# Getting Started

## introduction to nodejs

# 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 runs 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.

## An Example Node.js Application

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

```
const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

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

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

```
import { createServer } from 'node:http';

const hostname = '127.0.0.1';
const port = 3000;

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

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. If you use mjs version of the code, you should save it as a `server.mjs` file and run `node server.mjs` in your terminal.

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');
```

---

## how to install nodejs

## 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/download/.

One very convenient way to install Node.js is through a package manager. In this case, every operating system has its own. Other package managers for macOS, Linux, and Windows are listed in https://nodejs.org/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. It is also very useful to test your code with old Node.js versions.

> See https://github.com/nvm-sh/nvm for more information about this option.

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.

### What is recommended to learn before diving deep with Node.js?

- Lexical Structure
- Expressions
- Data Types
- Classes
- Variables
- Functions
- `this` operator
- Arrow Functions
- Loops
- Scopes
- Arrays
- Template Literals
- Strict Mode
- ECMAScript 2015 (ES6) and beyond

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.

#### Asynchronous Programming

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

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

## differences between nodejs and the browser

# 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 completely different from 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 ES2015+ 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 supports both the CommonJS and ES module systems (since Node.js v12), while in the browser, we are starting to see the ES Modules standard being implemented.

In practice, this means that you can use both `require()` and `import` in Node.js, while you are limited to `import` in the browser.

---

# 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 is the JavaScript engine i.e. it parses and executes JavaScript code. The DOM, and the other Web Platform APIs (they all makeup runtime environment) 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 V8 which 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 was originally based on **Chakra** but has more recently been rebuilt using Chromium and the V8 engine.

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, we 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 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 can now 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.

# An introduction to the npm package manager

## Introduction to npm

`npm` is the standard package manager for Node.js.

In September 2022 over 2.1 million packages were reported being listed in the npm registry, making it the biggest single language code repository on Earth, and you can be sure there is a package for (almost!) everything.

It started as a way to download and manage dependencies of Node.js packages, but it has since become a tool used also in frontend JavaScript.

> **Yarn** and **pnpm** are alternatives to npm cli. You can check them out as well.

## Packages

`npm` installs, updates and manages downloads of dependencies of your project. Dependencies are pre-built pieces of code, such as libraries and packages, that your Node.js application needs to work.

### Installing all dependencies

If a project has a `package.json` file, by running

```
npm install
```

it will install everything the project needs, in the `node_modules` folder, creating it if it's not existing already.

### Installing a single package

You can also install a specific package by running

```
npm install <package-name>
```

Furthermore, since npm 5, this command adds `<package-name>` to the `package.json` file *dependencies*. Before version 5, you needed to add the flag `--save`.

Often you'll see more flags added to this command:

- `--save-dev` installs and adds the entry to the `package.json` file *devDependencies*
- `--no-save` installs but does not add the entry to the `package.json` file *dependencies*
- `--save-optional` installs and adds the entry to the `package.json` file *optionalDependencies*
- `--no-optional` will prevent optional dependencies from being installed

Shorthands of the flags can also be used:

- -S: `--save`
- -D: `--save-dev`
- -O: `--save-optional`

The difference between *devDependencies* and *dependencies* is that the former contains development tools, like a testing library, while the latter is bundled with the app in production.

As for the *optionalDependencies* the difference is that build failure of the dependency will not cause installation to fail. But it is your program's responsibility to handle the lack of the dependency. Read more about optional dependencies.

### Updating packages

Updating is also made easy, by running

```
npm update
```

`npm` will check all packages for a newer version that satisfies your versioning constraints.

You can specify a single package to update as well:

```
npm update <package-name>
```

## Versioning

In addition to plain downloads, `npm` also manages **versioning**, so you can specify any specific version of a package, or require a version higher or lower than what you need.

Many times you'll find that a library is only compatible with a major release of another library.

Or a bug in the latest release of a lib, still unfixed, is causing an issue.

Specifying an explicit version of a library also helps to keep everyone on the same exact version of a package, so that the whole team runs the same version until the `package.json` file is updated.

In all those cases, versioning helps a lot, and `npm` follows the semantic versioning (semver) standard.

You can install a specific version of a package, by running

```
npm install <package-name>@<version>
```

## Running Tasks

The package.json file supports a format for specifying command line tasks that can be run by using

```
npm run <task-name>
```

For example:

```
{
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  }
}
```

It's very common to use this feature to run Webpack:

```
{
  "scripts": {
    "watch": "webpack --watch --progress --colors --config webpack.conf.js",
    "dev": "webpack --progress --colors --config webpack.conf.js",
    "prod": "NODE_ENV=production webpack -p --config webpack.conf.js"
  }
}
```

So instead of typing those long commands, which are easy to forget or mistype, you can run

```
$ npm run watch
$ npm run dev
$ npm run prod
```

# ECMAScript 2015 (ES6) and beyond

Node.js is built against modern versions of V8. By keeping up-to-date with the latest releases of this engine, we ensure new features from the JavaScript ECMA-262 specification are brought to Node.js developers in a timely manner, as well as continued performance and stability improvements.

All ECMAScript 2015 (ES6) features are split into three groups for **shipping**, **staged**, and **in progress** features:

- All **shipping** features, which V8 considers stable, are turned **on by default on Node.js** and do **NOT** require any kind of runtime flag.
- **Staged** features, which are almost-completed features that are not considered stable by the V8 team, require a runtime flag: `--harmony`.
- **In progress** features can be activated individually by their respective harmony flag, although this is highly discouraged unless for testing purposes. Note: these flags are exposed by V8 and will potentially change without any deprecation notice.

### Which features ship with which Node.js version by default?

The website node.green provides an excellent overview over supported ECMAScript features in various versions of Node.js, based on kangax's compat-table.

### Which features are in progress?

New features are constantly being added to the V8 engine. Generally speaking, expect them to land on a future Node.js release, although timing is unknown.

You may list all the *in progress* features available on each Node.js release by grepping through the `--v8-options` argument. Please note that these are incomplete and possibly broken features of V8, so use them at your own risk:

```
node --v8-options | grep "in progress"
```

### I have my infrastructure set up to leverage the --harmony flag. Should I remove it?

The current behavior of the `--harmony` flag on Node.js is to enable **staged** features only. After all, it is now a synonym of `--es_staging`. As mentioned above, these are completed features that have not been considered stable yet. If you want to play safe, especially on production environments, consider removing this runtime flag until it ships by default on V8 and, consequently, on Node.js. If you keep this enabled, you should be prepared for further Node.js upgrades to break your code if V8 changes their semantics to more closely follow the standard.

### How do I find which version of V8 ships with a particular version of Node.js?

Node.js provides a simple way to list all dependencies and respective versions that ship with a specific binary through the `process` global object. In case of the V8 engine, type the following in your terminal to retrieve its version:

```
node -p process.versions.v8
```

## nodejs the difference between development and production

# Node.js, the difference between development and production

**There is no difference between development and production in Node.js**, i.e., there are no specific settings you need to apply to make Node.js work in a production configuration. However, a few libraries in the npm registry recognize using the `NODE_ENV` variable and default it to a `development` setting. Always run your Node.js with the `NODE_ENV=production` set.

A popular way of configuring your application is by using the twelve factor methodology.

## NODE_ENV in Express

In the wildly popular express framework, setting the `NODE_ENV` to `production` generally ensures that:

- logging is kept to a minimum, essential level
- more caching levels take place to optimize performance

This is usually done by executing the command

```
export NODE_ENV=production
```

in the shell, but it's better to put it in your shell configuration file (e.g. `.bash_profile` with the Bash shell) because otherwise the setting does not persist in case of a system restart.

You can also apply the environment variable by prepending it to your application initialization command:

```
NODE_ENV=production node app.js
```

For example, in an Express app, you can use this to set different error handlers per environment:

```
if (process.env.NODE_ENV === 'development') {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
}

if (process.env.NODE_ENV === 'production') {
  app.use(express.errorHandler());
}
```

For example Pug, the templating library used by Express, compiles in debug mode if `NODE_ENV` is not set to `production`. Express views are compiled in every request in development mode, while in production they are cached. There are many more examples.

**This environment variable is a convention widely used in external libraries, but not within Node.js itself**.

## Why is NODE_ENV considered an antipattern?

An environment is a digital platform or a system where engineers can build, test, *deploy*, and manage software products. Conventionally, there are four stages or types of environments where our application is run:

- Development
- Testing
- Staging
- Production

The fundamental problem of `NODE_ENV` stems from developers combining optimizations and software behavior with the environment their software is running on. The result is code like the following:

```
if (process.env.NODE_ENV === 'development') {
  // ...
}

if (process.env.NODE_ENV === 'production') {
  // ...
}
```

```
if (['production', 'staging'].includes(process.env.NODE_ENV)) {
  // ...
}
```

While this might look harmless, it makes the production and staging environments different, thus making reliable testing impossible. For example a test and thus a functionality of your product could pass when `NODE_ENV` is set to `development` but fail when setting `NODE_ENV` to `production`. Therefore, setting `NODE_ENV` to anything but `production` is considered an *antipattern*.

---

# Node.js with WebAssembly

**WebAssembly** is a high-performance assembly-like language that can be compiled from various languages, including C/C++, Rust, and AssemblyScript. Currently, it is supported by Chrome, Firefox, Safari, Edge, and Node.js!

The WebAssembly specification details two file formats, a binary format called a WebAssembly Module with a `.wasm` extension and corresponding text representation called WebAssembly Text format with a `.wat` extension.

## Key Concepts

- Module - A compiled WebAssembly binary, ie a `.wasm` file.
- Memory - A resizable ArrayBuffer.
- Table - A resizable typed array of references not stored in Memory.
- Instance - An instantiation of a Module with its Memory, Table, and variables.

In order to use WebAssembly, you need a `.wasm` binary file and a set of APIs to communicate with WebAssembly. Node.js provides the necessary APIs via the global `WebAssembly` object.

```
console.log(WebAssembly);
/*
Object [WebAssembly] {
  compile: [Function: compile],
  validate: [Function: validate],
  instantiate: [Function: instantiate]
}
*/
```

## Generating WebAssembly Modules

There are multiple methods available to generate WebAssembly binary files including:

- Writing WebAssembly (`.wat`) by hand and converting to binary format using tools such as wabt
- Using emscripten with a C/C++ application
- Using wasm-pack with a Rust application
- Using AssemblyScript if you prefer a TypeScript-like experience

> Some of these tools generate not only the binary file, but the JavaScript "glue" code and corresponding HTML files to run in the browser.

## How to use it

Once you have a WebAssembly module, you can use the Node.js `WebAssembly` object to instantiate it.

```
// Assume add.wasm file exists that contains a single function adding 2 provided arguments
const fs = require('node:fs');

// Use the readFileSync function to read the contents of the "add.wasm" file
const wasmBuffer = fs.readFileSync('/path/to/add.wasm');

// Use the WebAssembly.instantiate method to instantiate the WebAssembly module
WebAssembly.instantiate(wasmBuffer).then(wasmModule => {
  // Exported function lives under instance.exports object
  const { add } = wasmModule.instance.exports;
  const sum = add(5, 6);
  console.log(sum); // Outputs: 11
});
```

```
// Assume add.wasm file exists that contains a single function adding 2 provided arguments
import fs from 'node:fs/promises';

// Use readFile to read contents of the "add.wasm" file
const wasmBuffer = await fs.readFile('path/to/add.wsm');
```

```
// Use the WebAssembly.instantiate method to instantiate the WebAssembly module
const wasmModule = await WebAssembly.instantiate(wasmBuffer);

// Exported function lives under instance.exports object
const { add } = wasmModule.instance.exports;

const sum = add(5, 6);

console.log(sum); // Outputs 11
```

## Interacting with the OS

WebAssembly modules cannot directly access OS functionality on its own. A third-party tool Wasmtime can be used to access this functionality. `Wasmtime` utilizes the WASI API to access the OS functionality.

## Resources

- General WebAssembly Information
- MDN Docs
- Write WebAssembly by hand

---

# Debugging Node.js

This guide will help you get started debugging your Node.js apps and scripts.

## Enable Inspector

When started with the `--inspect` switch, a Node.js process listens for a debugging client. By default, it will listen at host and port 127.0.0.1:9229. Each process is also assigned a unique UUID.

Inspector clients must know and specify host address, port, and UUID to connect. A full URL will look something like `ws://127.0.0.1:9229/0f2c936f-b1cd-4ac9-aab3-f63b0f33d55e`.

Node.js will also start listening for debugging messages if it receives a `SIGUSR1` signal. (`SIGUSR1` is not available on Windows.) In Node.js 7 and earlier, this activates the legacy Debugger API. In Node.js 8 and later, it will activate the Inspector API.

## Security Implications

Since the debugger has full access to the Node.js execution environment, a malicious actor able to connect to this port may be able to execute arbitrary code on behalf of the Node.js process. It is important to understand the security implications of exposing the debugger port on public and private networks.

### Exposing the debug port publicly is unsafe

If the debugger is bound to a public IP address, or to 0.0.0.0, any clients that can reach your IP address will be able to connect to the debugger without any restriction and will be able to run arbitrary code.

By default `node --inspect` binds to 127.0.0.1. You explicitly need to provide a public IP address or 0.0.0.0, etc., if you intend to allow external connections to the debugger. Doing so may expose you to a potentially significant security threat. We suggest you ensure appropriate firewalls and access controls in place to prevent a security exposure.

See the section on 'Enabling remote debugging scenarios' on some advice on how to safely allow remote debugger clients to connect.

### Local applications have full access to the inspector

Even if you bind the inspector port to 127.0.0.1 (the default), any applications running locally on your machine will have unrestricted access. This is by design to allow local debuggers to be able to attach conveniently.

### Browsers, WebSockets and same-origin policy

Websites open in a web-browser can make WebSocket and HTTP requests under the browser security model. An initial HTTP connection is necessary to obtain a unique debugger session id. The same-origin-policy prevents websites from being able to make this HTTP connection. For additional security against DNS rebinding attacks, Node.js verifies that the 'Host' headers for the connection either specify an IP address or `localhost` precisely.

These security policies disallow connecting to a remote debug server by specifying the hostname. You can work-around this restriction by specifying either the IP address or by using ssh tunnels as described below.

## Inspector Clients

A minimal CLI debugger is available with `node inspect myscript.js`. Several commercial and open source tools can also connect to the Node.js Inspector.

### Chrome DevTools 55+, Microsoft Edge

- **Option 1**: Open `chrome://inspect` in a Chromium-based browser or `edge://inspect` in Edge. Click the Configure button and ensure your target host and port are listed.
- **Option 2**: Copy the `devtoolsFrontendUrl` from the output of `/json/list` (see above) or the --inspect hint text and paste into Chrome.

See https://github.com/ChromeDevTools/devtools-frontend, https://www.microsoftedgeinsider.com for more information.

### Visual Studio Code 1.10+

- In the Debug panel, click the settings icon to open `.vscode/launch.json`. Select "Node.js" for initial setup.

See https://github.com/microsoft/vscode for more information.

### Visual Studio 2017+

- Choose "Debug > Start Debugging" from the menu or hit F5.
- Detailed instructions.

### JetBrains WebStorm and other JetBrains IDEs

- Create a new Node.js debug configuration and hit Debug. `--inspect` will be used by default for Node.js 7+. To disable uncheck `js.debugger.node.use.inspect` in the IDE Registry. To learn more about running and debugging Node.js in WebStorm and other JetBrains IDEs, check out WebStorm online help.

### chrome-remote-interface

- Library to ease connections to Inspector Protocol endpoints.

See https://github.com/cyrus-and/chrome-remote-interface for more information.

### Gitpod

- Start a Node.js debug configuration from the `Debug` view or hit `F5`. Detailed instructions

See https://www.gitpod.io for more information.

### Eclipse IDE with Eclipse Wild Web Developer extension

- From a .js file, choose "Debug As... > Node program", or
- Create a Debug Configuration to attach debugger to running Node.js application (already started with `--inspect`).

See https://eclipse.org/eclipseide for more information.

## Command-line options

The following table lists the impact of various runtime flags on debugging:

| Flag | Meaning | | -- | | --inspect | Enable inspector agent; Listen on default address and port (127.0.0.1:9229) | | --inspect=[host:port] | Enable inspector agent; Bind to address or hostname host (default: 127.0.0.1); Listen on port port (default: 9229) | | --inspect-brk | Enable inspector agent; Listen on default address and port (127.0.0.1:9229); Break before user code starts | | --inspect-brk=[host:port] | Enable inspector agent; Bind to address or hostname host (default: 127.0.0.1); Listen on port port (default: 9229); Break before user code starts | | --inspect-wait | Enable inspector agent; Listen on default address and port (127.0.0.1:9229); Wait for debugger to be attached. | | --inspect-wait=[host:port] | Enable inspector agent; Bind to address or hostname host (default: 127.0.0.1); Listen on port port (default: 9229); Wait for debugger to be attached. | | node inspect script.js | Spawn child process to run user's script under --inspect flag; and use main process to run CLI debugger. | | node inspect --port=xxxx script.js | Spawn child process to run user's script under --inspect flag; and use main process to run CLI debugger. Listen on port port (default: 9229) |

## Enabling remote debugging scenarios

We recommend that you never have the debugger listen on a public IP address. If you need to allow remote debugging connections we recommend the use of ssh tunnels instead. We provide the following example for illustrative purposes only. Please understand the security risk of allowing remote access to a privileged service before proceeding.

Let's say you are running Node.js on a remote machine, remote.example.com, that you want to be able to debug. On that machine, you should start the node process with the inspector listening only to localhost (the default).

```
node --inspect server.js
```

Now, on your local machine from where you want to initiate a debug client connection, you can setup an ssh tunnel:

```
ssh -L 9221:localhost:9229 user@remote.example.com
```

This starts a ssh tunnel session where a connection to port 9221 on your local machine will be forwarded to port 9229 on remote.example.com. You can now attach a debugger such as Chrome DevTools or Visual Studio Code to localhost:9221, which should be able to debug as if the Node.js application was running locally.

## Legacy Debugger

**The legacy debugger has been deprecated as of Node.js 7.7.0. Please use `--inspect` and Inspector instead.**

When started with the **--debug** or **--debug-brk** switches in version 7 and earlier, Node.js listens for debugging commands defined by the discontinued V8 Debugging Protocol on a TCP port, by default `5858`. Any debugger client which speaks this protocol can connect to and debug the running process; a couple popular ones are listed below.

The V8 Debugging Protocol is no longer maintained or documented.

### Built-in Debugger

Start `node debug script_name.js` to start your script under the builtin command-line debugger. Your script starts in another Node.js process started with the `--debug-brk` option, and the initial Node.js process runs the `_debugger.js` script and connects to your target. See docs for more information.

### node-inspector

Debug your Node.js app with Chrome DevTools by using an intermediary process which translates the Inspector Protocol used in Chromium to the V8 Debugger protocol used in Node.js. See https://github.com/node-inspector/node-inspector for more information.

# Profiling Node.js Applications

There are many third party tools available for profiling Node.js applications but, in many cases, the easiest option is to use the Node.js built-in profiler. The built-in profiler uses the profiler inside V8 which samples the stack at regular intervals during program execution. It records the results of these samples, along with important optimization events such as jit compiles, as a series of ticks:

```
code-creation,LazyCompile,0,0x2d5000a337a0,396,"bp native array.js:1153:16",0x289f644df68,~
code-creation,LazyCompile,0,0x2d5000a33940,716,"hasOwnProperty native v8natives.js:198:30",0x289f64438d0,~
code-creation,LazyCompile,0,0x2d5000a33c20,284,"ToName native runtime.js:549:16",0x289f643bb28,~
code-creation,Stub,2,0x2d5000a33d40,182,"DoubleToIStub"
code-creation,Stub,2,0x2d5000a33e00,507,"NumberToStringStub"
```

In the past, you needed the V8 source code to be able to interpret the ticks. Luckily, tools have been introduced since Node.js 4.4.0 that facilitate the consumption of this information without separately building V8 from source. Let's see how the built-in profiler can help provide insight into application performance.

To illustrate the use of the tick profiler, we will work with a simple Express application. Our application will have two handlers, one for adding new users to our system:

```
app.get('/newUser', (req, res) => {
  let username = req.query.username || '';
  const password = req.query.password || '';

  username = username.replace(/[!@#$%^&*]/g, '');

  if (!username || !password || users[username]) {
    return res.sendStatus(400);
  }

  const salt = crypto.randomBytes(128).toString('base64');
  const hash = crypto.pbkdf2Sync(password, salt, 10000, 512, 'sha512');

  users[username] = { salt, hash };

  res.sendStatus(200);
});
```

and another for validating user authentication attempts:

```
app.get('/auth', (req, res) => {
  let username = req.query.username || '';
  const password = req.query.password || '';

  username = username.replace(/[!@#$%^&*]/g, '');

  if (!username || !password || !users[username]) {
    return res.sendStatus(400);
  }

  const { salt, hash } = users[username];
  const encryptHash = crypto.pbkdf2Sync(password, salt, 10000, 512, 'sha512');

  if (crypto.timingSafeEqual(hash, encryptHash)) {
    res.sendStatus(200);
  } else {
    res.sendStatus(401);
  }
});
```

*Please note that these are NOT recommended handlers for authenticating users in your Node.js applications and are used purely for illustration purposes. You should not be trying to design your own cryptographic authentication mechanisms in general. It is much better to use existing, proven authentication solutions.*

Now assume that we've deployed our application and users are complaining about high latency on requests. We can easily run the app with the built-in profiler:

```
NODE_ENV=production node --prof app.js
```

and put some load on the server using `ab` (ApacheBench):

```
curl -X GET "http://localhost:8080/newUser?username=matt&password=password"
ab -k -c 20 -n 250 "http://localhost:8080/auth?username=matt&password=password"
```

and get an ab output of:

```
Concurrency Level:      20
Time taken for tests:   46.932 seconds
Complete requests:      250
Failed requests:        0
Keep-Alive requests:    250
Total transferred:      50250 bytes
HTML transferred:       500 bytes
Requests per second:    5.33 [#/sec] (mean)
Time per request:       3754.556 [ms] (mean)
Time per request:       187.728 [ms] (mean, across all concurrent requests)
Transfer rate:          1.05 [Kbytes/sec] received

...

Percentage of the requests served within a certain time (ms)
  50%   3755
  66%   3804
  75%   3818
  80%   3825
  90%   3845
  95%   3858
  98%   3874
  99%   3875
 100%   4225 (longest request)
```

From this output, we see that we're only managing to serve about 5 requests per second and that the average request takes just under 4 seconds round trip. In a real-world example, we could be doing lots of work in many functions on behalf of a user request but even in our simple example, time could be lost compiling regular expressions, generating random salts, generating unique hashes from user passwords, or inside the Express framework itself.

Since we ran our application using the `--prof` option, a tick file was generated in the same directory as your local run of the application. It should have the form `isolate-0xnnnnnnnnnnnn-v8.log` (where `n` is a digit).

In order to make sense of this file, we need to use the tick processor bundled with the Node.js binary. To run the processor, use the `--prof-process` flag:

```
node --prof-process isolate-0xnnnnnnnnnnnn-v8.log > processed.txt
```

Opening processed.txt in your favorite text editor will give you a few different types of information. The file is broken up into sections which are again broken up by language. First, we look at the summary section and see:

```
 [Summary]:
   ticks  total  nonlib   name
     79    0.2%    0.2%  JavaScript
  36703   97.2%   99.2%  C++
      7    0.0%    0.0%  GC
    767    2.0%          Shared libraries
    215    0.6%          Unaccounted
```

This tells us that 97% of all samples gathered occurred in C++ code and that when viewing other sections of the processed output we should pay most attention to work being done in C++ (as opposed to JavaScript). With this in mind, we next find the [C++] section which contains information about which C++ functions are taking the most CPU time and see:

```
 [C++]:
   ticks  total  nonlib   name
  19557   51.8%   52.9%  node::crypto::PBKDF2(v8::FunctionCallbackInfo<v8::Value> const&)
   4510   11.9%   12.2%  _sha1_block_data_order
   3165    8.4%    8.6%  _malloc_zone_malloc
```

We see that the top 3 entries account for 72.1% of CPU time taken by the program. From this output, we immediately see that at least 51.8% of CPU time is taken up by a function called PBKDF2 which corresponds to our hash generation from a user's password. However, it may not be immediately obvious how the lower two entries factor into our application (or if it is we will pretend otherwise for the sake of example). To better understand the relationship between these functions, we will next look at the [Bottom up (heavy) profile] section which provides information about the primary callers of each function. Examining this section, we find:

```
   ticks parent  name
  19557   51.8%  node::crypto::PBKDF2(v8::FunctionCallbackInfo<v8::Value> const&)
  19557  100.0%    v8::internal::Builtins::~Builtins()
  19557  100.0%      LazyCompile: ~pbkdf2 crypto.js:557:16

   4510   11.9%  _sha1_block_data_order
   4510  100.0%    LazyCompile: *pbkdf2 crypto.js:557:16
   4510  100.0%      LazyCompile: *exports.pbkdf2Sync crypto.js:552:30

   3165    8.4%  _malloc_zone_malloc
   3161   99.9%    LazyCompile: *pbkdf2 crypto.js:557:16
   3161  100.0%      LazyCompile: *exports.pbkdf2Sync crypto.js:552:30
```

Parsing this section takes a little more work than the raw tick counts above. Within each of the "call stacks" above, the percentage in the parent column tells you the percentage of samples for which the function in the row above was called by the function in the current row. For example, in the middle "call stack" above for _sha1_block_data_order, we see that `_sha1_block_data_order` occurred in 11.9% of samples, which we knew from the raw counts above. However, here, we can also tell that it was always called by the pbkdf2 function inside the Node.js crypto module. We see that similarly, `_malloc_zone_malloc` was called almost exclusively by the same pbkdf2 function. Thus, using the information in this view, we can tell that our hash computation from the user's password accounts not only for the 51.8% from above but also for all CPU time in the top 3 most sampled functions since the calls to `_sha1_block_data_order` and `_malloc_zone_malloc` were made on behalf of the pbkdf2 function.

At this point, it is very clear that the password-based hash generation should be the target of our optimization. Thankfully, you've fully internalized the benefits of asynchronous programming and you realize that the work to generate a hash from the user's password is being done in a synchronous way and thus tying down the event loop. This prevents us from working on other incoming requests while computing a hash.

To remedy this issue, you make a small modification to the above handlers to use the asynchronous version of the pbkdf2 function:

```
app.get('/auth', (req, res) => {
  let username = req.query.username || '';
  const password = req.query.password || '';

  username = username.replace(/[!@#$%^&*]/g, '');

  if (!username || !password || !users[username]) {
    return res.sendStatus(400);
  }

  crypto.pbkdf2(
    password,
    users[username].salt,
    10000,
    512,
    'sha512',
    (err, hash) => {
      if (users[username].hash.toString() === hash.toString()) {
        res.sendStatus(200);
      } else {
        res.sendStatus(401);
      }
    }
  );
});
```

A new run of the ab benchmark above with the asynchronous version of your app yields:

```
Concurrency Level:      20
Time taken for tests:   12.846 seconds
Complete requests:      250
Failed requests:        0
Keep-Alive requests:    250
Total transferred:      50250 bytes
HTML transferred:       500 bytes
```

```
Requests per second:    19.46 [#/sec] (mean)
Time per request:       1027.689 [ms] (mean)
Time per request:       51.384 [ms] (mean, across all concurrent requests)
Transfer rate:          3.82 [Kbytes/sec] received

...

