextTick() when you want to make sure that in the next event loop iteration that code is already executed.

# setImmediate()

When you want to execute some piece of code asynchronously, but as soon as possible, one option is to use the setImmediate() function provided by Node.js.

Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.

A function passed to process.nextTick() is going to be executed on the current iteration of the event loop, after the current operation ends. This means it will always execute before setTimeout and setImmediate.

A setTimeout() callback with a 0ms delay is very similar to setImmediate(). The execution order will depend on various factors, but they will be both run in the next iteration of the event loop.

## Recursive setTimeout

setInterval starts a function every n milliseconds, without any consideration about when a function finished its execution.

Asynchronous

Asynchronous means that things can happen independently of the main program flow.

JavaScript is **synchronous** by default and is single threaded. This means that code cannot create new threads and run in parallel.

Promise

A promise is commonly defined as **a proxy for a value that will eventually become available**.

Promises are one way to deal with asynchronous code, without getting stuck in [callback hell](http://callbackhell.com/).

Once a promise has been called, it will start in a **pending state**. This means that the calling function continues executing, while the promise is pending until it resolves, giving the calling function whatever data was being requested.

The created promise will eventually end in a **resolved state**, or in a **rejected state**, calling the respective callback functions (passed to then and catch) upon finishing.

The Promise API exposes a Promise constructor, which you initialize using new Promise().

## Chaining promises

A promise can be returned to another promise, creating a chain of promises.

A great example of chaining promises is the Fetch API, which we can use to get a resource and queue a chain of promises to execute when the resource is fetched.

The Fetch API is a promise-based mechanism, and calling fetch() is equivalent to defining our own promise using new Promise().

# os module

This module provides many functions that you can use to retrieve information from the underlying operating system and the computer the program runs on, and interact with it.

os.constants.signals tells us all the constants related to handling process signals, like SIGHUP, SIGKILL and so on.

os.constants.errno sets the constants for error reporting, like EADDRINUSE, EOVERFLOW and more.

## os.arch(): Return the string that identifies the underlying architecture, like arm, x64, arm64.

## os.cpus(): Return information on the CPUs available on your system.

## os.freemem(): Return the number of bytes that represent the free memory in the system.

## os.homedir(): Return the path to the home directory of the current user.

## os.hostname(): Return the host name.

## os.loadavg(): Return the calculation made by the operating system on the load average.

## os.networkInterfaces(): Returns the details of the network interfaces available on your system.

## os.release(): Returns a string that identifies the operating system release number.

## os.tmpdir(): Returns the path to the assigned temp folder.

## os.totalmem(): Returns the number of bytes that represent the total memory available in the system.

## os.uptime(): Returns the number of seconds the computer has been running since it was last rebooted.

## os.userInfo(): Returns an object that contains the current username, uid, gid, shell, and homedir.

## **buffer**

A buffer is created using the [Buffer.from()](https://nodejs.org/api/buffer.html" \l "buffer_buffer_from_buffer_alloc_and_buffer_allocunsafe), [Buffer.alloc()](https://nodejs.org/api/buffer.html#buffer_class_method_buffer_alloc_size_fill_encoding), and [Buffer.allocUnsafe()](https://nodejs.org/api/buffer.html" \l "buffer_class_method_buffer_allocunsafe_size) methods.

# Streams

Using streams you read it piece by piece, processing its content without keeping it all in memory.

Streams basically provide two major advantages over using other data handling methods: **Memory efficiency, Time efficiency.**