Percentage of the requests served within a certain time (ms)
  50%   1018
  66%   1035
  75%   1041
  80%   1043
  90%   1049
  95%   1063
  98%   1070
  99%   1071
 100%   1079 (longest request)
```

Yay! Your app is now serving about 20 requests per second, roughly 4 times more than it was with the synchronous hash generation. Additionally, the average latency is down from the 4 seconds before to just over 1 second.

Hopefully, through the performance investigation of this (admittedly contrived) example, you've seen how the V8 tick processor can help you gain a better understanding of the performance of your Node.js applications.

You may also find how to create a flame graph helpful.

# Security Best Practices

## Intent

This document intends to extend the current threat model and provide extensive guidelines on how to secure a Node.js application.

## Document Content

- Best practices: A simplified condensed way to see the best practices. We can use this issue or this guideline as the starting point. It is important to note that this document is specific to Node.js, if you are looking for something broad, consider OSSF Best Practices.
- Attacks explained: illustrate and document in plain English with some code examples (if possible) of the attacks that we are mentioning in the threat model.
- Third-Party Libraries: define threats (typosquatting attacks, malicious packages...) and best practices regarding node modules dependencies, etc...

## Threat List

### Denial of Service of HTTP server (CWE-400)

This is an attack where the application becomes unavailable for the purpose it was designed due to the way it processes incoming HTTP requests. These requests need not be deliberately crafted by a malicious actor: a misconfigured or buggy client can also send a pattern of requests to the server that result in a denial of service.

HTTP requests are received by the Node.js HTTP server and handed over to the application code via the registered request handler. The server does not parse the content of the request body. Therefore any DoS caused by the contents of the body after they are handed over to the request handler is not a vulnerability in Node.js itself, since it's the responsibility of the application code to handle it correctly.

Ensure that the WebServer handles socket errors properly, for instance, when a server is created without an error handler, it will be vulnerable to DoS

```
const net = require('node:net');

const server = net.createServer(function (socket) {
  // socket.on('error', console.error) // this prevents the server to crash
  socket.write('Echo server\r\n');
  socket.pipe(socket);
});

server.listen(5000, '0.0.0.0');
```

```
import net from 'node:net';

const server = net.createServer(function (socket) {
  // socket.on('error', console.error) // this prevents the server to crash
  socket.write('Echo server\r\n');
  socket.pipe(socket);
});

server.listen(5000, '0.0.0.0');
```

If a *bad request* is performed the server could crash.

An example of a DoS attack that is not caused by the request's contents is Slowloris. In this attack, HTTP requests are sent slowly and fragmented, one fragment at a time. Until the full request is delivered, the server will keep resources dedicated to the ongoing request. If enough of these requests are sent at the same time, the amount of concurrent connections will soon reach its maximum resulting in a denial of service. This is how the attack depends not on the request's contents but on the timing and pattern of the requests being sent to the server.

**Mitigations**

- Use a reverse proxy to receive and forward requests to the Node.js application. Reverse proxies can provide caching, load balancing, IP blacklisting, etc. which reduce the probability of a DoS attack being effective.
- Correctly configure the server timeouts, so that connections that are idle or where requests are arriving too slowly can be dropped. See the different timeouts in `http.Server`, particularly `headersTimeout`, `requestTimeout`, `timeout`, and `keepAliveTimeout`.
- Limit the number of open sockets per host and in total. See the http docs, particularly `agent.maxSockets`, `agent.maxTotalSockets`, `agent.maxFreeSockets` and `server.maxRequestsPerSocket`.

### DNS Rebinding (CWE-346)

This is an attack that can target Node.js applications being run with the debugging inspector enabled using the --inspect switch.

Since websites opened in a web browser can make WebSocket and HTTP requests, they can target the debugging inspector running locally. This is usually prevented by the same-origin policy implemented by modern browsers, which forbids scripts from reaching resources from different origins (meaning a malicious website cannot read data requested from a local IP address).

However, through DNS rebinding, an attacker can temporarily control the origin for their requests so that they seem to originate from a local IP address. This is done by controlling both a website and the DNS server used to resolve its IP address. See DNS Rebinding wiki for more details.

#### Mitigations

- Disable inspector on *SIGUSR1* signal by attaching a `process.on('SIGUSR1', …)` listener to it.
- Do not run the inspector protocol in production.

### Exposure of Sensitive Information to an Unauthorized Actor (CWE-552)

All the files and folders included in the current directory are pushed to the npm registry during the package publication.

There are some mechanisms to control this behavior by defining a blocklist with `.npmignore` and `.gitignore` or by defining an allowlist in the `package.json`

#### Mitigations

- Using `npm publish --dry-run` to list all the files to publish. Ensure to review the content before publishing the package.
- It's also important to create and maintain ignore files such as `.gitignore` and `.npmignore`. Throughout these files, you can specify which files/folders should not be published. The files property in `package.json` allows the inverse operation -- allowed list.
- In case of an exposure, make sure to unpublish the package.

### HTTP Request Smuggling (CWE-444)

This is an attack that involves two HTTP servers (usually a proxy and a Node.js application). A client sends an HTTP request that goes first through the front-end server (the proxy) and then is redirected to the back-end server (the application). When the front-end and back-end interpret ambiguous HTTP requests differently, there is potential for an attacker to send a malicious message that won't be seen by the front-end but will be seen by the back-end, effectively "smuggling" it past the proxy server.

See the CWE-444 for a more detailed description and examples.

Since this attack depends on Node.js interpreting HTTP requests differently from an (arbitrary) HTTP server, a successful attack can be due to a vulnerability in Node.js, the front-end server, or both. If the way the request is interpreted by Node.js is consistent with the HTTP specification (see RFC7230), then it is not considered a vulnerability in Node.js.

#### Mitigations

- Do not use the `insecureHTTPParser` option when creating a HTTP Server.
- Configure the front-end server to normalize ambiguous requests.
- Continuously monitor for new HTTP request smuggling vulnerabilities in both Node.js and the front-end server of choice.
- Use HTTP/2 end to end and disable HTTP downgrading if possible.

### Information Exposure through Timing Attacks (CWE-208)

This is an attack that allows the attacker to learn potentially sensitive information by, for example, measuring how long it takes for the application to respond to a request. This attack is not specific to Node.js and can target almost all runtimes.

The attack is possible whenever the application uses a secret in a timing-sensitive operation (e.g., branch). Consider handling authentication in a typical application. Here, a basic authentication method includes email and password as credentials. User information is retrieved from the input the user has supplied from ideally a DBMS. Upon retrieving user information, the password is compared with the user information retrieved from the database. Using the built-in string comparison takes a longer time for the same-length values. This comparison, when run for an acceptable amount unwillingly increases the response time of the request. By comparing the request response times, an attacker can guess the length and the value of the password in a large quantity of requests.

**Mitigations**

- The crypto API exposes a function `timingSafeEqual` to compare actual and expected sensitive values using a constant-time algorithm.
- For password comparison, you can use the scrypt available also on the native crypto module.
- More generally, avoid using secrets in variable-time operations. This includes branching on secrets and, when the attacker could be co-located on the same infrastructure (e.g., same cloud machine), using a secret as an index into memory. Writing constant-time code in JavaScript is hard (partly because of the JIT). For crypto applications, use the built-in crypto APIs or WebAssembly (for algorithms not implemented in natively).

### Malicious Third-Party Modules (CWE-1357)

Currently, in Node.js, any package can access powerful resources such as network access. Furthermore, because they also have access to the file system, they can send any data anywhere.

All code running into a node process has the ability to load and run additional arbitrary code by using `eval()`(or its equivalents). All code with file system write access may achieve the same thing by writing to new or existing files that are loaded.

Node.js has an experimental[1] policy mechanism to declare the loaded resource as untrusted or trusted. However, this policy is not enabled by default. Be sure to pin dependency versions and run automatic checks for vulnerabilities using common workflows or npm scripts. Before installing a package make sure that this package is maintained and includes all the content you expected. Be careful, the GitHub source code is not always the same as the published one, validate it in the *node_modules*.

#### Supply chain attacks

A supply chain attack on a Node.js application happens when one of its dependencies (either direct or transitive) are compromised. This can happen either due to the application being too lax on the specification of the dependencies (allowing for unwanted updates) and/or common typos in the specification (vulnerable to typosquatting).

An attacker who takes control of an upstream package can publish a new version with malicious code in it. If a Node.js application depends on that package without being strict on which version is safe to use, the package can be automatically updated to the latest malicious version, compromising the application.

Dependencies specified in the `package.json` file can have an exact version number or a range. However, when pinning a dependency to an exact version, its transitive dependencies are not themselves pinned. This still leaves the application vulnerable to unwanted/unexpected updates.

Possible attack vectors:

- Typosquatting attacks
- Lockfile poisoning
- Compromised maintainers
- Malicious Packages
- Dependency Confusions

**Mitigations**

- Prevent npm from executing arbitrary scripts with `--ignore-scripts`
  - Additionally, you can disable it globally with `npm config set ignore-scripts true`
- Pin dependency versions to a specific immutable version, not a version that is a range or from a mutable source.
- Use lockfiles, which pin every dependency (direct and transitive).
  - Use Mitigations for lockfile poisoning.
- Automate checks for new vulnerabilities using CI, with tools like [`npm-audit`][].
  - Tools such as `Socket` can be used to analyze packages with static analysis to find risky behaviors such as network or filesystem access.
- Use `npm ci` instead of `npm install`. This enforces the lockfile so that inconsistencies between it and the *package.json* file causes an error (instead of silently ignoring the lockfile in favor of *package.json*).
- Carefully check the *package.json* file for errors/typos in the names of the dependencies.

### Memory Access Violation (CWE-284)

Memory-based or heap-based attacks depend on a combination of memory management errors and an exploitable memory allocator. Like all runtimes, Node.js is vulnerable to these attacks if your projects run on a shared machine. Using a secure heap is useful for preventing sensitive information from leaking due to pointer overruns and underruns.

Unfortunately, a secure heap is not available on Windows. More information can be found on Node.js secure-heap documentation.

**Mitigations**

- Use `--secure-heap=n` depending on your application where *n* is the allocated maximum byte size.
- Do not run your production app on a shared machine.

## Monkey Patching (CWE-349)

Monkey patching refers to the modification of properties in runtime aiming to change the existing behavior. Example:

```
// eslint-disable-next-line no-extend-native
Array.prototype.push = function (item) {
  // overriding the global [].push
};
```

**Mitigations**

The `--frozen-intrinsics` flag enables experimental[1] frozen intrinsics, which means all the built-in JavaScript objects and functions are recursively frozen. Therefore, the following snippet **will not** override the default behavior of `Array.prototype.push`

```
// eslint-disable-next-line no-extend-native
Array.prototype.push = function (item) {
  // overriding the global [].push
};

// Uncaught:
// TypeError <Object <Object <[Object: null prototype] {}>>>:
// Cannot assign to read only property 'push' of object ''
```

However, it's important to mention you can still define new globals and replace existing globals using `globalThis`

```
> globalThis.foo = 3; foo; // you can still define new globals
3
> globalThis.Array = 4; Array; // However, you can also replace existing globals
4
```

Therefore, `Object.freeze(globalThis)` can be used to guarantee no globals will be replaced.

## Prototype Pollution Attacks (CWE-1321)

Prototype pollution refers to the possibility of modifying or injecting properties into Javascript language items by abusing the usage of __proto_, _*constructor*, *prototype*, and other properties inherited from built-in prototypes.

```
const a = { a: 1, b: 2 };
const data = JSON.parse('{"__proto__": { "polluted": true}}');

const c = Object.assign({}, a, data);
console.log(c.polluted); // true

// Potential DoS
const data2 = JSON.parse('{"__proto__": null}');
const d = Object.assign(a, data2);
d.hasOwnProperty('b'); // Uncaught TypeError: d.hasOwnProperty is not a function
```

This is a potential vulnerability inherited from the JavaScript language.

**Examples**:

- CVE-2022-21824 (Node.js)
- CVE-2018-3721 (3rd Party library: Lodash)

**Mitigations**

- Avoid insecure recursive merges, see CVE-2018-16487.

- Implement JSON Schema validations for external/untrusted requests.
- Create Objects without prototype by using `Object.create(null)`.
- Freezing the prototype: `Object.freeze(MyObject.prototype)`.
- Disable the `Object.prototype.__proto__` property using `--disable-proto` flag.
- Check that the property exists directly on the object, not from the prototype using `Object.hasOwn(obj, keyFromObj)`.
- Avoid using methods from `Object.prototype`.

### Uncontrolled Search Path Element (CWE-427)

Node.js loads modules following the Module Resolution Algorithm. Therefore, it assumes the directory in which a module is requested (require) is trusted.

By that, it means the following application behavior is expected. Assuming the following directory structure:

- *app/*
  - *server.js*
  - *auth.js*
  - *auth*

If server.js uses `require('./auth')` it will follow the module resolution algorithm and load *auth* instead of *auth.js*.

#### Mitigations

Using the experimental[1] policy mechanism with integrity checking can avoid the above threat. For the directory described above, one can use the following `policy.json`

```
{
  "resources": {
    "./app/auth.js": {
      "integrity": "sha256-iuGZ6SFVFpMuHUcJciQTIKpIyaQVigMZlvg9Lx66HV8="
    },
    "./app/server.js": {
      "dependencies": {
        "./auth": "./app/auth.js"
      },
      "integrity": "sha256-NPtLCQ0ntPPWgfVEgX46ryTNpdvTWdQPoZO3kHo0bKI="
    }
  }
}
```

Therefore, when requiring the *auth* module, the system will validate the integrity and throw an error if doesn't match the expected one.

```
» node --experimental-policy=policy.json app/server.js
node:internal/policy/sri:65
      throw new ERR_SRI_PARSE(str, str[prevIndex], prevIndex);
      ^

SyntaxError [ERR_SRI_PARSE]: Subresource Integrity string "sha256-iuGZ6SFVFpMuHUcJciQTIKpIyaQVigMZlvg9Lx66HV8=%" had an unexpected
    at new NodeError (node:internal/errors:393:5)
    at Object.parse (node:internal/policy/sri:65:13)
    at processEntry (node:internal/policy/manifest:581:38)
    at Manifest.assertIntegrity (node:internal/policy/manifest:588:32)
    at Module._compile (node:internal/modules/cjs/loader:1119:21)
    at Module._extensions..js (node:internal/modules/cjs/loader:1213:10)
    at Module.load (node:internal/modules/cjs/loader:1037:32)
    at Module._load (node:internal/modules/cjs/loader:878:12)
    at Module.require (node:internal/modules/cjs/loader:1061:19)
    at require (node:internal/modules/cjs/helpers:99:18) {
  code: 'ERR_SRI_PARSE'
}
```

Note, it's always recommended the use of `--policy-integrity` to avoid policy mutations.

## Experimental Features in Production

The use of experimental features in production isn't recommended. Experimental features can suffer breaking changes if needed, and their functionality isn't securely stable. Although, feedback is highly appreciated.

## OpenSSF Tools

The OpenSSF is leading several initiatives that can be very useful, especially if you plan to publish an npm package. These initiatives include:

- OpenSSF Scorecard Scorecard evaluates open source projects using a series of automated security risk checks. You can use it to proactively assess vulnerabilities and dependencies in your code base and make informed decisions about accepting vulnerabilities.
- OpenSSF Best Practices Badge Program Projects can voluntarily self-certify by describing how they comply with each best practice. This will generate a badge that can be added to the project.

# TypeScript

## introduction

# Introduction to TypeScript

## What is TypeScript

**TypeScript** is an open-source language maintained and developed by Microsoft.

Basically, TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor or in your CI/CD pipeline, and write more maintainable code.

We can talk about other TypeScript benefits later, let's see some examples now!

## First TypeScript code

Take a look at this code snippet and then we can unpack it together:

```
type User = {
  name: string;
  age: number;
};

function isAdult(user: User): boolean {
  return user.age >= 18;
}

const justine = {
  name: 'Justine',
  age: 23,
} satisfies User;

const isJustineAnAdult = isAdult(justine);
```

The first part (with the `type` keyword) is responsible for declaring our custom object type representing users. Later we utilize this newly created type to create function `isAdult` that accepts one argument of type `User` and returns `boolean`. After this, we create `justine`, our example data that can be used for calling the previously defined function. Finally, we create a new variable with information on whether `justine` is an adult.

There are additional things about this example that you should know. Firstly, if we do not comply with the declared types, TypeScript will inform us that something is wrong and prevent misuse. Secondly, not everything must be typed explicitly—TypeScript infers types for us. For example, the variable `isJustineAnAdult` is of type `boolean` even if we didn't type it explicitly, and `justine` would be a valid argument for our function even though we didn't declare this variable as of `User` type.

## How to run TypeScript code

Okay, so we have some TypeScript code. Now how do we run it? There are few possible ways to run TypeScript code, we will cover all of them in the next articles.

## transpile

# Running TypeScript code using transpilation

Transpilation is the process of converting source code from one language to another. In the case of TypeScript, it's the process of converting TypeScript code to JavaScript code. This is necessary because browsers and Node.js can't run TypeScript code directly.

## Compiling TypeScript to JavaScript

The most common way to run TypeScript code is to compile it to JavaScript first. You can do this using the TypeScript compiler `tsc`.

**Step 1:** Write your TypeScript code in a file, for example `example.ts`.

```
type User = {
  name: string;
  age: number;
};

function isAdult(user: User): boolean {
  return user.age >= 18;
}

const justine = {
  name: 'Justine',
  age: 23,
} satisfies User;

const isJustineAnAdult = isAdult(justine);
```

**Step 2:** Install TypeScript locally using a package manager:

In this example we're going to use npm, you can check our [our introduction to the npm package manager] for more information.

```
npm i -D typescript # -D is a shorthand for --save-dev
```

**Step 3:** Compile your TypeScript code to JavaScript using the `tsc` command:

```
npx tsc example.ts
```

> **NOTE:** `npx` is a tool that allows you to run Node.js packages without installing them globally.

`tsc` is the TypeScript compiler which will take our TypeScript code and compile it to JavaScript. This command will result in a new file named `example.js` that we can run using Node.js. Now when we know how to compile and run TypeScript code let's see TypeScript bug-preventing capabilities in action!

**Step 4:** Run your JavaScript code using Node.js:

```
node example.js
```

You should see the output of your TypeScript code in the terminal

## If there are type errors

If you have type errors in your TypeScript code, the TypeScript compiler will catch them and prevent you from running the code. For example, if you change the `age` property of `justine` to a string, TypeScript will throw an error:

We will modify our code like this, to voluntarily introduce a type error:

```
type User = {
  name: string;
  age: number;
};

function isAdult(user: User): boolean {
  return user.age >= 18;
}

const justine: User = {
  name: 'Justine',
  age: 'Secret!',
};

const isJustineAnAdult: string = isAdult(justine, "I shouldn't be here!");
```

And this is what TypeScript has to say about this:

```
example.ts:12:5 - error TS2322: Type 'string' is not assignable to type 'number'.

12     age: 'Secret!',
       ~~~

  example.ts:3:5
    3     age: number;
          ~~~
    The expected type comes from property 'age' which is declared here on type 'User'

example.ts:15:7 - error TS2322: Type 'boolean' is not assignable to type 'string'.

15 const isJustineAnAdult: string = isAdult(justine, "I shouldn't be here!");
         ~~~~~~~~~~~~~~~~

example.ts:15:51 - error TS2554: Expected 1 arguments, but got 2.

15 const isJustineAnAdult: string = isAdult(justine, "I shouldn't be here!");
                                                     ~~~~~~~~~~~~~~~~~~~~~~


Found 3 errors in the same file, starting at: example.ts:12
```

As you can see, TypeScript is very helpful in catching bugs before they even happen. This is one of the reasons why TypeScript is so popular among developers.

---

## run

# Running TypeScript with a runner

In the previous article, we learned how to run TypeScript code using transpilation. In this article, we will learn how to run TypeScript code using a runner.

## Running TypeScript code with `ts-node`

ts-node is a TypeScript execution environment for Node.js. It allows you to run TypeScript code directly in Node.js without the need to compile it first. Note, however, that it does not type check your code. So we recommend to type check your code first with `tsc` and then run it with `ts-node` before shipping it.

To use `ts-node`, you need to install it first:

```
npm i -D ts-node
```

Then you can run your TypeScript code like this:

```
npx ts-node example.ts
```

## Running TypeScript code with `tsx`

tsx is another TypeScript execution environment for Node.js. It allows you to run TypeScript code directly in Node.js without the need to compile it first. Note, however, that it does not type check your code. So we recommend to type check your code first with `tsc` and then run it with `tsx` before shipping it.

To use `tsx`, you need to install it first:

```
npm i -D tsx
```

Then you can run your TypeScript code like this:

```
npx tsx example.ts
```

### Registering `tsx` via `node`

If you want to use `tsx` via `node`, you can register `tsx` via `--import`:

```
node --import=tsx example.ts
```

## run natively

> ⚠️ **WARNING⚠️ :** All content in this article uses Node.js experimental features. Please make sure you are using a version of Node.js that supports the features mentioned in this article. And remember that experimental features can change in future versions of Node.js.

# Running TypeScript Natively

In the previous articles, we learned how to run TypeScript code using transpilation and with a runner. In this article, we will learn how to run TypeScript code using Node.js itself.

## Running TypeScript code with Node.js

Since V22.6.0, Node.js has experimental support for some TypeScript syntax. You can write code that's valid TypeScript directly in Node.js without the need to transpile it first.

So how do you run TypeScript code with Node.js?

```
node --experimental-strip-types example.ts
```

The `--experimental-strip-types` flag tells Node.js to strip the type annotations from the TypeScript code before running it.

And that's it! You can now run TypeScript code directly in Node.js without the need to transpile it first, and use TypeScript to catch type-related errors. Future versions of Node.js will include support for TypeScript without the need for a command line flag.

## Limitations

At the time of writing, the experimental support for TypeScript in Node.js has some limitations. To allow TypeScript to run in node.js, our collaborators have chosen to only strip types from the code.

You can get more information on the API docs

## Important notes

Thanks to all the contributors who have made this feature possible. We hope that this feature will be stable and available in the LTS version of Node.js soon.

We can understand that this feature is experimental and has some limitations; if that doesn't suit your use-case, please use something else, or contribute a fix. Bug reports are also welcome, please keep in mind the project is run by volunteers, without warranty of any kind, so please be patient if you can't contribute the fix yourself.

# Asynchronous Work

## asynchronous flow control

# Asynchronous flow control

> The material in this post is heavily inspired by Mixu's Node.js Book.

At its core, JavaScript is designed to be non-blocking on the "main" thread, this is where views are rendered. You can imagine the importance of this in the browser. When the main thread becomes blocked it results in the infamous "freezing" that end users dread, and no other events can be dispatched resulting in the loss of data acquisition, for example.

This creates some unique constraints that only a functional style of programming can cure. This is where callbacks come in to the picture.

However, callbacks can become challenging to handle in more complicated procedures. This often results in "callback hell" where multiple nested functions with callbacks make the code more challenging to read, debug, organize, etc.

```
async1(function (input, result1) {
  async2(function (result2) {
    async3(function (result3) {
      async4(function (result4) {
        async5(function (output) {
          // do something with output
        });
      });
    });
  });
});
```

Of course, in real life there would most likely be additional lines of code to handle `result1`, `result2`, etc., thus, the length and complexity of this issue usually results in code that looks much more messy than the example above.

**This is where *functions* come in to great use. More complex operations are made up of many functions:**

1. initiator style / input
2. middleware
3. terminator

**The "initiator style / input" is the first function in the sequence. This function will accept the original input, if any, for the operation. The operation is an executable series of functions, and the original input will primarily be:**

1. variables in a global environment
2. direct invocation with or without arguments
3. values obtained by file system or network requests

Network requests can be incoming requests initiated by a foreign network, by another application on the same network, or by the app itself on the same or foreign network.

A middleware function will return another function, and a terminator function will invoke the callback. The following illustrates the flow to network or file system requests. Here the latency is 0 because all these values are available in memory.

```
function final(someInput, callback) {
  callback(`${someInput} and terminated by executing callback `);
}

function middleware(someInput, callback) {
  return final(`${someInput} touched by middleware `, callback);
}

function initiate() {
  const someInput = 'hello this is a function ';
  middleware(someInput, function (result) {
    console.log(result);
    // requires callback to `return` result
  });
}
```

```
initiate();
```

## State management

Functions may or may not be state dependent. State dependency arises when the input or other variable of a function relies on an outside function.

**In this way there are two primary strategies for state management:**

1. passing in variables directly to a function, and
2. acquiring a variable value from a cache, session, file, database, network, or other outside source.

Note, I did not mention global variable. Managing state with global variables is often a sloppy anti-pattern that makes it difficult or impossible to guarantee state. Global variables in complex programs should be avoided when possible.

## Control flow

If an object is available in memory, iteration is possible, and there will not be a change to control flow:

```
function getSong() {
  let _song = '';
  let i = 100;
  for (i; i > 0; i -= 1) {
    _song += `${i} beers on the wall, you take one down and pass it around, ${
      i - 1
    } bottles of beer on the wall\n`;
    if (i === 1) {
      _song += "Hey let's get some more beer";
    }
  }

  return _song;
}

function singSong(_song) {
  if (!_song) throw new Error("song is '' empty, FEED ME A SONG!");
  console.log(_song);
}

const song = getSong();
// this will work
singSong(song);
```

However, if the data exists outside of memory the iteration will no longer work:

```
function getSong() {
  let _song = '';
  let i = 100;
  for (i; i > 0; i -= 1) {
    /* eslint-disable no-loop-func */
    setTimeout(function () {
      _song += `${i} beers on the wall, you take one down and pass it around, ${
        i - 1
      } bottles of beer on the wall\n`;
      if (i === 1) {
        _song += "Hey let's get some more beer";
      }
    }, 0);
    /* eslint-enable no-loop-func */
  }

  return _song;
}

function singSong(_song) {
  if (!_song) throw new Error("song is '' empty, FEED ME A SONG!");
  console.log(_song);
}

const song = getSong('beer');
```

```
// this will not work
singSong(song);
// Uncaught Error: song is '' empty, FEED ME A SONG!
```

Why did this happen? `setTimeout` instructs the CPU to store the instructions elsewhere on the bus, and instructs that the data is scheduled for pickup at a later time. Thousands of CPU cycles pass before the function hits again at the 0 millisecond mark, the CPU fetches the instructions from the bus and executes them. The only problem is that song ('') was returned thousands of cycles prior.

The same situation arises in dealing with file systems and network requests. The main thread simply cannot be blocked for an indeterminate period of time-- therefore, we use callbacks to schedule the execution of code in time in a controlled manner.

You will be able to perform almost all of your operations with the following 3 patterns:

1. **In series:** functions will be executed in a strict sequential order, this one is most similar to `for` loops.

```
// operations defined elsewhere and ready to execute
const operations = [
  { func: function1, args: args1 },
  { func: function2, args: args2 },
  { func: function3, args: args3 },
];

function executeFunctionWithArgs(operation, callback) {
  // executes function
  const { args, func } = operation;
  func(args, callback);
}

function serialProcedure(operation) {
  if (!operation) process.exit(0); // finished
  executeFunctionWithArgs(operation, function (result) {
    // continue AFTER callback
    serialProcedure(operations.shift());
  });
}

serialProcedure(operations.shift());
```

2. **Full parallel:** when ordering is not an issue, such as emailing a list of 1,000,000 email recipients.

```
let count = 0;
let success = 0;
const failed = [];
const recipients = [
  { name: 'Bart', email: 'bart@tld' },
  { name: 'Marge', email: 'marge@tld' },
  { name: 'Homer', email: 'homer@tld' },
  { name: 'Lisa', email: 'lisa@tld' },
  { name: 'Maggie', email: 'maggie@tld' },
];

function dispatch(recipient, callback) {
  // `sendEmail` is a hypothetical SMTP client
  sendMail(
    {
      subject: 'Dinner tonight',
      message: 'We have lots of cabbage on the plate. You coming?',
      smtp: recipient.email,
    },
    callback
  );
}

function final(result) {
  console.log(`Result: ${result.count} attempts \
      & ${result.success} succeeded emails`);
  if (result.failed.length)
    console.log(`Failed to send to: \
        \n${result.failed.join('\n')}\n`);
}

recipients.forEach(function (recipient) {
  dispatch(recipient, function (err) {
    if (!err) {
      success += 1;
    } else {
      failed.push(recipient.name);
    }
    count += 1;

    if (count === recipients.length) {
      final({
        count,
        success,
        failed,
      });
    }
  });
});
```

3. **Limited parallel:** parallel with limit, such as successfully emailing 1,000,000 recipients from a list of 10 million users.

```
let successCount = 0;

function final() {
  console.log(`dispatched ${successCount} emails`);
  console.log('finished');
}

function dispatch(recipient, callback) {
  // `sendEmail` is a hypothetical SMTP client
  sendMail(
    {
      subject: 'Dinner tonight',
      message: 'We have lots of cabbage on the plate. You coming?',
      smtp: recipient.email,
    },
    callback
  );
}

function sendOneMillionEmailsOnly() {
  getListOfTenMillionGreatEmails(function (err, bigList) {
    if (err) throw err;

    function serial(recipient) {
      if (!recipient || successCount >= 1000000) return final();
      dispatch(recipient, function (_err) {
        if (!_err) successCount += 1;
        serial(bigList.pop());
      });
    }

    serial(bigList.pop());
  });
}

sendOneMillionEmailsOnly();
```

Each has its own use cases, benefits, and issues you can experiment and read about in more detail. Most importantly, remember to modularize your operations and use callbacks! If you feel any doubt, treat everything as if it were middleware!

## overview of blocking vs non blocking

# Overview of Blocking vs Non-Blocking

This overview covers the difference between **blocking** and **non-blocking** calls in Node.js. This overview will refer to the event loop and libuv but no prior knowledge of those topics is required. Readers are assumed to have a basic understanding of the JavaScript language and Node.js callback pattern.

> "I/O" refers primarily to interaction with the system's disk and network supported by libuv.

## Blocking

**Blocking** is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as **blocking**. Synchronous methods in the Node.js standard library that use libuv are the most commonly used **blocking** operations. Native modules may also have **blocking** methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are **non-blocking**, and accept callback functions. Some methods also have **blocking** counterparts, which have names that end with `Sync`.

## Comparing Code

**Blocking** methods execute **synchronously** and **non-blocking** methods execute **asynchronously**.

Using the File System module as an example, this is a **synchronous** file read:

```
const fs = require('node:fs');

const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

And here is an equivalent **asynchronous** example:

```
const fs = require('node:fs');

fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

The first example appears simpler than the second but has the disadvantage of the second line **blocking** the execution of any additional JavaScript until the entire file is read. Note that in the synchronous version if an error is thrown it will need to be caught or the process will crash. In the asynchronous version, it is up to the author to decide whether an error should throw as shown.

Let's expand our example a little bit:

```
const fs = require('node:fs');

const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

And here is a similar, but not equivalent asynchronous example:

```
const fs = require('node:fs');

fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

In the first example above, `console.log` will be called before `moreWork()`. In the second example `fs.readFile()` is **non-blocking** so JavaScript execution can continue and `moreWork()` will be called first. The ability to run `moreWork()` without waiting for the file read to complete is a key design choice that allows for higher throughput.

## Concurrency and Throughput

JavaScript execution in Node.js is single threaded, so concurrency refers to the event loop's capacity to execute JavaScript callback functions after completing other work. Any code that is expected to run in a concurrent manner must allow the event loop to continue running as non-JavaScript operations, like I/O, are occurring.

As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing **non-blocking** asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use **non-blocking** methods instead of **blocking** methods.

The event loop is different than models in many other languages where additional threads may be created to handle concurrent work.

## Dangers of Mixing Blocking and Non-Blocking Code

There are some patterns that should be avoided when dealing with I/O. Let's look at an example:

```
const fs = require('node:fs');

fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
fs.unlinkSync('/file.md');
```

In the above example, `fs.unlinkSync()` is likely to be run before `fs.readFile()`, which would delete `file.md` before it is actually read. A better way to write this, which is completely **non-blocking** and guaranteed to execute in the correct order is:

```
const fs = require('node:fs');

fs.readFile('/file.md', (readFileErr, data) => {
  if (readFileErr) throw readFileErr;
  console.log(data);
  fs.unlink('/file.md', unlinkErr => {
    if (unlinkErr) throw unlinkErr;
  });
});
```

The above places a **non-blocking** call to `fs.unlink()` within the callback of `fs.readFile()` which guarantees the correct order of operations.

## Additional Resources

- libuv

## javascript asynchronous programming and callbacks

# JavaScript Asynchronous Programming and Callbacks

## Asynchronicity in Programming Languages

Computers are asynchronous by design.

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

In the current consumer computers, every program runs for a specific time slot and then it stops its execution to let another program continue their execution. This thing runs in a cycle so fast that it's impossible to notice. We think our computers run many programs simultaneously, but this is an illusion (except on multiprocessor machines).

Programs internally use *interrupts*, a signal that's emitted to the processor to gain the attention of the system.

Let's not go into the internals of this now, but just keep in mind that it's normal for programs to be asynchronous and halt their execution until they need attention, allowing the computer to execute other things in the meantime. When a program is waiting for a response from the network, it cannot halt the processor until the request finishes.

Normally, programming languages are synchronous and some provide a way to manage asynchronicity in the language or through libraries. C, Java, C#, PHP, Go, Ruby, Swift, and Python are all synchronous by default. Some of them handle async operations by using threads, spawning a new process.

## JavaScript

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

Lines of code are executed in series, one after another, for example:

```
const a = 1;
const b = 2;
const c = a * b;
console.log(c);
doSomething();
```

But JavaScript was born inside the browser, its main job, in the beginning, was to respond to user actions, like `onClick`, `onMouseOver`, `onChange`, `onSubmit` and so on. How could it do this with a synchronous programming model?

The answer was in its environment. The **browser** provides a way to do it by providing a set of APIs that can handle this kind of functionality.

More recently, Node.js introduced a non-blocking I/O environment to extend this concept to file access, network calls and so on.

## Callbacks

You can't know when a user is going to click a button. So, you **define an event handler for the click event**. This event handler accepts a function, which will be called when the event is triggered:

```
document.getElementById('button').addEventListener('click', () => {
  // item clicked
});
```

This is the so-called **callback**.

A callback is a simple function that's passed as a value to another function, and will only be executed when the event happens. We can do this because JavaScript has first-class functions, which can be assigned to variables and passed around to other functions (called **higher-order functions**)

It's common to wrap all your client code in a `load` event listener on the `window` object, which runs the callback function only when the page is ready:

```
window.addEventListener('load', () => {
  // window loaded
  // do what you want
});
```

Callbacks are used everywhere, not just in DOM events.

One common example is by using timers:

```
setTimeout(() => {
  // runs after 2 seconds
}, 2000);
```

XHR requests also accept a callback, in this example by assigning a function to a property that will be called when a particular event occurs (in this case, the state of the request changes):

```
const xhr = new XMLHttpRequest();
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    xhr.status === 200 ? console.log(xhr.responseText) : console.error('error');
  }
};
xhr.open('GET', 'https://yoursite.com');
xhr.send();
```

### Handling errors in callbacks

How do you handle errors with callbacks? One very common strategy is to use what Node.js adopted: the first parameter in any callback function is the error object: **error-first callbacks**

If there is no error, the object is `null`. If there is an error, it contains some description of the error and other information.

```
const fs = require('node:fs');

fs.readFile('/file.json', (err, data) => {
  if (err) {
    // handle error
    console.log(err);
    return;
  }

  // no errors, process data
  console.log(data);
});
```

### The problem with callbacks

Callbacks are great for simple cases!

However every callback adds a level of nesting, and when you have lots of callbacks, the code starts to be complicated very quickly:

```
window.addEventListener('load', () => {
  document.getElementById('button').addEventListener('click', () => {
    setTimeout(() => {
      items.forEach(item => {
        // your code here
      });
    }, 2000);
  });
});
```

This is just a simple 4-levels code, but I've seen much more levels of nesting and it's not fun.

How do we solve this?

### Alternatives to callbacks

Starting with ES6, JavaScript introduced several features that help us with asynchronous code that do not involve using callbacks: Promises (ES6) and Async/Await (ES2017).

# Discover JavaScript Timers

## `setTimeout()`

When writing JavaScript code, you might want to delay the execution of a function.

This is the job of `setTimeout`. You specify a callback function to execute later, and a value expressing how later you want it to run, in milliseconds:

```
setTimeout(() => {
  // runs after 2 seconds
}, 2000);

setTimeout(() => {
  // runs after 50 milliseconds
}, 50);
```

This syntax defines a new function. You can call whatever other function you want in there, or you can pass an existing function name, and a set of parameters:

```
const myFunction = (firstParam, secondParam) => {
  // do something
};

// runs after 2 seconds
setTimeout(myFunction, 2000, firstParam, secondParam);
```

`setTimeout` returns the timer id. This is generally not used, but you can store this id, and clear it if you want to delete this scheduled function execution:

```
const id = setTimeout(() => {
  // should run after 2 seconds
}, 2000);

// I changed my mind
clearTimeout(id);
```

### Zero delay

If you specify the timeout delay to `0`, the callback function will be executed as soon as possible, but after the current function execution:

```
setTimeout(() => {
  console.log('after ');
}, 0);

console.log(' before ');
```

This code will print

```
before
after
```

This is especially useful to avoid blocking the CPU on intensive tasks and let other functions be executed while performing a heavy calculation, by queuing functions in the scheduler.

> Some browsers (IE and Edge) implement a `setImmediate()` method that does this same exact functionality, but it's not standard and unavailable on other browsers. But it's a standard function in Node.js.

## `setInterval()`

`setInterval` is a function similar to `setTimeout`, with a difference: instead of running the callback function once, it will run it forever, at the specific time interval you specify (in milliseconds):

```
setInterval(() => {
  // runs every 2 seconds
}, 2000);
```

The function above runs every 2 seconds unless you tell it to stop, using `clearInterval`, passing it the interval id that `setInterval` returned:

```
const id = setInterval(() => {
  // runs every 2 seconds
}, 2000);

clearInterval(id);
```

It's common to call `clearInterval` inside the setInterval callback function, to let it auto-determine if it should run again or stop. For example this code runs something unless App.somethingIWait has the value `arrived`:

```
const interval = setInterval(() => {
  if (App.somethingIWait === 'arrived') {
    clearInterval(interval);
  }
  // otherwise do things
}, 100);
```

## Recursive setTimeout

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

If a function always takes the same amount of time, it's all fine:

setInterval working fine

Maybe the function takes different execution times, depending on network conditions for example:

setInterval varying duration

And maybe one long execution overlaps the next one:

setInterval overlapping

To avoid this, you can schedule a recursive setTimeout to be called when the callback function finishes:

```
const myFunction = () => {
  // do something

  setTimeout(myFunction, 1000);
};

setTimeout(myFunction, 1000);
```

to achieve this scenario:

Recursive setTimeout

`setTimeout` and `setInterval` are available in Node.js, through the Timers module.

Node.js also provides `setImmediate()`, which is equivalent to using `setTimeout(() => {}, 0)`, mostly used to work with the Node.js Event Loop.

## event loop timers and nexttick

# The Node.js Event Loop

## What is the Event Loop?

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that a single JavaScript thread is used by default — by offloading operations to the system kernel whenever possible.

Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the **poll** queue to eventually be executed. We'll explain this in further detail later in this topic.

## Event Loop Explained

When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL, which is not covered in this document) which may make async API calls, schedule timers, or call `process.nextTick()`, then begins processing the event loop.

The following diagram shows a simplified overview of the event loop's order of operations.

```
   ┌───────────────────────────┐
┌─>│           timers          │
│  └─────────────┬─────────────┘
│  ┌─────────────┴─────────────┐
│  │     pending callbacks     │
│  └─────────────┬─────────────┘
│  ┌─────────────┴─────────────┐
│  │       idle, prepare       │
│  └─────────────┬─────────────┘      ┌───────────────┐
│  ┌─────────────┴─────────────┐      │   incoming:   │
│  │           poll            │<─────┤  connections, │
│  └─────────────┬─────────────┘      │   data, etc.  │
│  ┌─────────────┴─────────────┐      └───────────────┘
│  │           check           │
│  └─────────────┬─────────────┘
│  ┌─────────────┴─────────────┐
└──┤      close callbacks      │
   └───────────────────────────┘
```

> Each box will be referred to as a "phase" of the event loop.

Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

Since any of these operations may schedule *more* operations and new events processed in the **poll** phase are queued by the kernel, poll events can be queued while polling events are being processed. As a result, long running callbacks can allow the poll phase to run much longer than a timer's threshold. See the **timers** and **poll** sections for more details.

> There is a slight discrepancy between the Windows and the Unix/Linux implementation, but that's not important for this demonstration. The most important parts are here. There are actually seven or eight steps, but the ones we care about — ones that Node.js actually uses - are those above.

## Phases Overview

- **timers**: this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.
- **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
- **idle, prepare**: only used internally.
- **poll**: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.
- **check**: `setImmediate()` callbacks are invoked here.
- **close callbacks**: some close callbacks, e.g. `socket.on('close', ...)`.

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

## Phases in Detail

### timers

A timer specifies the **threshold** *after which* a provided callback *may be executed* rather than the **exact** time a person *wants it to be executed*. Timers callbacks will run as early as they can be scheduled after the specified amount of time has passed; however, Operating System scheduling or the running of other callbacks may delay them.

> Technically, the **poll** phase controls when timers are executed.

For example, say you schedule a timeout to execute after a 100 ms threshold, then your script starts asynchronously reading a file which takes 95 ms:

```
const fs = require('node:fs');

function someAsyncOperation(callback) {
  // Assume this takes 95ms to complete
  fs.readFile('/path/to/file', callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;

  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);

// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();

  // do something that will take 10ms...
  while (Date.now() - startCallback < 10) {
    // do nothing
  }
});
```

When the event loop enters the **poll** phase, it has an empty queue (`fs.readFile()` has not completed), so it will wait for the number of ms remaining until the soonest timer's threshold is reached. While it is waiting 95 ms pass, `fs.readFile()` finishes reading the file and its callback which takes 10 ms to complete is added to the **poll** queue and executed. When the callback finishes, there are no more callbacks in the queue, so the event loop will see that the threshold of the soonest timer has been reached then wrap back to the **timers** phase to execute the timer's callback. In this example, you will see that the total delay between the timer being scheduled and its callback being executed will be 105ms.

> To prevent the **poll** phase from starving the event loop, libuv (the C library that implements the Node.js event loop and all of the asynchronous behaviors of the platform) also has a hard maximum (system dependent) before it stops polling for more events.

### pending callbacks

This phase executes callbacks for some system operations such as types of TCP errors. For example if a TCP socket receives `ECONNREFUSED` when attempting to connect, some *nix systems want to wait to report the error. This will be queued to execute in the **pending callbacks** phase.

### poll

The **poll** phase has two main functions:

1. Calculating how long it should block and poll for I/O, then
2. Processing events in the **poll** queue.

When the event loop enters the **poll** phase *and there are no timers scheduled*, one of two things will happen:

- *If the **poll** queue **is not empty***, the event loop will iterate through its queue of callbacks executing them synchronously until either the queue has been exhausted, or the system-dependent hard limit is reached.

- *If the **poll** queue **is empty***, one of two more things will happen:
  - If scripts have been scheduled by `setImmediate()`, the event loop will end the **poll** phase and continue to the **check** phase to execute those scheduled scripts.
  - If scripts **have not** been scheduled by `setImmediate()`, the event loop will wait for callbacks to be added to the queue, then execute them immediately.

Once the **poll** queue is empty the event loop will check for timers *whose time thresholds have been reached*. If one or more timers are ready, the event loop will wrap back to the **timers** phase to execute those timers' callbacks.

### check

This phase allows a person to execute callbacks immediately after the **poll** phase has completed. If the **poll** phase becomes idle and scripts have been queued with `setImmediate()`, the event loop may continue to the **check** phase rather than waiting.

`setImmediate()` is actually a special timer that runs in a separate phase of the event loop. It uses a libuv API that schedules callbacks to execute after the **poll** phase has completed.

Generally, as the code is executed, the event loop will eventually hit the **poll** phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with `setImmediate()` and the **poll** phase becomes idle, it will end and continue to the **check** phase rather than waiting for **poll** events.

### close callbacks

If a socket or handle is closed abruptly (e.g. `socket.destroy()`), the `'close'` event will be emitted in this phase. Otherwise it will be emitted via `process.nextTick()`.

## `setImmediate()` vs `setTimeout()`

`setImmediate()` and `setTimeout()` are similar, but behave in different ways depending on when they are called.

- `setImmediate()` is designed to execute a script once the current **poll** phase completes.
- `setTimeout()` schedules a script to be run after a minimum threshold in ms has elapsed.

The order in which the timers are executed will vary depending on the context in which they are called. If both are called from within the main module, then timing will be bound by the performance of the process (which can be impacted by other applications running on the machine).

For example, if we run the following script which is not within an I/O cycle (i.e. the main module), the order in which the two timers are executed is non-deterministic, as it is bound by the performance of the process:

```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);

setImmediate(() => {
  console.log('immediate');
});
```

```
$ node timeout_vs_immediate.js
timeout
immediate

$ node timeout_vs_immediate.js
immediate
timeout
```

However, if you move the two calls within an I/O cycle, the immediate callback is always executed first:

```
// timeout_vs_immediate.js
const fs = require('node:fs');

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
```

```
$ node timeout_vs_immediate.js
immediate
timeout

$ node timeout_vs_immediate.js
immediate
timeout
```

The main advantage to using `setImmediate()` over `setTimeout()` is `setImmediate()` will always be executed before any timers if scheduled within an I/O cycle, independently of how many timers are present.

## `process.nextTick()`

### Understanding `process.nextTick()`

You may have noticed that `process.nextTick()` was not displayed in the diagram, even though it's a part of the asynchronous API. This is because `process.nextTick()` is not technically part of the event loop. Instead, the `nextTickQueue` will be processed after the current operation is completed, regardless of the current phase of the event loop. Here, an *operation* is defined as a transition from the underlying C/C++ handler, and handling the JavaScript that needs to be executed.

Looking back at our diagram, any time you call `process.nextTick()` in a given phase, all callbacks passed to `process.nextTick()` will be resolved before the event loop continues. This can create some bad situations because **it allows you to "starve" your I/O by making recursive `process.nextTick()` calls**, which prevents the event loop from reaching the **poll** phase.

### Why would that be allowed?

Why would something like this be included in Node.js? Part of it is a design philosophy where an API should always be asynchronous even where it doesn't have to be. Take this code snippet for example:

```
function apiCall(arg, callback) {
  if (typeof arg !== 'string')
    return process.nextTick(
      callback,
      new TypeError('argument should be string')
    );
}
```

The snippet does an argument check and if it's not correct, it will pass the error to the callback. The API updated fairly recently to allow passing arguments to `process.nextTick()` allowing it to take any arguments passed after the callback to be propagated as the arguments to the callback so you don't have to nest functions.

What we're doing is passing an error back to the user but only *after* we have allowed the rest of the user's code to execute. By using `process.nextTick()` we guarantee that `apiCall()` always runs its callback *after* the rest of the user's code and *before* the event loop is allowed to proceed. To achieve this, the JS call stack is allowed to unwind then immediately execute the provided callback which allows a person to make recursive calls to `process.nextTick()` without reaching a `RangeError: Maximum call stack size exceeded from v8`.

This philosophy can lead to some potentially problematic situations. Take this snippet for example:

```
let bar;

// this has an asynchronous signature, but calls callback synchronously
function someAsyncApiCall(callback) {
  callback();
}

// the callback is called before `someAsyncApiCall` completes.
someAsyncApiCall(() => {
  // since someAsyncApiCall hasn't completed, bar hasn't been assigned any value
  console.log('bar', bar); // undefined
});

bar = 1;
```

The user defines `someAsyncApiCall()` to have an asynchronous signature, but it actually operates synchronously. When it is called, the callback provided to `someAsyncApiCall()` is called in the same phase of the event loop because `someAsyncApiCall()` doesn't actually do anything asynchronously. As a result, the callback tries to reference `bar` even though it may not have that variable in scope yet, because the script has not been able to run to completion.

By placing the callback in a `process.nextTick()`, the script still has the ability to run to completion, allowing all the variables, functions, etc., to be initialized prior to the callback being called. It also has the advantage of not allowing the event loop to continue. It may be useful for the user to be alerted to an error before the event loop is allowed to continue. Here is the previous example using `process.nextTick()`:

```
let bar;

function someAsyncApiCall(callback) {
  process.nextTick(callback);
}

someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});

bar = 1;
```

Here's another real world example:

```
const server = net.createServer(() => {}).listen(8080);

server.on('listening', () => {});
```

When only a port is passed, the port is bound immediately. So, the `'listening'` callback could be called immediately. The problem is that the `.on('listening')` callback will not have been set by that time.

To get around this, the `'listening'` event is queued in a `nextTick()` to allow the script to run to completion. This allows the user to set any event handlers they want.

### `process.nextTick()` vs `setImmediate()`

We have two calls that are similar as far as users are concerned, but their names are confusing.

- `process.nextTick()` fires immediately on the same phase
- `setImmediate()` fires on the following iteration or 'tick' of the event loop

In essence, the names should be swapped. `process.nextTick()` fires more immediately than `setImmediate()`, but this is an artifact of the past which is unlikely to change. Making this switch would break a large percentage of the packages on npm. Every day more new modules are being added, which means every day we wait, more potential breakages occur. While they are confusing, the names themselves won't change.

> We recommend developers use `setImmediate()` in all cases because it's easier to reason about.

## Why use `process.nextTick()`?

There are two main reasons:

1. Allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues.

2. At times it's necessary to allow a callback to run after the call stack has unwound but before the event loop continues.

One example is to match the user's expectations. Simple example:

```
const server = net.createServer();
server.on('connection', conn => {});

server.listen(8080);
server.on('listening', () => {});
```

Say that `listen()` is run at the beginning of the event loop, but the listening callback is placed in a `setImmediate()`. Unless a hostname is passed, binding to the port will happen immediately. For the event loop to proceed, it must hit the **poll** phase, which means there is a non-zero chance that a connection could have been received allowing the connection event to be fired before the listening event.

Another example is extending an `EventEmitter` and emitting an event from within the constructor:

```
const EventEmitter = require('node:events');

class MyEmitter extends EventEmitter {
  constructor() {
    super();
    this.emit('event');
  }
}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

You can't emit an event from the constructor immediately because the script will not have processed to the point where the user assigns a callback to that event. So, within the constructor itself, you can use `process.nextTick()` to set a callback to emit the event after the constructor has finished, which provides the expected results:

```
const EventEmitter = require('node:events');

class MyEmitter extends EventEmitter {
  constructor() {
    super();

    // use nextTick to emit the event once a handler is assigned
    process.nextTick(() => {
      this.emit('event');
    });
  }
}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

---

# The Node.js Event emitter

If you worked with JavaScript in the browser, you know how much of the interaction of the user is handled through events: mouse clicks, keyboard button presses, reacting to mouse movements, and so on.

On the backend side, Node.js offers us the option to build a similar system using the `events` module.

This module, in particular, offers the `EventEmitter` class, which we'll use to handle our events.

You initialize that using

```
const EventEmitter = require('node:events');

const eventEmitter = new EventEmitter();
```

```
import EventEmitter from 'node:events';

const eventEmitter = new EventEmitter();
```

This object exposes, among many others, the `on` and `emit` methods.

- `emit` is used to trigger an event
- `on` is used to add a callback function that's going to be executed when the event is triggered

For example, let's create a `start` event, and as a matter of providing a sample, we react to that by just logging to the console:

```
eventEmitter.on('start', () => {
  console.log('started');
});
```

When we run

```
eventEmitter.emit('start');
```

the event handler function is triggered, and we get the console log.

You can pass arguments to the event handler by passing them as additional arguments to `emit()`:

```
eventEmitter.on('start', number => {
  console.log(`started ${number}`);
});

eventEmitter.emit('start', 23);
```

Multiple arguments:

```
eventEmitter.on('start', (start, end) => {
  console.log(`started from ${start} to ${end}`);
});

eventEmitter.emit('start', 1, 100);
```

The EventEmitter object also exposes several other methods to interact with events, like

- `once()`: add a one-time listener
- `removeListener()` / `off()`: remove an event listener from an event
- `removeAllListeners()`: remove all listeners for an event

You can read more about these methods in the official documentation.

## understanding processnexttick

# Understanding process.nextTick()

As you try to understand the Node.js event loop, one important part of it is `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:

```
process.nextTick(() => {
  // do something
});
```

The event loop is busy processing the current function code. When this operation ends, the JS engine runs all the functions passed to `nextTick` calls during that operation.

It's the way we can tell the JS engine to process a function asynchronously (after the current function), but as soon as possible, not queue it.

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.

**An Example of the order of events:**

```
console.log('Hello => number 1');

setImmediate(() => {
  console.log('Running before the timeout => number 3');
});

setTimeout(() => {
  console.log('The timeout running last => number 4');
}, 0);

process.nextTick(() => {
  console.log('Running at next tick => number 2');
});
```

**Example output:**

```
Hello => number 1
Running at next tick => number 2
Running before the timeout => number 3
The timeout running last => number 4
```

The exact output may differ from run to run.

## understanding setimmediate

# Understanding 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:

```
setImmediate(() => {
  // run something
});
```

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

How is `setImmediate()` different from `setTimeout(() => {}, 0)` (passing a 0ms timeout), and from `process.nextTick()` and `Promise.then()`?

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.

A `process.nextTick` callback is added to `process.nextTick queue`. A `Promise.then()` callback is added to `promises microtask queue`. A `setTimeout`, `setImmediate` callback is added to `macrotask queue`.

Event loop executes tasks in `process.nextTick queue` first, and then executes `promises microtask queue`, and then executes `macrotask queue`.

Here is an example to show the order between `setImmediate()`, `process.nextTick()` and `Promise.then()`:

```
const baz = () => console.log('baz');
const foo = () => console.log('foo');
const zoo = () => console.log('zoo');

const start = () => {
  console.log('start');
  setImmediate(baz);
  new Promise((resolve, reject) => {
    resolve('bar');
  }).then(resolve => {
    console.log(resolve);
    process.nextTick(zoo);
  });
  process.nextTick(foo);
};

start();

// start foo bar zoo baz
```

This code will first call `start()`, then call `foo()` in `process.nextTick queue`. After that, it will handle `promises microtask queue`, which prints `bar` and adds `zoo()` in `process.nextTick queue` at the same time. Then it will call `zoo()` which has just been added. In the end, the `baz()` in `macrotask queue` is called.

The principle aforementioned holds true in CommonJS cases, but keep in mind in ES Modules, e.g. `mjs` files, the execution order will be different:

```
// start bar foo zoo baz
```

This is because the ES Module being loaded is wrapped as an asynchronous operation, and thus the entire script is actually already in the `promises microtask queue`. So when the promise is immediately resolved, its callback is appended to the `microtask` queue. Node.js will attempt to clear the queue until moving to any other queue, and hence you will see it outputs `bar` first.

# Don't Block the Event Loop (or the Worker Pool)

## Should you read this guide?

If you're writing anything more complicated than a brief command-line script, reading this should help you write higher-performance, more-secure applications.

This document is written with Node.js servers in mind, but the concepts apply to complex Node.js applications as well. Where OS-specific details vary, this document is Linux-centric.

## Summary

Node.js runs JavaScript code in the Event Loop (initialization and callbacks), and offers a Worker Pool to handle expensive tasks like file I/O. Node.js scales well, sometimes better than more heavyweight approaches like Apache. The secret to the scalability of Node.js is that it uses a small number of threads to handle many clients. If Node.js can make do with fewer threads, then it can spend more of your system's time and memory working on clients rather than on paying space and time overheads for threads (memory, context-switching). But because Node.js has only a few threads, you must structure your application to use them wisely.

Here's a good rule of thumb for keeping your Node.js server speedy: *Node.js is fast when the work associated with each client at any given time is "small"*.

This applies to callbacks on the Event Loop and tasks on the Worker Pool.

## Why should I avoid blocking the Event Loop and the Worker Pool?

Node.js uses a small number of threads to handle many clients. In Node.js there are two types of threads: one Event Loop (aka the main loop, main thread, event thread, etc.), and a pool of `k` Workers in a Worker Pool (aka the threadpool).

If a thread is taking a long time to execute a callback (Event Loop) or a task (Worker), we call it "blocked". While a thread is blocked working on behalf of one client, it cannot handle requests from any other clients. This provides two motivations for blocking neither the Event Loop nor the Worker Pool:

1. Performance: If you regularly perform heavyweight activity on either type of thread, the *throughput* (requests/second) of your server will suffer.
2. Security: If it is possible that for certain input one of your threads might block, a malicious client could submit this "evil input", make your threads block, and keep them from working on other clients. This would be a Denial of Service attack.

## A quick review of Node

Node.js uses the Event-Driven Architecture: it has an Event Loop for orchestration and a Worker Pool for expensive tasks.

### What code runs on the Event Loop?

When they begin, Node.js applications first complete an initialization phase, `require`'ing modules and registering callbacks for events. Node.js applications then enter the Event Loop, responding to incoming client requests by executing the appropriate callback. This callback executes synchronously, and may register asynchronous requests to continue processing after it completes. The callbacks for these asynchronous requests will also be executed on the Event Loop.

The Event Loop will also fulfill the non-blocking asynchronous requests made by its callbacks, e.g., network I/O.

In summary, the Event Loop executes the JavaScript callbacks registered for events, and is also responsible for fulfilling non-blocking asynchronous requests like network I/O.

### What code runs on the Worker Pool?

The Worker Pool of Node.js is implemented in libuv (docs), which exposes a general task submission API.

Node.js uses the Worker Pool to handle "expensive" tasks. This includes I/O for which an operating system does not provide a non-blocking version, as well as particularly CPU-intensive tasks.

These are the Node.js module APIs that make use of this Worker Pool:

1. I/O-intensive
   1. DNS: `dns.lookup()`, `dns.lookupService()`.
   2. File System: All file system APIs except `fs.FSWatcher()` and those that are explicitly synchronous use libuv's threadpool.
2. CPU-intensive
   1. Crypto: `crypto.pbkdf2()`, `crypto.scrypt()`, `crypto.randomBytes()`, `crypto.randomFill()`, `crypto.generateKeyPair()`.
   2. Zlib: All zlib APIs except those that are explicitly synchronous use libuv's threadpool.

In many Node.js applications, these APIs are the only sources of tasks for the Worker Pool. Applications and modules that use a C++ add-on can submit other tasks to the Worker Pool.

For the sake of completeness, we note that when you call one of these APIs from a callback on the Event Loop, the Event Loop pays some minor setup costs as it enters the Node.js C++ bindings for that API and submits a task to the Worker Pool. These costs are negligible compared to the overall cost of the task, which is why the Event Loop is offloading it. When submitting one of these tasks to the Worker Pool, Node.js provides a pointer to the corresponding C++ function in the Node.js C++ bindings.

### How does Node.js decide what code to run next?

Abstractly, the Event Loop and the Worker Pool maintain queues for pending events and pending tasks, respectively.

In truth, the Event Loop does not actually maintain a queue. Instead, it has a collection of file descriptors that it asks the operating system to monitor, using a mechanism like epoll (Linux), kqueue (OSX), event ports (Solaris), or IOCP (Windows). These file descriptors correspond to network sockets, any files it is watching, and so on. When the operating system says that one of these file descriptors is ready, the Event Loop translates it to the appropriate event and invokes the callback(s) associated with that event. You can learn more about this process here.

In contrast, the Worker Pool uses a real queue whose entries are tasks to be processed. A Worker pops a task from this queue and works on it, and when finished the Worker raises an "At least one task is finished" event for the Event Loop.

### What does this mean for application design?

In a one-thread-per-client system like Apache, each pending client is assigned its own thread. If a thread handling one client blocks, the operating system will interrupt it and give another client a turn. The operating system thus ensures that clients that require a small amount of work are not penalized by clients that require more work.

Because Node.js handles many clients with few threads, if a thread blocks handling one client's request, then pending client requests may not get a turn until the thread finishes its callback or task. *The fair treatment of clients is thus the responsibility of your application*. This means that you shouldn't do too much work for any client in any single callback or task.

This is part of why Node.js can scale well, but it also means that you are responsible for ensuring fair scheduling. The next sections talk about how to ensure fair scheduling for the Event Loop and for the Worker Pool.

## Don't block the Event Loop

The Event Loop notices each new client connection and orchestrates the generation of a response. All incoming requests and outgoing responses pass through the Event Loop. This means that if the Event Loop spends too long at any point, all current and new clients will not get a turn.

You should make sure you never block the Event Loop. In other words, each of your JavaScript callbacks should complete quickly. This of course also applies to your `await`'s, your `Promise.then`'s, and so on.

A good way to ensure this is to reason about the "computational complexity" of your callbacks. If your callback takes a constant number of steps no matter what its arguments are, then you'll always give every pending client a fair turn. If your callback takes a different number of steps depending on its arguments, then you should think about how long the arguments might be.

Example 1: A constant-time callback.

```
app.get('/constant-time', (req, res) => {
  res.sendStatus(200);
});
```

Example 2: An `O(n)` callback. This callback will run quickly for small `n` and more slowly for large `n`.

```
app.get('/countToN', (req, res) => {
  let n = req.query.n;
  // n iterations before giving someone else a turn
  for (let i = 0; i < n; i++) {
    console.log(`Iter ${i}`);
  }

  res.sendStatus(200);
});
```

Example 3: An `O(n^2)` callback. This callback will still run quickly for small `n`, but for large `n` it will run much more slowly than the previous `O(n)` example.

```
app.get('/countToN2', (req, res) => {
  let n = req.query.n;

  // n^2 iterations before giving someone else a turn
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      console.log(`Iter ${i}.${j}`);
    }
  }

  res.sendStatus(200);
});
```

## How careful should you be?

Node.js uses the Google V8 engine for JavaScript, which is quite fast for many common operations. Exceptions to this rule are regexps and JSON operations, discussed below.

However, for complex tasks you should consider bounding the input and rejecting inputs that are too long. That way, even if your callback has large complexity, by bounding the input you ensure the callback cannot take more than the worst-case time on the longest acceptable input. You can then evaluate the worst-case cost of this callback and determine whether its running time is acceptable in your context.

## Blocking the Event Loop: REDOS

One common way to block the Event Loop disastrously is by using a "vulnerable" regular expression.

#### Avoiding vulnerable regular expressions

A regular expression (regexp) matches an input string against a pattern. We usually think of a regexp match as requiring a single pass through the input string `O(2^n)` time. An exponential number of trips means that if the engine requires `x` trips to determine a match, it will need `2*x` trips if we add only one more character to the input string. Since the number of trips is linearly related to the time required, the effect of this evaluation will be to block the Event Loop.

A *vulnerable regular expression* is one on which your regular expression engine might take exponential time, exposing you to REDOS on "evil input". Whether or not your regular expression pattern is vulnerable (i.e. the regexp engine might take exponential time on it) is actually a difficult question to answer, and varies depending on whether you're using Perl, Python, Ruby, Java, JavaScript, etc., but here are some rules of thumb that apply across all of these languages:

1. Avoid nested quantifiers like `(a+)*`. V8's regexp engine can handle some of these quickly, but others are vulnerable.
2. Avoid OR's with overlapping clauses, like `(a|a)*`. Again, these are sometimes-fast.
3. Avoid using backreferences, like `(a.*) \1`. No regexp engine can guarantee evaluating these in linear time.
4. If you're doing a simple string match, use `indexOf` or the local equivalent. It will be cheaper and will never take more than `O(n)`.

If you aren't sure whether your regular expression is vulnerable, remember that Node.js generally doesn't have trouble reporting a *match* even for a vulnerable regexp and a long input string. The exponential behavior is triggered when there is a mismatch but Node.js can't be certain until it tries many paths through the input string.

#### A REDOS example

Here is an example vulnerable regexp exposing its server to REDOS:

```
app.get('/redos-me', (req, res) => {
  let filePath = req.query.filePath;

  // REDOS
  if (filePath.match(/(\/.+)+$/)) {
    console.log('valid path');
  } else {
    console.log('invalid path');
  }

  res.sendStatus(200);
});
```

The vulnerable regexp in this example is a (bad!) way to check for a valid path on Linux. It matches strings that are a sequence of "/"-delimited names, like "/a/b/c". It is dangerous because it violates rule 1: it has a doubly-nested quantifier.

If a client queries with filePath `///.../\n` (100 /'s followed by a newline character that the regexp's "." won't match), then the Event Loop will take effectively forever, blocking the Event Loop. This client's REDOS attack causes all other clients not to get a turn until the regexp match finishes.

For this reason, you should be leery of using complex regular expressions to validate user input.

**Anti-REDOS Resources**

There are some tools to check your regexps for safety, like

- safe-regex
- rxxr2.

However, neither of these will catch all vulnerable regexps.

Another approach is to use a different regexp engine. You could use the node-re2 module, which uses Google's blazing-fast RE2 regexp engine. But be warned, RE2 is not 100% compatible with V8's regexps, so check for regressions if you swap in the node-re2 module to handle your regexps. And particularly complicated regexps are not supported by node-re2.

If you're trying to match something "obvious", like a URL or a file path, find an example in a regexp library or use an npm module, e.g. ip-regex.

### Blocking the Event Loop: Node.js core modules

Several Node.js core modules have synchronous expensive APIs, including:

- Encryption
- Compression
- File system
- Child process

These APIs are expensive, because they involve significant computation (encryption, compression), require I/O (file I/O), or potentially both (child process). These APIs are intended for scripting convenience, but are not intended for use in the server context. If you execute them on the Event Loop, they will take far longer to complete than a typical JavaScript instruction, blocking the Event Loop.

In a server, *you should not use the following synchronous APIs from these modules*:

- Encryption:
  - `crypto.randomBytes` (synchronous version)
  - `crypto.randomFillSync`
  - `crypto.pbkdf2Sync`
  - You should also be careful about providing large input to the encryption and decryption routines.
- Compression:
  - `zlib.inflateSync`
  - `zlib.deflateSync`
- File system:
  - Do not use the synchronous file system APIs. For example, if the file you access is in a distributed file system like NFS, access times can vary widely.
- Child process:
  - `child_process.spawnSync`
  - `child_process.execSync`
  - `child_process.execFileSync`

This list is reasonably complete as of Node.js v9.

### Blocking the Event Loop: JSON DOS

`JSON.parse` and `JSON.stringify` are other potentially expensive operations. While these are `O(n)` in the length of the input, for large `n` they can take surprisingly long.

If your server manipulates JSON objects, particularly those from a client, you should be cautious about the size of the objects or strings you work with on the Event Loop.

Example: JSON blocking. We create an object `obj` of size 2^21 and `JSON.stringify` it, run `indexOf` on the string, and then JSON.parse it. The `JSON.stringify`'d string is 50MB. It takes 0.7 seconds to stringify the object, 0.03 seconds to indexOf on the 50MB string, and 1.3 seconds to parse the string.

```
let obj = { a: 1 };
let niter = 20;

let before, str, pos, res, took;

for (let i = 0; i < niter; i++) {
  obj = { obj1: obj, obj2: obj }; // Doubles in size each iter
}

before = process.hrtime();
str = JSON.stringify(obj);
took = process.hrtime(before);
console.log('JSON.stringify took ' + took);

before = process.hrtime();
pos = str.indexOf('nomatch');
took = process.hrtime(before);
console.log('Pure indexof took ' + took);

before = process.hrtime();
res = JSON.parse(str);
took = process.hrtime(before);
console.log('JSON.parse took ' + took);
```

There are npm modules that offer asynchronous JSON APIs. See for example:

- JSONStream, which has stream APIs.
- Big-Friendly JSON, which has stream APIs as well as asynchronous versions of the standard JSON APIs using the partitioning-on-the-Event-Loop paradigm outlined below.

## Complex calculations without blocking the Event Loop

Suppose you want to do complex calculations in JavaScript without blocking the Event Loop. You have two options: partitioning or offloading.

### Partitioning

You could *partition* your calculations so that each runs on the Event Loop but regularly yields (gives turns to) other pending events. In JavaScript it's easy to save the state of an ongoing task in a closure, as shown in example 2 below.

For a simple example, suppose you want to compute the average of the numbers `1` to `n`.

Example 1: Un-partitioned average, costs `O(n)`

```
for (let i = 0; i < n; i++) sum += i;
let avg = sum / n;
console.log('avg: ' + avg);
```

Example 2: Partitioned average, each of the `n` asynchronous steps costs `O(1)`.

```
function asyncAvg(n, avgCB) {
  // Save ongoing sum in JS closure.
  let sum = 0;
  function help(i, cb) {
    sum += i;
    if (i == n) {
      cb(sum);
      return;
    }
```

```
      // "Asynchronous recursion".
      // Schedule next operation asynchronously.
      setImmediate(help.bind(null, i + 1, cb));
    }

    // Start the helper, with CB to call avgCB.
    help(1, function (sum) {
      let avg = sum / n;
      avgCB(avg);
    });
  }

  asyncAvg(n, function (avg) {
    console.log('avg of 1-n: ' + avg);
  });
```

You can apply this principle to array iterations and so forth.

### Offloading

If you need to do something more complex, partitioning is not a good option. This is because partitioning uses only the Event Loop, and you won't benefit from multiple cores almost certainly available on your machine. *Remember, the Event Loop should orchestrate client requests, not fulfill them itself.* For a complicated task, move the work off of the Event Loop onto a Worker Pool.

##### How to offload

You have two options for a destination Worker Pool to which to offload work.

1. You can use the built-in Node.js Worker Pool by developing a C++ addon. On older versions of Node, build your C++ addon using NAN, and on newer versions use N-API. node-webworker-threads offers a JavaScript-only way to access the Node.js Worker Pool.
2. You can create and manage your own Worker Pool dedicated to computation rather than the Node.js I/O-themed Worker Pool. The most straightforward ways to do this is using Child Process or Cluster.

You should *not* simply create a Child Process for every client. You can receive client requests more quickly than you can create and manage children, and your server might become a fork bomb.

##### Downside of offloading

The downside of the offloading approach is that it incurs overhead in the form of *communication costs*. Only the Event Loop is allowed to see the "namespace" (JavaScript state) of your application. From a Worker, you cannot manipulate a JavaScript object in the Event Loop's namespace. Instead, you have to serialize and deserialize any objects you wish to share. Then the Worker can operate on its own copy of these object(s) and return the modified object (or a "patch") to the Event Loop.

For serialization concerns, see the section on JSON DOS.

##### Some suggestions for offloading

You may wish to distinguish between CPU-intensive and I/O-intensive tasks because they have markedly different characteristics.

A CPU-intensive task only makes progress when its Worker is scheduled, and the Worker must be scheduled onto one of your machine's logical cores. If you have 4 logical cores and 5 Workers, one of these Workers cannot make progress. As a result, you are paying overhead (memory and scheduling costs) for this Worker and getting no return for it.

I/O-intensive tasks involve querying an external service provider (DNS, file system, etc.) and waiting for its response. While a Worker with an I/O-intensive task is waiting for its response, it has nothing else to do and can be de-scheduled by the operating system, giving another Worker a chance to submit their request. Thus, *I/O-intensive tasks will be making progress even while the associated thread is not running*. External service providers like databases and file systems have been highly optimized to handle many pending requests concurrently. For example, a file system will examine a large set of pending write and read requests to merge conflicting updates and to retrieve files in an optimal order.

If you rely on only one Worker Pool, e.g. the Node.js Worker Pool, then the differing characteristics of CPU-bound and I/O-bound work may harm your application's performance.

For this reason, you might wish to maintain a separate Computation Worker Pool.

### Offloading: conclusions

For simple tasks, like iterating over the elements of an arbitrarily long array, partitioning might be a good option. If your computation is more complex, offloading is a better approach: the communication costs, i.e. the overhead of passing serialized objects between the Event Loop and the Worker Pool, are offset by the benefit of using multiple cores.

However, if your server relies heavily on complex calculations, you should think about whether Node.js is really a good fit. Node.js excels for I/O-bound work, but for expensive computation it might not be the best option.

If you take the offloading approach, see the section on not blocking the Worker Pool.

## Don't block the Worker Pool

Node.js has a Worker Pool composed of `k` Workers. If you are using the Offloading paradigm discussed above, you might have a separate Computational Worker Pool, to which the same principles apply. In either case, let us assume that `k` is much smaller than the number of clients you might be handling concurrently. This is in keeping with the "one thread for many clients" philosophy of Node.js, the secret to its scalability.

As discussed above, each Worker completes its current Task before proceeding to the next one on the Worker Pool queue.

Now, there will be variation in the cost of the Tasks required to handle your clients' requests. Some Tasks can be completed quickly (e.g. reading short or cached files, or producing a small number of random bytes), and others will take longer (e.g reading larger or uncached files, or generating more random bytes). Your goal should be to *minimize the variation in Task times*, and you should use *Task partitioning* to accomplish this.

### Minimizing the variation in Task times

If a Worker's current Task is much more expensive than other Tasks, then it will be unavailable to work on other pending Tasks. In other words, *each relatively long Task effectively decreases the size of the Worker Pool by one until it is completed*. This is undesirable because, up to a point, the more Workers in the Worker Pool, the greater the Worker Pool throughput (tasks/second) and thus the greater the server throughput (client requests/second). One client with a relatively expensive Task will decrease the throughput of the Worker Pool, in turn decreasing the throughput of the server.

To avoid this, you should try to minimize variation in the length of Tasks you submit to the Worker Pool. While it is appropriate to treat the external systems accessed by your I/O requests (DB, FS, etc.) as black boxes, you should be aware of the relative cost of these I/O requests, and should avoid submitting requests you can expect to be particularly long.

Two examples should illustrate the possible variation in task times.

#### Variation example: Long-running file system reads

Suppose your server must read files in order to handle some client requests. After consulting the Node.js File system APIs, you opted to use `fs.readFile()` for simplicity. However, `fs.readFile()` is (currently) not partitioned: it submits a single `fs.read()` Task spanning the entire file. If you read shorter files for some users and longer files for others, `fs.readFile()` may introduce significant variation in Task lengths, to the detriment of Worker Pool throughput.

For a worst-case scenario, suppose an attacker can convince your server to read an *arbitrary* file (this is a directory traversal vulnerability). If your server is running Linux, the attacker can name an extremely slow file: `/dev/random`. For all practical purposes, `/dev/random` is infinitely slow, and every Worker asked to read from `/dev/random` will never finish that Task. An attacker then submits `k` requests, one for each Worker, and no other client requests that use the Worker Pool will make progress.

#### Variation example: Long-running crypto operations

Suppose your server generates cryptographically secure random bytes using `crypto.randomBytes()`. `crypto.randomBytes()` is not partitioned: it creates a single `randomBytes()` Task to generate as many bytes as you requested. If you create fewer bytes for some users and more bytes for others, `crypto.randomBytes()` is another source of variation in Task lengths.

### Task partitioning

Tasks with variable time costs can harm the throughput of the Worker Pool. To minimize variation in Task times, as far as possible you should *partition* each Task into comparable-cost sub-Tasks. When each sub-Task completes it should submit the next sub-Task, and when the final sub-Task completes it should notify the submitter.

To continue the `fs.readFile()` example, you should instead use `fs.read()` (manual partitioning) or `ReadStream` (automatically partitioned).

The same principle applies to CPU-bound tasks; the `asyncAvg` example might be inappropriate for the Event Loop, but it is well suited to the Worker Pool.

When you partition a Task into sub-Tasks, shorter Tasks expand into a small number of sub-Tasks, and longer Tasks expand into a larger number of sub-Tasks. Between each sub-Task of a longer Task, the Worker to which it was assigned can work on a sub-Task from another, shorter, Task, thus improving the overall Task throughput of the Worker Pool.

Note that the number of sub-Tasks completed is not a useful metric for the throughput of the Worker Pool. Instead, concern yourself with the number of *Tasks* completed.

### Avoiding Task partitioning

Recall that the purpose of Task partitioning is to minimize the variation in Task times. If you can distinguish between shorter Tasks and longer Tasks (e.g. summing an array vs. sorting an array), you could create one Worker Pool for each class of Task. Routing shorter Tasks and longer Tasks to separate Worker Pools is another way to minimize Task time variation.

In favor of this approach, partitioning Tasks incurs overhead (the costs of creating a Worker Pool Task representation and of manipulating the Worker Pool queue), and avoiding partitioning saves you the costs of additional trips to the Worker Pool. It also keeps you from making mistakes in partitioning your Tasks.

The downside of this approach is that Workers in all of these Worker Pools will incur space and time overheads and will compete with each other for CPU time. Remember that each CPU-bound Task makes progress only while it is scheduled. As a result, you should only consider this approach after careful analysis.

### Worker Pool: conclusions

Whether you use only the Node.js Worker Pool or maintain separate Worker Pool(s), you should optimize the Task throughput of your Pool(s).

To do this, minimize the variation in Task times by using Task partitioning.

## The risks of npm modules

While the Node.js core modules offer building blocks for a wide variety of applications, sometimes something more is needed. Node.js developers benefit tremendously from the npm ecosystem, with hundreds of thousands of modules offering functionality to accelerate your development process.

Remember, however, that the majority of these modules are written by third-party developers and are generally released with only best-effort guarantees. A developer using an npm module should be concerned about two things, though the latter is frequently forgotten.

1. Does it honor its APIs?
2. Might its APIs block the Event Loop or a Worker? Many modules make no effort to indicate the cost of their APIs, to the detriment of the community.

For simple APIs you can estimate the cost of the APIs; the cost of string manipulation isn't hard to fathom. But in many cases it's unclear how much an API might cost.

*If you are calling an API that might do something expensive, double-check the cost. Ask the developers to document it, or examine the source code yourself (and submit a PR documenting the cost).*

Remember, even if the API is asynchronous, you don't know how much time it might spend on a Worker or on the Event Loop in each of its partitions. For example, suppose in the `asyncAvg` example given above, each call to the helper function summed *half* of the numbers rather than one of them. Then this function would still be asynchronous, but the cost of each partition would be `O(n)`, not `O(1)`, making it much less safe to use for arbitrary values of `n`.

## Conclusion

Node.js has two types of threads: one Event Loop and `k` Workers. The Event Loop is responsible for JavaScript callbacks and non-blocking I/O, and a Worker executes tasks corresponding to C++ code that completes an asynchronous request, including blocking I/O and CPU-intensive work. Both types of threads work on no more than one activity at a time. If any callback or task takes a long time, the thread running it becomes *blocked*. If your application makes blocking callbacks or tasks, this can lead to degraded throughput (clients/second) at best, and complete denial of service at worst.

To write a high-throughput, more DoS-proof web server, you must ensure that on benign and on malicious input, neither your Event Loop nor your Workers will block.

# Manipulating Files

## nodejs file stats

# Node.js file stats

Every file comes with a set of details that we can inspect using Node.js. In particular, using the `stat()` method provided by the `fs` module.

You call it passing a file path, and once Node.js gets the file details it will call the callback function you pass, with 2 parameters: an error message, and the file stats:

```
const fs = require('node:fs');

fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err);
  }
  // we have access to the file stats in `stats`
});
```

```
import fs from 'node:fs';

fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err);
  }
  // we have access to the file stats in `stats`
});
```

Node.js also provides a sync method, which blocks the thread until the file stats are ready:

```
const fs = require('node:fs');

try {
  const stats = fs.statSync('/Users/joe/test.txt');
} catch (err) {
  console.error(err);
}
```

```
import fs from 'node:fs';

try {
  const stats = fs.statSync('/Users/joe/test.txt');
} catch (err) {
  console.error(err);
}
```

The file information is included in the stats variable. What kind of information can we extract using the stats?

**A lot, including:**

- if the file is a directory or a file, using `stats.isFile()` and `stats.isDirectory()`
- if the file is a symbolic link using `stats.isSymbolicLink()`
- the file size in bytes using `stats.size`.

There are other advanced methods, but the bulk of what you'll use in your day-to-day programming is this.

```
const fs = require('node:fs');

fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err);
    return;
  }

  stats.isFile(); // true
  stats.isDirectory(); // false
  stats.isSymbolicLink(); // false
  stats.size; // 1024000 //= 1MB
});
```

```
import fs from 'node:fs';

fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err);
    return;
  }

  stats.isFile(); // true
  stats.isDirectory(); // false
  stats.isSymbolicLink(); // false
  stats.size; // 1024000 //= 1MB
});
```

You can also use promise-based `fsPromises.stat()` method offered by the `fs/promises` module if you like:

```
const fs = require('node:fs/promises');

async function example() {
  try {
    const stats = await fs.stat('/Users/joe/test.txt');
    stats.isFile(); // true
    stats.isDirectory(); // false
    stats.isSymbolicLink(); // false
    stats.size; // 1024000 //= 1MB
  } catch (err) {
    console.log(err);
  }
}
example();
```

```
import fs from 'node:fs/promises';

try {
  const stats = await fs.stat('/Users/joe/test.txt');
  stats.isFile(); // true
  stats.isDirectory(); // false
  stats.isSymbolicLink(); // false
  stats.size; // 1024000 //= 1MB
} catch (err) {
  console.log(err);
}
```

You can read more about the `fs` module in the official documentation.

# Node.js File Paths

Every file in the system has a path. On Linux and macOS, a path might look like: `/users/joe/file.txt` while Windows computers are different, and have a structure such as: `C:\users\joe\file.txt`

You need to pay attention when using paths in your applications, as this difference must be taken into account.

You include this module in your files using `const path = require('node:path');` and you can start using its methods.

## Getting information out of a path

Given a path, you can extract information out of it using those methods:

- `dirname`: gets the parent folder of a file
- `basename`: gets the filename part
- `extname`: gets the file extension

### Example

```
const path = require('node:path');

const notes = '/users/joe/notes.txt';

path.dirname(notes); // /users/joe
path.basename(notes); // notes.txt
path.extname(notes); // .txt
```

```
import path from 'node:path';

const notes = '/users/joe/notes.txt';

path.dirname(notes); // /users/joe
path.basename(notes); // notes.txt
path.extname(notes); // .txt
```

You can get the file name without the extension by specifying a second argument to `basename`:

```
path.basename(notes, path.extname(notes)); // notes
```

## Working with paths

You can join two or more parts of a path by using `path.join()`:

```
const name = 'joe';
path.join('/', 'users', name, 'notes.txt'); // '/users/joe/notes.txt'
```

You can get the absolute path calculation of a relative path using `path.resolve()`:

```
path.resolve('joe.txt'); // '/Users/joe/joe.txt' if run from my home folder
```

In this case Node.js will simply append `/joe.txt` to the current working directory. If you specify a second parameter folder, `resolve` will use the first as a base for the second:

```
path.resolve('tmp', 'joe.txt'); // '/Users/joe/tmp/joe.txt' if run from my home folder
```

If the first parameter starts with a slash, that means it's an absolute path:

```
path.resolve('/etc', 'joe.txt'); // '/etc/joe.txt'
```

`path.normalize()` is another useful function, that will try and calculate the actual path, when it contains relative specifiers like `.` or `..`, or double slashes:

```
path.normalize('/users/joe/..//test.txt'); // '/users/test.txt'
```

**Neither resolve nor normalize will check if the path exists**. They just calculate a path based on the information they got.

## working with file descriptors in nodejs

# Working with file descriptors in Node.js

Before you're able to interact with a file that sits in your filesystem, you must get a file descriptor.

A file descriptor is a reference to an open file, a number (fd) returned by opening the file using the `open()` method offered by the `fs` module. This number (`fd`) uniquely identifies an open file in operating system:

```
const fs = require('node:fs');

fs.open('/Users/joe/test.txt', 'r', (err, fd) => {
  // fd is our file descriptor
});
```

```
import fs from 'node:fs';

fs.open('/Users/joe/test.txt', 'r', (err, fd) => {
  // fd is our file descriptor
});
```

Notice the `r` we used as the second parameter to the `fs.open()` call.

That flag means we open the file for reading.

**Other flags you'll commonly use are:**

| Flag | Description | File gets created if it doesn't exist | | ---- | | r+ | This flag opens the file for reading and writing | ❌ | | w+ | This flag opens the file for reading and writing and it also positions the stream at the beginning of the file | ✅ | | a | This flag opens the file for writing and it also positions the stream at the end of the file | ✅ | | a+ | This flag opens the file for reading and writing and it also positions the stream at the end of the file | ✅ |

You can also open the file by using the `fs.openSync` method, which returns the file descriptor, instead of providing it in a callback:

```
const fs = require('node:fs');

try {
  const fd = fs.openSync('/Users/joe/test.txt', 'r');
} catch (err) {
  console.error(err);
}
```

```
import fs from 'node:fs';

try {
  const fd = fs.openSync('/Users/joe/test.txt', 'r');
} catch (err) {
  console.error(err);
}
```

Once you get the file descriptor, in whatever way you choose, you can perform all the operations that require it, like calling `fs.close()` and many other operations that interact with the filesystem.

You can also open the file by using the promise-based `fsPromises.open` method offered by the `fs/promises` module.

The `fs/promises` module is available starting only from Node.js v14. Before v14, after v10, you can use `require('fs').promises` instead. Before v10, after v8, you can use `util.promisify` to convert `fs` methods into promise-based methods.

```
const fs = require('node:fs/promises');
// Or const fs = require('fs').promises before v14.
async function example() {
  let filehandle;
  try {
    filehandle = await fs.open('/Users/joe/test.txt', 'r');
    console.log(filehandle.fd);
    console.log(await filehandle.readFile({ encoding: 'utf8' }));
  } finally {
    if (filehandle) await filehandle.close();
  }
}
example();
```

```
import fs from 'node:fs/promises';
// Or const fs = require('fs').promises before v14.
let filehandle;
try {
  filehandle = await fs.open('/Users/joe/test.txt', 'r');
  console.log(filehandle.fd);
  console.log(await filehandle.readFile({ encoding: 'utf8' }));
} finally {
  if (filehandle) await filehandle.close();
}
```

Here is an example of `util.promisify`:

```
const fs = require('node:fs');
const util = require('node:util');

async function example() {
  const open = util.promisify(fs.open);
  const fd = await open('/Users/joe/test.txt', 'r');
}
example();
```

```
import fs from 'node:fs';
import util from 'node:util';

async function example() {
  const open = util.promisify(fs.open);
  const fd = await open('/Users/joe/test.txt', 'r');
}
example();
```

To see more details about the `fs/promises` module, please check fs/promises API.

# Reading files with Node.js

The simplest way to read a file in Node.js is to use the `fs.readFile()` method, passing it the file path, encoding and a callback function that will be called with the file data (and the error):

```
const fs = require('node:fs');

fs.readFile('/Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

```
import fs from 'node:fs';

fs.readFile('/Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

Alternatively, you can use the synchronous version `fs.readFileSync()`:

```
const fs = require('node:fs');

try {
  const data = fs.readFileSync('/Users/joe/test.txt', 'utf8');
  console.log(data);
} catch (err) {
  console.error(err);
}
```

```
import fs from 'node:fs';

try {
  const data = fs.readFileSync('/Users/joe/test.txt', 'utf8');
  console.log(data);
} catch (err) {
  console.error(err);
}
```

You can also use the promise-based `fsPromises.readFile()` method offered by the `fs/promises` module:

```
const fs = require('node:fs/promises');

async function example() {
  try {
    const data = await fs.readFile('/Users/joe/test.txt', { encoding: 'utf8' });
    console.log(data);
  } catch (err) {
    console.log(err);
  }
}
example();
```

```
import fs from 'node:fs/promises';

try {
  const data = await fs.readFile('/Users/joe/test.txt', { encoding: 'utf8' });
  console.log(data);
} catch (err) {
  console.log(err);
}
```

All three of `fs.readFile()`, `fs.readFileSync()` and `fsPromises.readFile()` read the full content of the file in memory before returning the data.

This means that big files are going to have a major impact on your memory consumption and speed of execution of the program.

In this case, a better option is to read the file content using streams.

---

# writing files with nodejs

# Writing files with Node.js

## Writing a file

The easiest way to write to files in Node.js is to use the `fs.writeFile()` API.

```
const fs = require('node:fs');

const content = 'Some content!';

fs.writeFile('/Users/joe/test.txt', content, err => {
  if (err) {
    console.error(err);
  } else {
    // file written successfully
  }
});
```

### Writing a file synchronously

Alternatively, you can use the synchronous version `fs.writeFileSync()`:

```
const fs = require('node:fs');

const content = 'Some content!';

try {
  fs.writeFileSync('/Users/joe/test.txt', content);
  // file written successfully
} catch (err) {
  console.error(err);
}
```

You can also use the promise-based `fsPromises.writeFile()` method offered by the `fs/promises` module:

```
const fs = require('node:fs/promises');

async function example() {
  try {
    const content = 'Some content!';
    await fs.writeFile('/Users/joe/test.txt', content);
  } catch (err) {
    console.log(err);
  }
}

example();
```

By default, this API will **replace the contents of the file** if it does already exist.

**You can modify the default by specifying a flag:**

```
fs.writeFile('/Users/joe/test.txt', content, { flag: 'a+' }, err => {});
```

**The flags you'll likely use are**

| Flag | Description | File gets created if it doesn't exist | | --: | | `r+` | This flag opens the file for **reading** and **writing** | ❌ | | `w+` | This flag opens the file for **reading** and **writing** and it also positions the stream at the **beginning** of the file | ✅ | | `a` | This flag opens the file for **writing** and it also positions the stream at the **end** of the file | ✅ | | `a+` | This flag opens the file for **reading** and **writing** and it also positions the stream at the **end** of the file | ✅ |

- You can find more information about the flags in the fs documentation.

## Appending content to a file

Appending to files is handy when you don't want to overwrite a file with new content, but rather add to it.

### Examples

A handy method to append content to the end of a file is `fs.appendFile()` (and its `fs.appendFileSync()` counterpart):

```
const fs = require('node:fs');

const content = 'Some content!';

fs.appendFile('file.log', content, err => {
  if (err) {
    console.error(err);
  } else {
    // done!
  }
});
```

#### Example with Promises

Here is a `fsPromises.appendFile()` example:

```
const fs = require('node:fs/promises');

async function example() {
  try {
    const content = 'Some content!';
    await fs.appendFile('/Users/joe/test.txt', content);
  } catch (err) {
    console.log(err);
  }
}

example();
```

# Working with folders in Node.js

The Node.js `fs` core module provides many handy methods you can use to work with folders.

## Check if a folder exists

Use `fs.access()` (and its promise-based `fsPromises.access()` counterpart) to check if the folder exists and Node.js can access it with its permissions.

## Create a new folder

Use `fs.mkdir()` or `fs.mkdirSync()` or `fsPromises.mkdir()` to create a new folder.

```
const fs = require('node:fs');

const folderName = '/Users/joe/test';

try {
  if (!fs.existsSync(folderName)) {
    fs.mkdirSync(folderName);
  }
} catch (err) {
  console.error(err);
}
```

```
import fs from 'node:fs';

const folderName = '/Users/joe/test';

try {
  if (!fs.existsSync(folderName)) {
    fs.mkdirSync(folderName);
  }
} catch (err) {
  console.error(err);
}
```

## Read the content of a directory

Use `fs.readdir()` or `fs.readdirSync()` or `fsPromises.readdir()` to read the contents of a directory.

This piece of code reads the content of a folder, both files and subfolders, and returns their relative path:

```
const fs = require('node:fs');

const folderPath = '/Users/joe';

fs.readdirSync(folderPath);
```

```
import fs from 'node:fs';

const folderPath = '/Users/joe';

fs.readdirSync(folderPath);
```

You can get the full path:

```
fs.readdirSync(folderPath).map(fileName => {
  return path.join(folderPath, fileName);
});
```

You can also filter the results to only return the files, and exclude the folders:

```
const fs = require('node:fs');

const isFile = fileName => {
  return fs.lstatSync(fileName).isFile();
};

fs.readdirSync(folderPath)
  .map(fileName => {
    return path.join(folderPath, fileName);
  })
  .filter(isFile);
```

```
import fs from 'node:fs';

const isFile = fileName => {
  return fs.lstatSync(fileName).isFile();
};

fs.readdirSync(folderPath)
  .map(fileName => {
    return path.join(folderPath, fileName);
  })
  .filter(isFile);
```

## Rename a folder

Use `fs.rename()` or `fs.renameSync()` or `fsPromises.rename()` to rename folder. The first parameter is the current path, the second the new path:

```
const fs = require('node:fs');

fs.rename('/Users/joe', '/Users/roger', err => {
  if (err) {
    console.error(err);
  }
  // done
});
```

```
import fs from 'node:fs';

fs.rename('/Users/joe', '/Users/roger', err => {
  if (err) {
    console.error(err);
  }
  // done
});
```

`fs.renameSync()` is the synchronous version:

```
const fs = require('node:fs');

try {
  fs.renameSync('/Users/joe', '/Users/roger');
} catch (err) {
  console.error(err);
}
```

```
import fs from 'node:fs';

try {
  fs.renameSync('/Users/joe', '/Users/roger');
} catch (err) {
  console.error(err);
}
```

`fsPromises.rename()` is the promise-based version:

```
const fs = require('node:fs/promises');
```

```
async function example() {
  try {
    await fs.rename('/Users/joe', '/Users/roger');
  } catch (err) {
    console.log(err);
  }
}
example();
```

```
import fs from 'node:fs/promises';

try {
  await fs.rename('/Users/joe', '/Users/roger');
} catch (err) {
  console.log(err);
}
```

## Remove a folder

Use `fs.rmdir()` or `fs.rmdirSync()` or `fsPromises.rmdir()` to remove a folder.

```
const fs = require('node:fs');

fs.rmdir(dir, err => {
  if (err) {
    throw err;
  }

  console.log(`${dir} is deleted!`);
});
```

```
import fs from 'node:fs';

fs.rmdir(dir, err => {
  if (err) {
    throw err;
  }

  console.log(`${dir} is deleted!`);
});
```

To remove a folder that has contents use `fs.rm()` with the option `{ recursive: true }` to recursively remove the contents.

`{ recursive: true, force: true }` makes it so that exceptions will be ignored if the folder does not exist.

```
const fs = require('node:fs');

fs.rm(dir, { recursive: true, force: true }, err => {
  if (err) {
    throw err;
  }

  console.log(`${dir} is deleted!`);
});
```

```
import fs from 'node:fs';

fs.rm(dir, { recursive: true, force: true }, err => {
  if (err) {
    throw err;
  }

  console.log(`${dir} is deleted!`);
});
```

# How to Work with Different Filesystems

Node.js exposes many features of the filesystem. But not all filesystems are alike. The following are suggested best practices to keep your code simple and safe when working with different filesystems.

## Filesystem Behavior

Before you can work with a filesystem, you need to know how it behaves. Different filesystems behave differently and have more or less features than others: case sensitivity, case insensitivity, case preservation, Unicode form preservation, timestamp resolution, extended attributes, inodes, Unix permissions, alternate data streams etc.

Be wary of inferring filesystem behavior from `process.platform`. For example, do not assume that because your program is running on Darwin that you are therefore working on a case-insensitive filesystem (HFS+), as the user may be using a case-sensitive filesystem (HFSX). Similarly, do not assume that because your program is running on Linux that you are therefore working on a filesystem which supports Unix permissions and inodes, as you may be on a particular external drive, USB or network drive which does not.

The operating system may not make it easy to infer filesystem behavior, but all is not lost. Instead of keeping a list of every known filesystem and behavior (which is always going to be incomplete), you can probe the filesystem to see how it actually behaves. The presence or absence of certain features which are easy to probe, are often enough to infer the behavior of other features which are more difficult to probe.

Remember that some users may have different filesystems mounted at various paths in the working tree.

## Avoid a Lowest Common Denominator Approach

You might be tempted to make your program act like a lowest common denominator filesystem, by normalizing all filenames to uppercase, normalizing all filenames to NFC Unicode form, and normalizing all file timestamps to say 1-second resolution. This would be the lowest common denominator approach.

Do not do this. You would only be able to interact safely with a filesystem which has the exact same lowest common denominator characteristics in every respect. You would be unable to work with more advanced filesystems in the way that users expect, and you would run into filename or timestamp collisions. You would most certainly lose and corrupt user data through a series of complicated dependent events, and you would create bugs that would be difficult if not impossible to solve.

What happens when you later need to support a filesystem that only has 2-second or 24-hour timestamp resolution? What happens when the Unicode standard advances to include a slightly different normalization algorithm (as has happened in the past)?

A lowest common denominator approach would tend to try to create a portable program by using only "portable" system calls. This leads to programs that are leaky and not in fact portable.

## Adopt a Superset Approach

Make the best use of each platform you support by adopting a superset approach. For example, a portable backup program should sync btimes (the created time of a file or folder) correctly between Windows systems, and should not destroy or alter btimes, even though btimes are not supported on Linux systems. The same portable backup program should sync Unix permissions correctly between Linux systems, and should not destroy or alter Unix permissions, even though Unix permissions are not supported on Windows systems.

Handle different filesystems by making your program act like a more advanced filesystem. Support a superset of all possible features: case-sensitivity, case-preservation, Unicode form sensitivity, Unicode form preservation, Unix permissions, high-resolution nanosecond timestamps, extended attributes etc.

Once you have case-preservation in your program, you can always implement case-insensitivity if you need to interact with a case-insensitive filesystem. But if you forego case-preservation in your program, you cannot interact safely with a case-preserving filesystem. The same is true for Unicode form preservation and timestamp resolution preservation.

If a filesystem provides you with a filename in a mix of lowercase and uppercase, then keep the filename in the exact case given. If a filesystem provides you with a filename in mixed Unicode form or NFC or NFD (or NFKC or NFKD), then keep the filename in the exact byte sequence given. If a filesystem provides you with a millisecond timestamp, then keep the timestamp in millisecond resolution.

When you work with a lesser filesystem, you can always downsample appropriately, with comparison functions as required by the behavior of the filesystem on which your program is running. If you know that the filesystem does not support Unix permissions, then you should not expect to read the same Unix permissions you write. If you know that the filesystem does not preserve case, then you should be prepared to see `ABC` in a directory listing when your program creates `abc`. But if you know that the filesystem does preserve case, then you should consider `ABC` to be a different filename to `abc`, when detecting file renames or if the filesystem is case-sensitive.

## Case Preservation

You may create a directory called `test/abc` and be surprised to see sometimes that `fs.readdir('test')` returns `['ABC']`. This is not a bug in Node. Node returns the filename as the filesystem stores it, and not all filesystems support case-preservation. Some filesystems convert all filenames to uppercase (or lowercase).

## Unicode Form Preservation

*Case preservation and Unicode form preservation are similar concepts. To understand why Unicode form should be preserved , make sure that you first understand why case should be preserved. Unicode form preservation is just as simple when understood correctly.*

Unicode can encode the same characters using several different byte sequences. Several strings may look the same, but have different byte sequences. When working with UTF-8 strings, be careful that your expectations are in line with how Unicode works. Just as you would not expect all UTF-8 characters to encode to a single byte, you should not expect several UTF-8 strings that look the same to the human eye to have the same byte representation. This may be an expectation that you can have of ASCII, but not of UTF-8.

You may create a directory called `test/café` (NFC Unicode form with byte sequence `<63 61 66 c3 a9>` and `string.length === 5`) and be surprised to see sometimes that `fs.readdir('test')` returns `['café']` (NFD Unicode form with byte sequence `<63 61 66 65 cc 81>` and `string.length === 6`). This is not a bug in Node. Node.js returns the filename as the filesystem stores it, and not all filesystems support Unicode form preservation.

HFS+, for example, will normalize all filenames to a form almost always the same as NFD form. Do not expect HFS+ to behave the same as NTFS or EXT4 and vice-versa. Do not try to change data permanently through normalization as a leaky abstraction to paper over Unicode differences between filesystems. This would create problems without solving any. Rather, preserve Unicode form and use normalization as a comparison function only.

## Unicode Form Insensitivity

Unicode form insensitivity and Unicode form preservation are two different filesystem behaviors often mistaken for each other. Just as case-insensitivity has sometimes been incorrectly implemented by permanently normalizing filenames to uppercase when storing and transmitting filenames, so Unicode form insensitivity has sometimes been incorrectly implemented by permanently normalizing filenames to a certain Unicode form (NFD in the case of HFS+) when storing and transmitting filenames. It is possible and much better to implement Unicode form insensitivity without sacrificing Unicode form preservation, by using Unicode normalization for comparison only.

## Comparing Different Unicode Forms

Node.js provides `string.normalize('NFC' / 'NFD')` which you can use to normalize a UTF-8 string to either NFC or NFD. You should never store the output from this function but only use it as part of a comparison function to test whether two UTF-8 strings would look the same to the user.

You can use `string1.normalize('NFC') === string2.normalize('NFC')` or `string1.normalize('NFD') === string2.normalize('NFD')` as your comparison function. Which form you use does not matter.

Normalization is fast but you may want to use a cache as input to your comparison function to avoid normalizing the same string many times over. If the string is not present in the cache then normalize it and cache it. Be careful not to store or persist the cache, use it only as a cache.

Note that using `normalize()` requires that your version of Node.js include ICU (otherwise `normalize()` will just return the original string). If you download the latest version of Node.js from the website then it will include ICU.

## Timestamp Resolution

You may set the `mtime` (the modified time) of a file to `1444291759414` (millisecond resolution) and be surprised to see sometimes that `fs.stat` returns the new mtime as `1444291759000` (1-second resolution) or `1444291758000` (2-second resolution). This is not a bug in Node. Node.js returns the timestamp as the filesystem stores it, and not all filesystems support nanosecond, millisecond or 1-second timestamp resolution. Some filesystems even have very coarse resolution for the atime timestamp in particular, e.g. 24 hours for some FAT filesystems.

## Do Not Corrupt Filenames and Timestamps Through Normalization

Filenames and timestamps are user data. Just as you would never automatically rewrite user file data to uppercase the data or normalize `CRLF` to `LF` line-endings, so you should never change, interfere or corrupt filenames or timestamps through case / Unicode form / timestamp normalization. Normalization should only ever be used for comparison, never for altering data.

Normalization is effectively a lossy hash code. You can use it to test for certain kinds of equivalence (e.g. do several strings look the same even though they have different byte sequences) but you can never use it as a substitute for the actual data. Your program should pass on filename and timestamp data as is.

Your program can create new data in NFC (or in any combination of Unicode form it prefers) or with a lowercase or uppercase filename, or with a 2-second resolution timestamp, but your program should not corrupt existing user data by imposing case / Unicode form / timestamp normalization. Rather, adopt a superset approach and preserve case, Unicode form and timestamp resolution in your program. That way, you will be able to interact safely with filesystems which do the same.

## Use Normalization Comparison Functions Appropriately

Make sure that you use case / Unicode form / timestamp comparison functions appropriately. Do not use a case-insensitive filename comparison function if you are working on a case-sensitive filesystem. Do not use a Unicode form insensitive comparison function if you are working on a Unicode form sensitive filesystem (e.g. NTFS and most Linux filesystems which preserve both NFC and NFD or mixed Unicode forms). Do not compare timestamps at 2-second resolution if you are working on a nanosecond timestamp resolution filesystem.

## Be Prepared for Slight Differences in Comparison Functions

Be careful that your comparison functions match those of the filesystem (or probe the filesystem if possible to see how it would actually compare). Case-insensitivity for example is more complex than a simple `toLowerCase()` comparison. In fact, `toUpperCase()` is usually better than `toLowerCase()` (since it handles certain foreign language characters differently). But better still would be to probe the filesystem since every filesystem has its own case comparison table baked in.

As an example, Apple's HFS+ normalizes filenames to NFD form but this NFD form is actually an older version of the current NFD form and may sometimes be slightly different from the latest Unicode standard's NFD form. Do not expect HFS+ NFD to be exactly the same as Unicode NFD all the time.

# Command Line

## run nodejs scripts from the command line

# Run Node.js scripts from the command line

The usual way to run a Node.js program is to run the globally available `node` 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
```

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 javascript 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
```

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

## Pass string as argument to `node` instead of file path

To execute a string as argument you can use `-e`, `--eval "script"`. Evaluate the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script.

On Windows, using cmd.exe a single quote will not work correctly because it only recognizes double `"` for quoting. In Powershell or Git bash, both `'` and `"` are usable.

```
node -e "console.log(123)"
```

## Restart the application automatically

As of nodejs V16, there is a built-in option to automatically restart the application when a file changes. This is useful for development purposes. To use this feature, you need to pass the `--watch' flag to nodejs.

```
node --watch app.js
```

So when you change the file, the application will restart automatically. Read the `--watch` flag documentation.

## How to read environment variables from Node.js

The `process` core module of Node.js provides the `env` property which hosts all the environment variables that were set at the moment the process was started.

The below code runs `app.js` and set `USER_ID` and `USER_KEY`.

```
USER_ID=239482 USER_KEY=foobar node app.js
```

That will pass the user `USER_ID` as **239482** and the `USER_KEY` as **foobar**. This is suitable for testing, however for production, you will probably be configuring some bash scripts to export variables.

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

Here is an example that accesses the `USER_ID` and `USER_KEY` environment variables, which we set in above code.

```
process.env.USER_ID; // "239482"
process.env.USER_KEY; // "foobar"
```

In the same way you can access any custom environment variable you set.

Node.js 20 introduced **experimental** support for .env files.

Now, you can use the `--env-file` flag to specify an environment file when running your Node.js application. Here's an example `.env` file and how to access its variables using `process.env`.

```
# .env file
PORT=3000
```

In your js file

```
process.env.PORT; // "3000"
```

Run `app.js` file with environment variables set in `.env` file.

```
node --env-file=.env app.js
```

This command loads all the environment variables from the `.env` file, making them available to the application on `process.env`

Also, you can pass multiple `--env-file` arguments. Subsequent files override pre-existing variables defined in previous files.

```
node --env-file=.env --env-file=.development.env app.js
```

> Note: if the same variable is defined in the environment and in the file, the value from the environment takes precedence.

## how to use the nodejs repl

# How to use the Node.js REPL

The `node` command is the one we use to run our Node.js scripts:

```
node script.js
```

If we run the `node` command without any script to execute or without any arguments, we start a REPL session:

```
node
```

> **Note:** `REPL` stands for Read Evaluate Print Loop, and it is a programming language environment (basically a console window) that takes single expression as user input and returns the result back to the console after execution. The REPL session provides a convenient way to quickly test simple JavaScript code.

If you try it now in your terminal, this is what happens:

```
❯ node
>
```

The command stays in idle mode and waits for us to enter something.

> **Tip:** if you are unsure how to open your terminal, google "How to open terminal on your-operating-system".

The REPL is waiting for us to enter some JavaScript code, to be more precise.

Start simple and enter

```
> console.log('test')
test
undefined
>
```

The first value, `test`, is the output we told the console to print, then we get `undefined` which is the return value of running `console.log()`. Node read this line of code, evaluated it, printed the result, and then went back to waiting for more lines of code. Node will loop through these three steps for every piece of code we execute in the REPL until we exit the session. That is where the REPL got its name.

Node automatically prints the result of any line of JavaScript code without the need to instruct it to do so. For example, type in the following line and press enter:

```
> 5 === '5'
false
>
```

Note the difference in the outputs of the above two lines. The Node REPL printed `undefined` after executing `console.log()`, while on the other hand, it just printed the result of `5 === '5'`. You need to keep in mind that the former is just a statement in JavaScript, and the latter is an expression.

In some cases, the code you want to test might need multiple lines. For example, say you want to define a function that generates a random number, in the REPL session type in the following line and press enter:

```
function generateRandom() {
...
```

The Node REPL is smart enough to determine that you are not done writing your code yet, and it will go into a multi-line mode for you to type in more code. Now finish your function definition and press enter:

```
function generateRandom() {
...return Math.random()
}
undefined
```

### The _ special variable

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

### The Up arrow key

If you press the `up` arrow key, you will get access to the history of the previous lines of code executed in the current, and even previous REPL sessions.

### Dot commands

The REPL has some special commands, all starting with a dot `.`. They are

- `.help`: shows the dot commands help
- `.editor`: enables editor mode, to write multiline JavaScript code with ease. Once you are in this mode, enter ctrl-D to run the code you wrote.
- `.break`: when inputting a multi-line expression, entering the .break command will abort further input. Same as pressing ctrl-C.
- `.clear`: resets the REPL context to an empty object and clears any multi-line expression currently being input.
- `.load`: loads a JavaScript file, relative to the current working directory
- `.save`: saves all you entered in the REPL session to a file (specify the filename)
- `.exit`: exits the repl (same as pressing ctrl-C two times)

The REPL knows when you are typing a multi-line statement without the need to invoke `.editor`.

For example if you start typing an iteration like this:

```
[1, 2, 3].forEach(num => {
```

and you press `enter`, the REPL will go to a new line that starts with 3 dots, indicating you can now continue to work on that block.

```
... console.log(num)
... })
```

If you type `.break` at the end of a line, the multiline mode will stop and the statement will not be executed.

### Run REPL from JavaScript file

We can import the REPL in a JavaScript file using `repl`.

```
const repl = require('node:repl');
```

Using the repl variable we can perform various operations. To start the REPL command prompt, type in the following line

```
repl.start();
```

Run the file in the command line.

```
node repl.js
```

```
> const n = 10
```

You can pass a string which shows when the REPL starts. The default is '> ' (with a trailing space), but we can define custom prompt.

```
// a Unix style prompt
const local = repl.start('$ ');
```

You can display a message while exiting the REPL

```
local.on('exit', () => {
  console.log('exiting repl');
  process.exit();
});
```

You can read more about the REPL module in the [repl documentation](repl documentation).

## output to the command line using nodejs

# Output to the command line using Node.js

### Basic output using the console module

Node.js provides a `console` module which provides tons of very useful ways to interact with the command line.

It is basically the same as the `console` object you find in the browser.

The most basic and most used method is `console.log()`, which prints the string you pass to it to the console.

If you pass an object, it will render it as a string.

You can pass multiple variables to `console.log`, for example:

```
const x = 'x';
const y = 'y';
console.log(x, y);
```

and Node.js will print both.

We can also format pretty phrases by passing variables and a format specifier.

For example:

```
console.log('My %s has %d ears', 'cat', 2);
```

- `%s` format a variable as a string
- `%d` format a variable as a number
- `%i` format a variable as its integer part only
- `%o` format a variable as an object

Example:

```
console.log('%o', Number);
```

### Clear the console

`console.clear()` clears the console (the behavior might depend on the console used)

### Counting elements

`console.count()` is a handy method.

Take this code:

```
const x = 1;
const y = 2;
const z = 3;

console.count(
  'The value of x is ' + x + ' and has been checked .. how many times?'
);

console.count(
  'The value of x is ' + x + ' and has been checked .. how many times?'
);

console.count(
  'The value of y is ' + y + ' and has been checked .. how many times?'
);
```

What happens is that `console.count()` will count the number of times a string is printed, and print the count next to it:

You can just count apples and oranges:

```
const oranges = ['orange', 'orange'];
const apples = ['just one apple'];

oranges.forEach(fruit => {
  console.count(fruit);
});
apples.forEach(fruit => {
  console.count(fruit);
});
```

### Reset counting

The console.countReset() method resets counter used with console.count().

We will use the apples and orange example to demonstrate this.

```
const oranges = ['orange', 'orange'];
const apples = ['just one apple'];

oranges.forEach(fruit => {
  console.count(fruit);
});
apples.forEach(fruit => {
  console.count(fruit);
});

console.countReset('orange');

oranges.forEach(fruit => {
  console.count(fruit);
});
```

Notice how the call to `console.countReset('orange')` resets the value counter to zero.

### Print the stack trace

There might be cases where it's useful to print the call stack trace of a function, maybe to answer the question *how did you reach that part of the code?*

You can do so using `console.trace()`:

```
const function2 = () => console.trace();
const function1 = () => function2();
function1();
```

This will print the stack trace. This is what's printed if we try this in the Node.js REPL:

```
Trace
    at function2 (repl:1:33)
    at function1 (repl:1:25)
    at repl:1:1
    at ContextifyScript.Script.runInThisContext (vm.js:44:33)
    at REPLServer.defaultEval (repl.js:239:29)
    at bound (domain.js:301:14)
    at REPLServer.runBound [as eval] (domain.js:314:12)
    at REPLServer.onLine (repl.js:440:10)
    at emitOne (events.js:120:20)
    at REPLServer.emit (events.js:210:7)
```

### Calculate the time spent

You can easily calculate how much time a function takes to run, using `time()` and `timeEnd()`

```
const doSomething = () => console.log('test');
const measureDoingSomething = () => {
  console.time('doSomething()');
  // do something, and measure the time it takes
  doSomething();
  console.timeEnd('doSomething()');
};
measureDoingSomething();
```

### stdout and stderr

As we saw console.log is great for printing messages in the Console. This is what's called the standard output, or `stdout`.

`console.error` prints to the `stderr` stream.

It will not appear in the console, but it will appear in the error log.

### Color the output

You can color the output of your text in the console by using escape sequences. An escape sequence is a set of characters that identifies a color.

Example:

```
console.log('\x1b[33m%s\x1b[0m', 'hi!');
```

You can try that in the Node.js REPL, and it will print `hi!` in yellow.

However, this is the low-level way to do this. The simplest way to go about coloring the console output is by using a library. Chalk is such a library, and in addition to coloring it also helps with other styling facilities, like making text bold, italic or underlined.

You install it with `npm install chalk`, then you can use it:

```
const chalk = require('chalk');

console.log(chalk.yellow('hi!'));
```

Using `chalk.yellow` is much more convenient than trying to remember the escape codes, and the code is much more readable.

Check the project link posted above for more usage examples.

### Create a progress bar

Progress is an awesome package to create a progress bar in the console. Install it using `npm install progress`

This snippet creates a 10-step progress bar, and every 100ms one step is completed. When the bar completes we clear the interval:

```
const ProgressBar = require('progress');

const bar = new ProgressBar(':bar', { total: 10 });
const timer = setInterval(() => {
  bar.tick();
  if (bar.complete) {
    clearInterval(timer);
  }
}, 100);
```

## accept input from the command line in nodejs

# Accept input from the command line in Node.js

How to make a Node.js CLI program interactive?

Node.js since version 7 provides the `readline` module to perform exactly this: get input from a readable stream such as the `process.stdin` stream, which during the execution of a Node.js program is the terminal input, one line at a time.

```
const readline = require('node:readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question(`What's your name?`, name => {
  console.log(`Hi ${name}!`);
  rl.close();
});
```

```
import readline from 'node:readline';

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question(`What's your name?`, name => {
  console.log(`Hi ${name}!`);
  rl.close();
});
```

This piece of code asks the user's *name*, and once the text is entered and the user presses enter, we send a greeting.

The `question()` method shows the first parameter (a question) and waits for the user input. It calls the callback function once enter is pressed.

In this callback function, we close the readline interface.

`readline` offers several other methods, please check them out on the package documentation linked above.

If you need to require a password, it's best not to echo it back, but instead show a $*$ symbol.

The simplest way is to use the `readline-sync` package which is very similar in terms of the API and handles this out of the box.

A more complete and abstract solution is provided by the Inquirer.js package.

You can install it using `npm install inquirer`, and then you can replicate the above code like this:

```
const inquirer = require('inquirer');

const questions = [
  {
    type: 'input',
    name: 'name',
    message: "What's your name?",
  },
];

inquirer.prompt(questions).then(answers => {
  console.log(`Hi ${answers.name}!`);
});
```

```
import inquirer from 'inquirer';

const questions = [
  {
    type: 'input',
    name: 'name',
    message: "What's your name?",
  },
];

inquirer.prompt(questions).then(answers => {
  console.log(`Hi ${answers.name}!`);
});
```

Inquirer.js lets you do many things like asking multiple choices, having radio buttons, confirmations, and more.

It's worth knowing all the alternatives, especially the built-in ones provided by Node.js, but if you plan to take CLI input to the next level, Inquirer.js is an optimal choice.

---

## Modules

### publishing node api modules

## How to publish a Node-API version of a package alongside a non-Node-API version

The following steps are illustrated using the package `iotivity-node`:

- First, publish the non-Node-API version:
  - Update the version in `package.json`. For `iotivity-node`, the version becomes `1.2.0-2`.
  - Go through the release checklist (ensure tests/demos/docs are OK)
  - `npm publish`
- Then, publish the Node-API version:
  - Update the version in `package.json`. In the case of `iotivity-node`, the version becomes `1.2.0-3`. For versioning, we recommend following the pre-release version scheme as described by semver.org e.g. `1.2.0-napi`.
  - Go through the release checklist (ensure tests/demos/docs are OK)
  - `npm publish --tag n-api`

In this example, tagging the release with `n-api` has ensured that, although version 1.2.0-3 is later than the non-Node-API published version (1.2.0-2), it will not be installed if someone chooses to install `iotivity-node` by simply running `npm install iotivity-node`. This will install the non-Node-API version by default. The user will have to run `npm install iotivity-node@n-api` to receive the Node-API version. For more information on using tags with npm check out "Using dist-tags".

### How to introduce a dependency on a Node-API version of a package

To add the Node-API version of `iotivity-node` as a dependency, the `package.json` will look like this:

```
"dependencies": {
  "iotivity-node": "n-api"
}
```

> As explained in "Using dist-tags", unlike regular versions, tagged versions cannot be addressed by version ranges such as `"^2.0.0"` inside `package.json`. The reason for this is that the tag refers to exactly one version. So, if the package maintainer chooses to tag a later version of the package using the same tag, `npm update` will receive the later version. This should be acceptable version other than the latest published, the `package.json` dependency will have to refer to the exact version like the following:

```
"dependencies": {
  "iotivity-node": "1.2.0-3"
}
```

# Anatomy of an HTTP Transaction

The purpose of this guide is to impart a solid understanding of the process of Node.js HTTP handling. We'll assume that you know, in a general sense, how HTTP requests work, regardless of language or programming environment. We'll also assume a bit of familiarity with Node.js `EventEmitters` and `Streams`. If you're not quite familiar with them, it's worth taking a quick read through the API docs for each of those.

## Create the Server

Any node web server application will at some point have to create a web server object. This is done by using `createServer`.

```
const http = require('node:http');

const server = http.createServer((request, response) => {
  // magic happens here!
});
```

```
import http from 'node:http';

const server = http.createServer((request, response) => {
  // magic happens here!
});
```

The function that's passed in to `createServer` is called once for every HTTP request that's made against that server, so it's called the request handler. In fact, the `Server` object returned by `createServer` is an `EventEmitter`, and what we have here is just shorthand for creating a `server` object and then adding the listener later.

```
const server = http.createServer();
server.on('request', (request, response) => {
  // the same kind of magic happens here!
});
```

When an HTTP request hits the server, node calls the request handler function with a few handy objects for dealing with the transaction, `request` and `response`. We'll get to those shortly.

In order to actually serve requests, the `listen` method needs to be called on the `server` object. In most cases, all you'll need to pass to `listen` is the port number you want the server to listen on. There are some other options too, so consult the API reference.

## Method, URL and Headers

When handling a request, the first thing you'll probably want to do is look at the method and URL, so that appropriate actions can be taken. Node.js makes this relatively painless by putting handy properties onto the `request` object.

```
const { method, url } = request;
```

> The `request` object is an instance of `IncomingMessage`.

The `method` here will always be a normal HTTP method/verb. The `url` is the full URL without the server, protocol or port. For a typical URL, this means everything after and including the third forward slash.

Headers are also not far away. They're in their own object on `request` called `headers`.

```
const { headers } = request;
const userAgent = headers['user-agent'];
```

It's important to note here that all headers are represented in lower-case only, regardless of how the client actually sent them. This simplifies the task of parsing headers for whatever purpose.

If some headers are repeated, then their values are overwritten or joined together as comma-separated strings, depending on the header. In some cases, this can be problematic, so `rawHeaders` is also available.

## Request Body

When receiving a `POST` or `PUT` request, the request body might be important to your application. Getting at the body data is a little more involved than accessing request headers. The `request` object that's passed in to a handler implements the `ReadableStream` interface. This stream can be listened to or piped elsewhere just like any other stream. We can grab the data right out of the stream by listening to the stream's `'data'` and `'end'` events.

The chunk emitted in each `'data'` event is a `Buffer`. If you know it's going to be string data, the best thing to do is collect the data in an array, then at the `'end'`, concatenate and stringify it.

```
let body = [];
request
  .on('data', chunk => {
    body.push(chunk);
  })
  .on('end', () => {
    body = Buffer.concat(body).toString();
    // at this point, `body` has the entire request body stored in it as a string
  });
```

> This may seem a tad tedious, and in many cases, it is. Luckily, there are modules like `concat-stream` and `body` on `npm` which can help hide away some of this logic. It's important to have a good understanding of what's going on before going down that road, and that's why you're here!

## A Quick Thing About Errors

Since the `request` object is a `ReadableStream`, it's also an `EventEmitter` and behaves like one when an error happens.

An error in the `request` stream presents itself by emitting an `'error'` event on the stream. **If you don't have a listener for that event, the error will be *thrown*, which could crash your Node.js program.** You should therefore add an `'error'` listener on your request streams, even if you just log it and continue on your way. (Though it's probably best to send some kind of HTTP error response. More on that later.)

```
request.on('error', err => {
  // This prints the error message and stack trace to `stderr`.
  console.error(err.stack);
});
```

There are other ways of handling these errors such as other abstractions and tools, but always be aware that errors can and do happen, and you're going to have to deal with them.

## What We've Got so Far

At this point, we've covered creating a server, and grabbing the method, URL, headers and body out of requests. When we put that all together, it might look something like this:

```
const http = require('node:http');

http
  .createServer((request, response) => {
    const { headers, method, url } = request;
    let body = [];
    request
      .on('error', err => {
        console.error(err);
      })
      .on('data', chunk => {
        body.push(chunk);
      })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        // At this point, we have the headers, method, url and body, and can now
        // do whatever we need to in order to respond to this request.
      });
  })
  .listen(8080); // Activates this server, listening on port 8080.
```

```
import http from 'node:http';

http
  .createServer((request, response) => {
    const { headers, method, url } = request;
    let body = [];
    request
      .on('error', err => {
        console.error(err);
      })
      .on('data', chunk => {
        body.push(chunk);
      })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        // At this point, we have the headers, method, url and body, and can now
        // do whatever we need to in order to respond to this request.
      });
  })
  .listen(8080); // Activates this server, listening on port 8080.
```

If we run this example, we'll be able to *receive* requests, but not *respond* to them. In fact, if you hit this example in a web browser, your request would time out, as nothing is being sent back to the client.

So far we haven't touched on the `response` object at all, which is an instance of `ServerResponse`, which is a `WritableStream`. It contains many useful methods for sending data back to the client. We'll cover that next.

## HTTP Status Code

If you don't bother setting it, the HTTP status code on a response will always be 200. Of course, not every HTTP response warrants this, and at some point you'll definitely want to send a different status code. To do that, you can set the `statusCode` property.

```
response.statusCode = 404; // Tell the client that the resource wasn't found.
```

There are some other shortcuts to this, as we'll see soon.

## Setting Response Headers

Headers are set through a convenient method called `setHeader`.

```
response.setHeader('Content-Type', 'application/json');
response.setHeader('X-Powered-By', 'bacon');
```

When setting the headers on a response, the case is insensitive on their names. If you set a header repeatedly, the last value you set is the value that gets sent.

## Explicitly Sending Header Data

The methods of setting the headers and status code that we've already discussed assume that you're using "implicit headers". This means you're counting on node to send the headers for you at the correct time before you start sending body data.

If you want, you can *explicitly* write the headers to the response stream. To do this, there's a method called `writeHead`, which writes the status code and the headers to the stream.

```
response.writeHead(200, {
  'Content-Type': 'application/json',
  'X-Powered-By': 'bacon',
});
```

Once you've set the headers (either implicitly or explicitly), you're ready to start sending response data.

## Sending Response Body

Since the `response` object is a `WritableStream`, writing a response body out to the client is just a matter of using the usual stream methods.

```
response.write('<html>');
response.write('<body>');
response.write('<h1>Hello, World!</h1>');
response.write('</body>');
response.write('</html>');
response.end();
```

The `end` function on streams can also take in some optional data to send as the last bit of data on the stream, so we can simplify the example above as follows.

```
response.end('<html><body><h1>Hello, World!</h1></body></html>');
```

> It's important to set the status and headers *before* you start writing chunks of data to the body. This makes sense, since headers come before the body in HTTP responses.

## Another Quick Thing About Errors

The `response` stream can also emit `'error'` events, and at some point you're going to have to deal with that as well. All of the advice for `request` stream errors still applies here.

## Put It All Together

Now that we've learned about making HTTP responses, let's put it all together. Building on the earlier example, we're going to make a server that sends back all of the data that was sent to us by the user. We'll format that data as JSON using `JSON.stringify`.

```
const http = require('node:http');

http
  .createServer((request, response) => {
    const { headers, method, url } = request;
    let body = [];
    request
      .on('error', err => {
        console.error(err);
      })
      .on('data', chunk => {
        body.push(chunk);
      })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        // BEGINNING OF NEW STUFF

        response.on('error', err => {
          console.error(err);
        });

        response.statusCode = 200;
        response.setHeader('Content-Type', 'application/json');
        // Note: the 2 lines above could be replaced with this next one:
        // response.writeHead(200, {'Content-Type': 'application/json'})

        const responseBody = { headers, method, url, body };

        response.write(JSON.stringify(responseBody));
        response.end();
        // Note: the 2 lines above could be replaced with this next one:
        // response.end(JSON.stringify(responseBody))

        // END OF NEW STUFF
      });
  })
  .listen(8080);
```

```
import http from 'node:http';

http
  .createServer((request, response) => {
    const { headers, method, url } = request;
    let body = [];
    request
      .on('error', err => {
        console.error(err);
      })
      .on('data', chunk => {
        body.push(chunk);
      })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        // BEGINNING OF NEW STUFF

        response.on('error', err => {
          console.error(err);
        });

        response.statusCode = 200;
        response.setHeader('Content-Type', 'application/json');
        // Note: the 2 lines above could be replaced with this next one:
        // response.writeHead(200, {'Content-Type': 'application/json'})

        const responseBody = { headers, method, url, body };

        response.write(JSON.stringify(responseBody));
        response.end();
        // Note: the 2 lines above could be replaced with this next one:
        // response.end(JSON.stringify(responseBody))

        // END OF NEW STUFF
      });
  })
  .listen(8080);
```

## Echo Server Example

Let's simplify the previous example to make a simple echo server, which just sends whatever data is received in the request right back in the response. All we need to do is grab the data from the request stream and write that data to the response stream, similar to what we did previously.

```
const http = require('node:http');

http
  .createServer((request, response) => {
    let body = [];
    request
      .on('data', chunk => {
        body.push(chunk);
      })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        response.end(body);
      });
  })
  .listen(8080);
```

```
import http from 'node:http';

http
  .createServer((request, response) => {
    let body = [];
    request
      .on('data', chunk => {
        body.push(chunk);
      })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        response.end(body);
      });
  })
  .listen(8080);
```

Now let's tweak this. We want to only send an echo under the following conditions:

- The request method is POST.
- The URL is `/echo`.

In any other case, we want to simply respond with a 404.

```
const http = require('node:http');

http
  .createServer((request, response) => {
    if (request.method === 'POST' && request.url === '/echo') {
      let body = [];
      request
        .on('data', chunk => {
          body.push(chunk);
        })
        .on('end', () => {
          body = Buffer.concat(body).toString();
          response.end(body);
        });
    } else {
      response.statusCode = 404;
      response.end();
    }
  })
  .listen(8080);
```

```
import http from 'node:http';

http
  .createServer((request, response) => {
    if (request.method === 'POST' && request.url === '/echo') {
      let body = [];
      request
        .on('data', chunk => {
          body.push(chunk);
        })
        .on('end', () => {
          body = Buffer.concat(body).toString();
          response.end(body);
        });
    } else {
      response.statusCode = 404;
      response.end();
    }
  })
  .listen(8080);
```

> By checking the URL in this way, we're doing a form of "routing". Other forms of routing can be as simple as `switch` statements or as complex as whole frameworks like `express`. If you're looking for something that does routing and nothing else, try `router`.

Great! Now let's take a stab at simplifying this. Remember, the `request` object is a `ReadableStream` and the `response` object is a `WritableStream`. That means we can use `pipe` to direct data from one to the other. That's exactly what we want for an echo server!

```
const http = require('node:http');

http
  .createServer((request, response) => {
    if (request.method === 'POST' && request.url === '/echo') {
      request.pipe(response);
    } else {
      response.statusCode = 404;
      response.end();
    }
  })
  .listen(8080);
```

```
import http from 'node:http';

http
  .createServer((request, response) => {
    if (request.method === 'POST' && request.url === '/echo') {
      request.pipe(response);
    } else {
      response.statusCode = 404;
      response.end();
    }
  })
  .listen(8080);
```

Yay streams!

We're not quite done yet though. As mentioned multiple times in this guide, errors can and do happen, and we need to deal with them.

To handle errors on the request stream, we'll log the error to `stderr` and send a 400 status code to indicate a `Bad Request`. In a real-world application, though, we'd want to inspect the error to figure out what the correct status code and message would be. As usual with errors, you should consult the `Error` documentation.

On the response, we'll just log the error to `stderr`.

```
const http = require('node:http');

http
  .createServer((request, response) => {
    request.on('error', err => {
      console.error(err);
      response.statusCode = 400;
      response.end();
    });
    response.on('error', err => {
      console.error(err);
    });
    if (request.method === 'POST' && request.url === '/echo') {
      request.pipe(response);
    } else {
      response.statusCode = 404;
      response.end();
    }
  })
  .listen(8080);
```

```
import http from 'node:http';

http
  .createServer((request, response) => {
    request.on('error', err => {
      console.error(err);
      response.statusCode = 400;
      response.end();
    });
    response.on('error', err => {
      console.error(err);
    });
    if (request.method === 'POST' && request.url === '/echo') {
      request.pipe(response);
    } else {
      response.statusCode = 404;
      response.end();
    }
  })
  .listen(8080);
```

We've now covered most of the basics of handling HTTP requests. At this point, you should be able to:

- Instantiate an HTTP server with a request handler function, and have it listen on a port.
- Get headers, URL, method and body data from `request` objects.
- Make routing decisions based on URL and/or other data in `request` objects.
- Send headers, HTTP status codes and body data via `response` objects.
- Pipe data from `request` objects and to `response` objects.
- Handle stream errors in both the `request` and `response` streams.

From these basics, Node.js HTTP servers for many typical use cases can be constructed. There are plenty of other things these APIs provide, so be sure to read through the API docs for `EventEmitters`, `Streams`, and `HTTP`.

## abi stability

# ABI Stability

## Introduction

An Application Binary Interface (ABI) is a way for programs to call functions and use data structures from other compiled programs. It is the compiled version of an Application Programming Interface (API). In other words, the headers files describing the classes, functions, data structures, enumerations, and constants which enable an application to perform a desired task correspond by way of compilation to a set of addresses and expected parameter values and memory structure sizes and layouts with which the provider of the ABI was compiled.

The application using the ABI must be compiled such that the available addresses, expected parameter values, and memory structure sizes and layouts agree with those with which the ABI provider was compiled. This is usually accomplished by compiling against the headers provided by the ABI provider.

Since the provider of the ABI and the user of the ABI may be compiled at different times with different versions of the compiler, a portion of the responsibility for ensuring ABI compatibility lies with the compiler. Different versions of the compiler, perhaps provided by different vendors, must all produce the same ABI from a header file with a certain content, and must produce code for the application using the ABI that accesses the API described in a given header according to the conventions of the ABI resulting from the description in the header. Modern compilers have a fairly good track record of not breaking the ABI compatibility of the applications they compile.

The remaining responsibility for ensuring ABI compatibility lies with the team maintaining the header files which provide the API that results, upon compilation, in the ABI that is to remain stable. Changes to the header files can be made, but the nature of the changes has to be closely tracked to ensure that, upon compilation, the ABI does not change in a way that will render existing users of the ABI incompatible with the new version.

## ABI Stability in Node.js

Node.js provides header files maintained by several independent teams. For example, header files such as `node.h` and `node_buffer.h` are maintained by the Node.js team. `v8.h` is maintained by the V8 team, which, although in close co-operation with the Node.js team, is independent, and with its own schedule and priorities. Thus, the Node.js team has only partial control over the changes that are introduced in the headers the project provides. As a result, the Node.js project has adopted semantic versioning. This ensures that the APIs provided by the project will result in a stable ABI for all minor and patch versions of Node.js released within one major version. In practice, this means that the Node.js project has committed itself to ensuring that a Node.js native addon compiled against a given major version of Node.js will load successfully when loaded by any Node.js minor or patch version within the major version against which it was compiled.

## N-API

Demand has arisen for equipping Node.js with an API that results in an ABI that remains stable across multiple Node.js major versions. The motivation for creating such an API is as follows:

- The JavaScript language has remained compatible with itself since its very early days, whereas the ABI of the engine executing the JavaScript code changes with every major version of Node.js. This means that applications consisting of Node.js packages written entirely in JavaScript need not be recompiled, reinstalled, or redeployed as a new major version of Node.js is dropped into the production environment in which such applications run. In contrast, if an application depends on a package that contains a native addon, the application has to be recompiled, reinstalled, and redeployed whenever a new major version of Node.js is introduced into the production environment. This disparity between Node.js packages containing native addons and those that are written entirely in JavaScript has added to the maintenance burden of production systems which rely on native addons.
- Other projects have started to produce JavaScript interfaces that are essentially alternative implementations of Node.js. Since these projects are usually built on a different JavaScript engine than V8, their native addons necessarily take on a different structure and use a different API. Nevertheless, using a single API for a native addon across different implementations of the Node.js JavaScript API would allow these projects to take advantage of the ecosystem of JavaScript packages that has accrued around Node.js.
- Node.js may contain a different JavaScript engine in the future. This means that, externally, all Node.js interfaces would remain the same, but the V8 header file would be absent. Such a step would cause the disruption of the Node.js ecosystem in general, and that of the native addons in particular, if an API that is JavaScript engine agnostic is not first provided by Node.js and adopted by native addons.

To these ends Node.js has introduced N-API in version 8.6.0 and marked it as a stable component of the project as of Node.js 8.12.0. The API is defined in the headers `node_api.h` and `node_api_types.h`, and provides a forward- compatibility guarantee that crosses the Node.js major version boundary. The guarantee can be stated as follows:

**A given version *n* of N-API will be available in the major version of Node.js in which it was published, and in all subsequent versions of Node.js, including subsequent major versions.**

A native addon author can take advantage of the N-API forward compatibility guarantee by ensuring that the addon makes use only of APIs defined in `node_api.h` and data structures and constants defined in `node_api_types.h`. By doing so, the author facilitates adoption of their addon by indicating to production users that the maintenance burden for their application will increase no more by the addition of the native addon to their project than it would by the addition of a package written purely in JavaScript.

N-API is versioned because new APIs are added from time to time. Unlike semantic versioning, N-API versioning is cumulative. That is, each version of N-API conveys the same meaning as a minor version in the semver system, meaning that all changes made to N-API will be backwards compatible. Additionally, new N-APIs are added under an experimental flag to give the community an opportunity to vet them in a production environment. Experimental status means that, although care has been taken to ensure that the new API will not have to be modified in an ABI-incompatible way in the future, it has not yet been sufficiently proven in production to be correct and useful as designed and, as such, may undergo ABI-incompatible changes before it is finally incorporated into a forthcoming version of N-API. That is, an experimental N-API is not yet covered by the forward compatibility guarantee.

# Backpressuring in Streams

There is a general problem that occurs during data handling called `backpressure` and describes a buildup of data behind a buffer during data transfer. When the receiving end of the transfer has complex operations, or is slower for whatever reason, there is a tendency for data from the incoming source to accumulate, like a clog.

To solve this problem, there must be a delegation system in place to ensure a smooth flow of data from one source to another. Different communities have resolved this issue uniquely to their programs, Unix pipes and TCP sockets are good examples of this, and are often referred to as *flow control*. In Node.js, streams have been the adopted solution.

The purpose of this guide is to further detail what backpressure is, and how exactly streams address this in Node.js' source code. The second part of the guide will introduce suggested best practices to ensure your application's code is safe and optimized when implementing streams.

We assume a little familiarity with the general definition of `backpressure`, `Buffer`, and `EventEmitters` in Node.js, as well as some experience with `Stream`. If you haven't read through those docs, it's not a bad idea to take a look at the API documentation first, as it will help expand your understanding while reading this guide.

## The Problem with Data Handling

In a computer system, data is transferred from one process to another through pipes, sockets, and signals. In Node.js, we find a similar mechanism called `Stream`. Streams are great! They do so much for Node.js and almost every part of the internal codebase utilizes that module. As a developer, you are more than encouraged to use them too!

```
const readline = require('node:readline');

// process.stdin and process.stdout are both instances of Streams.
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question('Why should you use streams? ', answer => {
  console.log(`Maybe it's ${answer}, maybe it's because they are awesome! :)`);

  rl.close();
});
```

```
import readline from 'node:readline';

// process.stdin and process.stdout are both instances of Streams.
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question('Why should you use streams? ', answer => {
  console.log(`Maybe it's ${answer}, maybe it's because they are awesome! :)`);

  rl.close();
});
```

A good example of why the backpressure mechanism implemented through streams is a great optimization can be demonstrated by comparing the internal system tools from Node.js' `Stream` implementation.

In one scenario, we will take a large file (approximately ~9 GB) and compress it using the familiar `zip(1)` tool.

```
zip The.Matrix.1080p.mkv
```

While that will take a few minutes to complete, in another shell we may run a script that takes Node.js' module `zlib`, that wraps around another compression tool, `gzip(1)`.

```
const gzip = require('node:zlib').createGzip();
const fs = require('node:fs');

const inp = fs.createReadStream('The.Matrix.1080p.mkv');
const out = fs.createWriteStream('The.Matrix.1080p.mkv.gz');

inp.pipe(gzip).pipe(out);
```

```
import { createGzip } from 'node:zlib';
import { createReadStream, createWriteStream } from 'node:fs';

const gzip = createGzip();

const inp = createReadStream('The.Matrix.1080p.mkv');
const out = createWriteStream('The.Matrix.1080p.mkv.gz');

inp.pipe(gzip).pipe(out);
```

To test the results, try opening each compressed file. The file compressed by the `zip(1)` tool will notify you the file is corrupt, whereas the compression finished by `Stream` will decompress without error.

> In this example, we use `.pipe()` to get the data source from one end to the other. However, notice there are no proper error handlers attached. If a chunk of data were to fail to be properly received, the `Readable` source or `gzip` stream will not be destroyed. `pump` is a utility tool that would properly destroy all the streams in a pipeline if one of them fails or closes, and is a must-have in this case!

`pump` is only necessary for Node.js 8.x or earlier, as for Node.js 10.x or later version, `pipeline` is introduced to replace for `pump`. This is a module method to pipe between streams forwarding errors and properly cleaning up and providing a callback when the pipeline is complete.

Here is an example of using pipeline:

```
const { pipeline } = require('node:stream');
const fs = require('node:fs');
const zlib = require('node:zlib');

// Use the pipeline API to easily pipe a series of streams
// together and get notified when the pipeline is fully done.
// A pipeline to gzip a potentially huge video file efficiently:

pipeline(
  fs.createReadStream('The.Matrix.1080p.mkv'),
  zlib.createGzip(),
  fs.createWriteStream('The.Matrix.1080p.mkv.gz'),
  err => {
    if (err) {
      console.error('Pipeline failed', err);
    } else {
      console.log('Pipeline succeeded');
    }
  }
);
```

```
import { pipeline } from 'node:stream';
import fs from 'node:fs';
import zlib from 'node:zlib';

// Use the pipeline API to easily pipe a series of streams
// together and get notified when the pipeline is fully done.
// A pipeline to gzip a potentially huge video file efficiently:

pipeline(
  fs.createReadStream('The.Matrix.1080p.mkv'),
  zlib.createGzip(),
  fs.createWriteStream('The.Matrix.1080p.mkv.gz'),
  err => {
    if (err) {
      console.error('Pipeline failed', err);
    } else {
      console.log('Pipeline succeeded');
    }
  }
);
```

You can also use the `stream/promises` module to use pipeline with `async` / `await`:

```
const { pipeline } = require('node:stream/promises');
const fs = require('node:fs');
const zlib = require('node:zlib');

async function run() {
  try {
    await pipeline(
      fs.createReadStream('The.Matrix.1080p.mkv'),
      zlib.createGzip(),
      fs.createWriteStream('The.Matrix.1080p.mkv.gz')
    );
    console.log('Pipeline succeeded');
  } catch (err) {
    console.error('Pipeline failed', err);
  }
}
```

```
import { pipeline } from 'node:stream/promises';
import fs from 'node:fs';
import zlib from 'node:zlib';

async function run() {
  try {
    await pipeline(
      fs.createReadStream('The.Matrix.1080p.mkv'),
      zlib.createGzip(),
      fs.createWriteStream('The.Matrix.1080p.mkv.gz')
    );
    console.log('Pipeline succeeded');
  } catch (err) {
    console.error('Pipeline failed', err);
  }
}
```

## Too Much Data, Too Quickly

There are instances where a `Readable` stream might give data to the `Writable` much too quickly — much more than the consumer can handle!

When that occurs, the consumer will begin to queue all the chunks of data for later consumption. The write queue will get longer and longer, and because of this more data must be kept in memory until the entire process has been completed.

Writing to a disk is a lot slower than reading from a disk, thus, when we are trying to compress a file and write it to our hard disk, backpressure will occur because the write disk will not be able to keep up with the speed from the read.

```
// Secretly the stream is saying: "whoa, whoa! hang on, this is way too much!"
// Data will begin to build up on the read side of the data buffer as
// `write` tries to keep up with the incoming data flow.
inp.pipe(gzip).pipe(outputFile);
```

This is why a backpressure mechanism is important. If a backpressure system was not present, the process would use up your system's memory, effectively slowing down other processes, and monopolizing a large part of your system until completion.

This results in a few things:

- Slowing down all other current processes
- A very overworked garbage collector
- Memory exhaustion

In the following examples, we will take out the return value of the `.write()` function and change it to `true`, which effectively disables backpressure support in Node.js core. In any reference to 'modified' binary, we are talking about running the `node` binary without the `return ret;` line, and instead with the replaced `return true;`.

## Excess Drag on Garbage Collection

Let's take a look at a quick benchmark. Using the same example from above, we ran a few time trials to get a median time for both binaries.

```
   trial (#)  | `node` binary (ms) | modified `node` binary (ms)
=================================================================
      1       |      56924         |           55011
      2       |      52686         |           55869
      3       |      59479         |           54043
      4       |      54473         |           55229
      5       |      52933         |           59723
=================================================================
average time: |      55299         |           55975
```

Both take around a minute to run, so there's not much of a difference at all, but let's take a closer look to confirm whether our suspicions are correct. We use the Linux tool `dtrace` to evaluate what's happening with the V8 garbage collector.

The GC (garbage collector) measured time indicates the intervals of a full cycle of a single sweep done by the garbage collector:

```
approx. time (ms) | GC (ms) | modified GC (ms)
=================================================
          0       |    0    |      0
          1       |    0    |      0
         40       |    0    |      2
        170       |    3    |      1
        300       |    3    |      1

         *             *           *
         *             *           *
         *             *           *

      39000       |    6    |     26
      42000       |    6    |     21
      47000       |    5    |     32
      50000       |    8    |     28
      54000       |    6    |     35
```

While the two processes start the same and seem to work the GC at the same rate, it becomes evident that after a few seconds with a properly working backpressure system in place, it spreads the GC load across consistent intervals of 4-8 milliseconds until the end of the data transfer.

However, when a backpressure system is not in place, the V8 garbage collection starts to drag out. The normal binary called the GC fires approximately **75** times in a minute, whereas, the modified binary fires only **36** times.

This is the slow and gradual debt accumulating from growing memory usage. As data gets transferred, without a backpressure system in place, more memory is being used for each chunk transfer.

The more memory that is being allocated, the more the GC has to take care of in one sweep. The bigger the sweep, the more the GC needs to decide what can be freed up, and scanning for detached pointers in a larger memory space will consume more computing power.

## Memory Exhaustion

To determine the memory consumption of each binary, we've clocked each process with `/usr/bin/time -lp sudo ./node ./backpressure-example/zlib.js` individually.

This is the output on the normal binary:

```
Respecting the return value of .write()
=============================================
real        58.88
user        56.79
sys          8.79
  87810048  maximum resident set size
         0  average shared memory size
         0  average unshared data size
         0  average unshared stack size
     19427  page reclaims
      3134  page faults
         0  swaps
         5  block input operations
       194  block output operations
         0  messages sent
         0  messages received
         1  signals received
        12  voluntary context switches
    666037  involuntary context switches
```

The maximum byte size occupied by virtual memory turns out to be approximately 87.81 mb.

And now changing the return value of the `.write()` function, we get:

```
Without respecting the return value of .write():
==================================================
real        54.48
user        53.15
sys          7.43
1524965376  maximum resident set size
         0  average shared memory size
         0  average unshared data size
         0  average unshared stack size
    373617  page reclaims
      3139  page faults
         0  swaps
        18  block input operations
       199  block output operations
         0  messages sent
         0  messages received
         1  signals received
        25  voluntary context switches
    629566  involuntary context switches
```

The maximum byte size occupied by virtual memory turns out to be approximately 1.52 gb.

Without streams in place to delegate the backpressure, there is an order of magnitude greater of memory space being allocated - a huge margin of difference between the same process!

This experiment shows how optimized and cost-effective Node.js' backpressure mechanism is for your computing system. Now, let's do a breakdown of how it works!

## How Does Backpressure Resolve These Issues?

There are different functions to transfer data from one process to another. In Node.js, there is an internal built-in function called `.pipe()`. There are other packages out there you can use too! Ultimately though, at the basic level of this process, we have two separate components: the *source* of the data and the *consumer*.

When `.pipe()` is called from the source, it signals to the consumer that there is data to be transferred. The pipe function helps to set up the appropriate backpressure closures for the event triggers.

In Node.js the source is a `Readable` stream and the consumer is the `Writable` stream (both of these may be interchanged with a `Duplex` or a `Transform` stream, but that is out-of-scope for this guide).

The moment that backpressure is triggered can be narrowed exactly to the return value of a `Writable`'s `.write()` function. This return value is determined by a few conditions, of course.

In any scenario where the data buffer has exceeded the `highWaterMark` or the write queue is currently busy, `.write()` will return `false`.

When a `false` value is returned, the backpressure system kicks in. It will pause the incoming `Readable` stream from sending any data and wait until the consumer is ready again. Once the data buffer is emptied, a `'drain'` event will be emitted and resume the incoming data flow.

Once the queue is finished, backpressure will allow data to be sent again. The space in memory that was being used will free itself up and prepare for the next batch of data.

This effectively allows a fixed amount of memory to be used at any given time for a `.pipe()` function. There will be no memory leakage, and no infinite buffering, and the garbage collector will only have to deal with one area in memory!

So, if backpressure is so important, why have you (probably) not heard of it? Well, the answer is simple: Node.js does all of this automatically for you.

That's so great! But also not so great when we are trying to understand how to implement our custom streams.

> In most machines, there is a byte size that determines when a buffer is full (which will vary across different machines). Node.js allows you to set your custom `highWaterMark`, but commonly, the default is set to 16kb (16384, or 16 for objectMode streams). In

instances where you might want to raise that value, go for it, but do so with caution!

## Lifecycle of `.pipe()`

To achieve a better understanding of backpressure, here is a flow-chart on the lifecycle of a `Readable` stream being piped into a `Writable` stream:

```
                                                     +===================+
                         x-->  Piping functions   +-->   src.pipe(dest)  |
                         x     are set up during     |===================|
                         x     the .pipe method.     |  Event callbacks  |
  +===============+      x                           |-|
  |   Your Data   |      x     They exist outside    | .on('close', cb)  |
  +=======+=======+      x     the data flow, but    | .on('data', cb)   |
          |              x     importantly attach    | .on('drain', cb)  |
          |              x     events, and their     | .on('unpipe', cb) |
++    x     respective callbacks. | .on('error', cb)   |
|  Readable Stream  +->  Writable Stream  ++
  ^       |       +-> if (!chunk)               |    Is this chunk too big?  |
  ^       |       |     emit .end();            |    Is the queue busy?      |
  |       |       +-> else                     +-++
  |       ^       |     emit .write();                |                |
  |       ^       ^                                 +--vv--<  No  |        |  Yes  |
  ^       |                                         ++        ++
  ^       |                                                            |
  |       ^               emit .pause();          +=================+     |
  |       ^--+  return false;  <+
  |                                               +=================+         |
  |                                                                           |
  ^            when queue is empty     +============+                         |
  ^^^---<
                                       +============+
```

> If you are setting up a pipeline to chain together a few streams to manipulate your data, you will most likely be implementing `Transform` stream.

In this case, your output from your `Readable` stream will enter in the `Transform` and will pipe into the `Writable`.

```
Readable.pipe(Transformable).pipe(Writable);
```

Backpressure will be automatically applied, but note that both the incoming and outgoing `highWaterMark` of the `Transform` stream may be manipulated and will affect the backpressure system.

## Backpressure Guidelines

Since Node.js v0.10, the `Stream` class has offered the ability to modify the behavior of the `.read()` or `.write()` by using the underscore version of these respective functions (`._read()` and `._write()`).

There are guidelines documented for implementing Readable streams and implementing Writable streams. We will assume you've read these over, and the next section will go a little bit more in-depth.

## Rules to Abide By When Implementing Custom Streams

The golden rule of streams is **to always respect backpressure**. What constitutes as best practice is non-contradictory practice. So long as you are careful to avoid behaviors that conflict with internal backpressure support, you can be sure you're following good practice.

In general,

1. Never `.push()` if you are not asked.
2. Never call `.write()` after it returns false but wait for 'drain' instead.
3. Streams changes between different Node.js versions, and the library you use. Be careful and test things.

> In regards to point 3, an incredibly useful package for building browser streams is `readable-stream`. Rodd Vagg has written a great blog post describing the utility of this library. In short, it provides a type of automated graceful degradation for `Readable` streams, and supports older versions of browsers and Node.js.

## Rules specific to Readable Streams

So far, we have taken a look at how `.write()` affects backpressure and have focused much on the `Writable` stream. Because of Node.js' functionality, data is technically flowing downstream from `Readable` to `Writable`. However, as we can observe in any transmission of data, matter, or energy, the source is just as important as the destination, and the `Readable` stream is vital to how backpressure is handled.

Both these processes rely on one another to communicate effectively, if the `Readable` ignores when the `Writable` stream asks for it to stop sending in data, it can be just as problematic as when the `.write()`'s return value is incorrect.

So, as well as respecting the `.write()` return, we must also respect the return value of `.push()` used in the `._read()` method. If `.push()` returns a `false` value, the stream will stop reading from the source. Otherwise, it will continue without pause.

Here is an example of bad practice using `.push()`:

```
// This is problematic as it completely ignores the return value from the push
// which may be a signal for backpressure from the destination stream!
class MyReadable extends Readable {
  _read(size) {
    let chunk;
    while (null !== (chunk = getNextChunk())) {
      this.push(chunk);
    }
  }
}
```

Additionally, from outside the custom stream, there are pitfalls to ignoring backpressure. In this counter-example of good practice, the application's code forces data through whenever it is available (signaled by the `'data'` event):

```
// This ignores the backpressure mechanisms Node.js has set in place,
// and unconditionally pushes through data, regardless if the
// destination stream is ready for it or not.
readable.on('data', data => writable.write(data));
```

Here's an example of using `.push()` with a Readable stream.

```
const { Readable } = require('node:stream');

// Create a custom Readable stream
const myReadableStream = new Readable({
  objectMode: true,
  read(size) {
    // Push some data onto the stream
    this.push({ message: 'Hello, world!' });
    this.push(null); // Mark the end of the stream
  },
});

// Consume the stream
myReadableStream.on('data', chunk => {
  console.log(chunk);
});

// Output:
// { message: 'Hello, world!' }
```

```
import { Readable } from 'node:stream';

// Create a custom Readable stream
const myReadableStream = new Readable({
  objectMode: true,
  read(size) {
    // Push some data onto the stream
    this.push({ message: 'Hello, world!' });
    this.push(null); // Mark the end of the stream
  },
});

// Consume the stream
myReadableStream.on('data', chunk => {
  console.log(chunk);
});
```

```
// Output:
// { message: 'Hello, world!' }
```

In this example, we create a custom Readable stream that pushes a single object onto the stream using `.push()`. The `._read()` method is called when the stream is ready to consume data, and in this case, we immediately push some data onto the stream and mark the end of the stream by pushing null.

We then consume the stream by listening for the 'data' event and logging each chunk of data that is pushed onto the stream. In this case, we only push a single chunk of data onto the stream, so we only see one log message.

## Rules specific to Writable Streams

Recall that a `.write()` may return true or false dependent on some conditions. Luckily for us, when building our own `Writable` stream, the `stream state machine` will handle our callbacks and determine when to handle backpressure and optimize the flow of data for us.

However, when we want to use a `Writable` directly, we must respect the `.write()` return value and pay close attention to these conditions:

- If the write queue is busy, `.write()` will return false.
- If the data chunk is too large, `.write()` will return false (the limit is indicated by the variable, `highWaterMark`).

```
// This writable is invalid because of the async nature of JavaScript callbacks.
// Without a return statement for each callback prior to the last,
// there is a great chance multiple callbacks will be called.
class MyWritable extends Writable {
  _write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) callback();
    else if (chunk.toString().indexOf('b') >= 0) callback();
    callback();
  }
}

// The proper way to write this would be:
if (chunk.contains('a')) return callback();
if (chunk.contains('b')) return callback();
callback();
```

There are also some things to look out for when implementing `._writev()`. The function is coupled with `.cork()`, but there is a common mistake when writing:

```
// Using .uncork() twice here makes two calls on the C++ layer, rendering the
// cork/uncork technique useless.
ws.cork();
ws.write('hello ');
ws.write('world ');
ws.uncork();

ws.cork();
ws.write('from ');
ws.write('Matteo');
ws.uncork();

// The correct way to write this is to utilize process.nextTick(), which fires
// on the next event loop.
ws.cork();
ws.write('hello ');
ws.write('world ');
process.nextTick(doUncork, ws);

ws.cork();
ws.write('from ');
ws.write('Matteo');
process.nextTick(doUncork, ws);

// As a global function.
function doUncork(stream) {
  stream.uncork();
}
```

`.cork()` can be called as many times as we want, we just need to be careful to call `.uncork()` the same amount of times to make it flow again.

## Conclusion

Streams are an often-used module in Node.js. They are important to the internal structure, and for developers, to expand and connect across the Node.js modules ecosystem.

Hopefully, you will now be able to troubleshoot and safely code your own `Writable` and `Readable` streams with backpressure in mind, and share your knowledge with colleagues and friends.

Be sure to read up more on `Stream` for other API functions to help improve and unleash your streaming capabilities when building an application with Node.js.

## Diagnostics

### user journey

## User Journey

These diagnostics guides were created by the Diagnostics Working Group with the objective of providing guidance when diagnosing an issue in a user's application.

The documentation project is organized based on user journey. Those journeys are a coherent set of step-by-step procedures that a user can follow to root-cause their issues.

# Memory

In this document you can learn about how to debug memory related issues.

## My process runs out of memory

Node.js *(JavaScript)* is a garbage collected language, so having memory leaks is possible through retainers. As Node.js applications are usually multi-tenant, business critical, and long-running, providing an accessible and efficient way of finding a memory leak is essential.

### Symptoms

The user observes continuously increasing memory usage *(can be fast or slow, over days or even weeks)* then sees the process crashing and restarting by the process manager. The process is maybe running slower than before and the restarts cause some requests to fail *(load balancer responds with 502)*.

### Side Effects

- Process restarts due to the memory exhaustion and requests are dropped on the floor
- Increased GC activity leads to higher CPU usage and slower response time
  - GC blocking the Event Loop causing slowness
- Increased memory swapping slows down the process (GC activity)
- May not have enough available memory to get a Heap Snapshot

## My process utilizes memory inefficiently

### Symptoms

The application uses an unexpected amount of memory and/or we observe elevated garbage collector activity.

### Side Effects

- An elevated number of page faults
- Higher GC activity and CPU usage

## Debugging

Most memory issues can be solved by determining how much space our specific type of objects take and what variables are preventing them from being garbage collected. It can also help to know the allocation pattern of our program over time.

- Using Heap Profiler
- Using Heap Snapshot
- GC Traces

## live debugging/index

# Live Debugging

In this document you can learn about how to live debug a Node.js process.

## My application doesn’t behave as expected

### Symptoms

The user may observe that the application doesn’t provide the expected output for certain inputs, for example, an HTTP server returns a JSON response where certain fields are empty. Various things can go wrong in the process but in this use case, we are mainly focused on the application logic and its correctness.

### Debugging

In this use case, the user would like to understand the code path that our application executes for a certain trigger like an incoming HTTP request. They may also want to step through the code and control the execution as well as inspect what values variables hold in memory.

- Using Inspector

# Poor Performance

In this document you can learn about how to profile a Node.js process.

## My application has a poor performance

### Symptoms

My applications latency is high and I have already confirmed that the bottleneck is not my dependencies like databases and downstream services. So I suspect that my application spends significant time to run code or process information.

You are satisfied with your application performance in general but would like to understand which part of our application can be improved to run faster or more efficient. It can be useful when we want to improve the user experience or save computation cost.

### Debugging

In this use-case, we are interested in code pieces that use more CPU cycles than the others. When we do this locally, we usually try to optimize our code.

This document provides two simple ways to profile a Node.js application:

- Using V8 Sampling Profiler
- Using Linux Perf

## flame graphs

# Flame Graphs

## What's a flame graph useful for?

Flame graphs are a way of visualizing CPU time spent in functions. They can help you pin down where you spend too much time doing synchronous operations.

## How to create a flame graph

You might have heard creating a flame graph for Node.js is difficult, but that's not true (anymore). Solaris vms are no longer needed for flame graphs!

Flame graphs are generated from `perf` output, which is not a node-specific tool. While it's the most powerful way to visualize CPU time spent, it may have issues with how JavaScript code is optimized in Node.js 8 and above. See perf output issues section below.

### Use a pre-packaged tool

If you want a single step that produces a flame graph locally, try 0x

For diagnosing production deployments, read these notes: 0x production servers.

### Create a flame graph with system perf tools

The purpose of this guide is to show the steps involved in creating a flame graph and keep you in control of each step.

If you want to understand each step better, take a look at the sections that follow where we go into more detail.

Now let's get to work.

1. Install `perf` (usually available through the linux-tools-common package if not already installed)
2. Try running `perf` - it might complain about missing kernel modules, install them too
3. Run node with perf enabled (see perf output issues for tips specific to Node.js versions)

   ```
   perf record -e cycles:u -g -- node --perf-basic-prof app.js
   ```

4. Disregard warnings unless they're saying you can't run perf due to missing packages; you may get some warnings about not being able to access kernel module samples which you're not after anyway.
5. Run `perf script > perfs.out` to generate the data file you'll visualize in a moment. It's useful to apply some cleanup for a more readable graph
6. Install stackvis if not yet installed `npm i -g stackvis`
7. Run `stackvis perf < perfs.out > flamegraph.htm`

Now open the flame graph file in your favorite browser and watch it burn. It's color-coded so you can focus on the most saturated orange bars first. They're likely to represent CPU heavy functions.

Worth mentioning - if you click an element of a flame graph a zoom-in of its surroundings will be displayed above the graph.

### Using `perf` to sample a running process

This is great for recording flame graph data from an already running process that you don't want to interrupt. Imagine a production process with a hard to reproduce issue.

```
perf record -F99 -p `pgrep -n node` -g -- sleep 3
```

Wait, what is that `sleep 3` for? It's there to keep the perf running - despite `-p` option pointing to a different pid, the command needs to be executed on a process and end with it. perf runs for the life of the command you pass to it, whether or not you're actually profiling that command. `sleep 3` ensures that perf runs for 3 seconds.

Why is `-F` (profiling frequency) set to 99? It's a reasonable default. You can adjust if you want. `-F99` tells perf to take 99 samples per second, for more precision increase the value. Lower values should produce less output with less precise results. The precision you need depends on how long your CPU intensive functions really run. If you're looking for the reason for a noticeable slowdown, 99 frames per second should be more than enough.

After you get that 3 second perf record, proceed with generating the flame graph with the last two steps from above.

### Filtering out Node.js internal functions

Usually, you just want to look at the performance of your calls, so filtering out Node.js and V8 internal functions can make the graph much easier to read. You can clean up your perf file with:

```
sed -i -r \
  -e "/( __libc_start| LazyCompile | v8::internal::| Builtin:| Stub:| LoadIC:|\[unknown\]| LoadPolymorphicIC:)/d" \
  -e 's/ LazyCompile:[*~]?/ /' \
  perfs.out
```

If you read your flame graph and it seems odd, as if something is missing in the key function taking up most time, try generating your flame graph without the filters - maybe you got a rare case of an issue with Node.js itself.

### Node.js's profiling options

`--perf-basic-prof-only-functions` and `--perf-basic-prof` are the two that are useful for debugging your JavaScript code. Other options are used for profiling Node.js itself, which is outside the scope of this guide.

`--perf-basic-prof-only-functions` produces less output, so it's the option with the least overhead.

### Why do I need them at all?

Well, without these options, you'll still get a flame graph, but with most bars labeled `v8::Function::Call`.

## `perf` output issues

### Node.js 8.x V8 pipeline changes

Node.js 8.x and above ships with new optimizations to the JavaScript compilation pipeline in the V8 engine which makes function names/references unreachable for perf sometimes. (It's called Turbofan)

The result is you might not get your function names right in the flame graph.

You'll notice `ByteCodeHandler:` where you'd expect function names.

0x has some mitigations for that built in.

For details see:

- https://github.com/nodejs/benchmarking/issues/168
- https://github.com/nodejs/diagnostics/issues/148#issuecomment-369348961

### Node.js 10+

Node.js 10.x addresses the issue with Turbofan using the `--interpreted-frames-native-stack` flag.

Run `node --interpreted-frames-native-stack --perf-basic-prof-only-functions` to get function names in the flame graph regardless of which pipeline V8 used to compile your JavaScript.

### Broken labels in the flame graph

If you're seeing labels looking like this

```
node`_ZN2v88internal11interpreter17BytecodeGenerator15VisitStatementsEPNS0_8ZoneListIPNS0_9StatementEEE
```

it means the Linux perf you're using was not compiled with demangle support, see https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1396654 for example

## Examples

Practice capturing flame graphs yourself with a flame graph exercise!

# Test Runner

## introduction

# Discovering Node.js's test runner

In this series of resources, we will discover how to use Node.js's test runner to test our code.

## What is a test runner?

A test runner is a tool that allows you to run tests on your code. It will execute your tests and give you feedback on whether they pass or fail. It can also provide you with additional information such as code coverage.

There are many test runners available for Node.js, but we will focus on the built-in test runner that comes with Node.js. It's cool because you don't need to install any additional dependencies to use it.

## Why test your code?

Testing your code is important because it allows you to verify that your code works as expected. It helps you catch bugs early in the development process and ensures that your code is reliable and maintainable.

## Running tests with Node.js

To run tests with Node.js, we recommend you to read the test runner documentation.

# Using Node.js's test runner

Node.js has a flexible and robust built-in test runner. This guide will show you how to set up and use it.

```
example/
  ├ …
  ├ src/
    ├ app/…
    └ sw/…
  └ test/
    ├ globals/
      ├ …
      ├ IndexedDb.js
      └ ServiceWorkerGlobalScope.js
    ├ setup.mjs
    ├ setup.units.mjs
    └ setup.ui.mjs
```

```
npm init -y
npm install --save-dev concurrently
```

```
{
  "name": "example",
  "scripts": {
    "test": "concurrently --kill-others-on-fail --prefix none npm:test:*",
    "test:sw": "node --import ./test/setup.sw.mjs --test './src/sw/**/*.spec.*'",
    "test:units": "node --import ./test/setup.units.mjs --test './src/app/**/*.spec.*'",
    "test:ui": "node --import ./test/setup.ui.mjs --test './src/app/**/*.test.*'"
  }
}
```

> **Note**: globs require node v21+, and the globs must themselves be wrapped in quotes (without, you'll get different behaviour than expected, wherein it may first appear to be working but isn't).

There are some things you always want, so put them in a base setup file like the following. This file will get imported by other, more bespoke setups.

## General setup

▶ `test/setup.mjs`

Then for each setup, create a dedicated `setup` file (ensuring the base `setup.mjs` file is imported within each). There are a number of reasons to isolate the setups, but the most obvious reason is YAGNI + performance: much of what you may be setting up are environment-specific mocks/stubs, which can be quite expensive and will slow down test runs. You want to avoid those costs (literal money you pay to CI, time waiting for tests to finish, etc) when you don't need them.

Each example below was taken from real-world projects; they may not be appropriate/applicable to yours, but each demonstrate general concepts that are broadly applicable.

## ServiceWorker tests

`ServiceWorkerGlobalScope` contains very specific APIs that don't exist in other environments, and some of its APIs are seemingly similar to others (ex `fetch`) but have augmented behaviour. You do not want these to spill into unrelated tests.

▶ `test/setup.sw.mjs`

```
import assert from 'node:assert/strict';
import { describe, mock, it } from 'node:test';

import { onActivate } from './onActivate.js';

describe('ServiceWorker::onActivate()', () => {
  const globalSelf = globalThis.self;
  const claim = mock.fn(async function mock__claim() {});
  const matchAll = mock.fn(async function mock__matchAll() {});

  class ActivateEvent extends Event {
    constructor(...args) {
      super('activate', ...args);
    }
  }

  before(() => {
    globalThis.self = {
      clients: { claim, matchAll },
    };
  });
  after(() => {
    global.self = globalSelf;
  });

  it('should claim all clients', async () => {
    await onActivate(new ActivateEvent());

    assert.equal(claim.mock.callCount(), 1);
    assert.equal(matchAll.mock.callCount(), 1);
  });
});
```

## Snapshot tests

These were popularised by Jest; now, many libraries implement such functionality, including Node.js as of v22.3.0. There are several use-cases such as verifying component rendering output and Infrastructure as Code config. The concept is the same regardless of use-case.

There is no specific configuration *required* except enabling the feature via `--experimental-test-snapshots`. But to demonstrate the optional configuration, you would probably add something like the following to one of your existing test config files.

▶ `test/setup.ui.mjs`

The example below demonstrates snapshot testing with testing library for UI components; note the two different ways of accessing `assert.snapshot`):

```
import { describe, it } from 'node:test';

import { prettyDOM } from '@testing-library/dom';
import { render } from '@testing-library/react'; // Any framework (ex svelte)

import { SomeComponent } from './SomeComponent.jsx';


describe('<SomeComponent>', () => {
  // For people preferring "fat-arrow" syntax, the following is probably better for consistency
  it('should render defaults when no props are provided', (t) => {
    const component = render(<SomeComponent />).container.firstChild;

    t.assert.snapshot(prettyDOM(component));
  });

  it('should consume `foo` when provided', function() {
    const component = render(<SomeComponent foo="bar" />).container.firstChild;

    this.assert.snapshot(prettyDOM(component));
    // `this` works only when `function` is used (not "fat arrow").
  });
});
```

> ⚠️ `assert.snapshot` comes from the test's context (`t` or `this`), **not** `node:assert`. This is necessary because the test context has access to scope that is impossible for `node:assert` (you would have to manually provide it every time `assert.snapshot` is used, like `snapshot(this, value)`, which would be rather tedious).

## Unit tests

Unit tests are the simplest tests and generally require relatively nothing special. The vast majority of your tests will likely be unit tests, so it is important to keep this setup minimal because a small decrease to setup performance will magnify and cascade.

▶ `test/setup.units.mjs`

```
import assert from 'node:assert/strict';
import { describe, it } from 'node:test';

import { Cat } from './Cat.js';
import { Fish } from './Fish.js';
import { Plastic } from './Plastic.js';

describe('Cat', () => {
  it('should eat fish', () => {
    const cat = new Cat();
    const fish = new Fish();

    assert.doesNotThrow(() => cat.eat(fish));
  });

  it('should NOT eat plastic', () => {
    const cat = new Cat();
    const plastic = new Plastic();

    assert.throws(() => cat.eat(plastic));
  });
});
```

## User Interface tests

UI tests generally require a DOM, and possibly other browser-specific APIs (such as `IndexedDb` used below). These tend to be very complicated and expensive to setup.

▶ `test/setup.ui.mjs`

You can have 2 different levels of UI tests: a unit-like (wherein externals & dependencies are mocked) and a more end-to-end (where only externals like IndexedDb are mocked but the rest of the chain is real). The former is generally the purer option, and the latter is generally deferred to a fully end-to-end automated usability test via something like Playwright or Puppeteer. Below is an example of the former.

```
import { before, describe, mock, it } from 'node:test';

import { screen } from '@testing-library/dom';
import { render } from '@testing-library/react'; // Any framework (ex svelte)

// ⚠ Note that SomeOtherComponent is NOT a static import;
// this is necessary in order to facilitate mocking its own imports.


describe('<SomeOtherComponent>', () => {
  let SomeOtherComponent;
  let calcSomeValue;

  before(async () => {
    // ⚠ Sequence matters: the mock must be set up BEFORE its consumer is imported.

    // Requires the `--experimental-test-module-mocks` be set.
    calcSomeValue = mock.module('./calcSomeValue.js', { calcSomeValue: mock.fn() });

    ({ SomeOtherComponent } = await import('./SomeOtherComponent.jsx'));
  });

  describe('when calcSomeValue fails', () => {
    // This you would not want to handle with a snapshot because that would be brittle:
    // When inconsequential updates are made to the error message,
    // the snapshot test would erroneously fail
    // (and the snapshot would need to be updated for no real value).

    it('should fail gracefully by displaying a pretty error', () => {
      calcSomeValue.mockImplementation(function mock__calcSomeValue() { return null });

      render(<SomeOtherComponent>);

      const errorMessage = screen.queryByText('unable');

      assert.ok(errorMessage);
    });
  });
});
```

## mocking

# Mocking in tests

Mocking is a means of creating a facsimile, a puppet. This is generally done in a `when 'a', do 'b'` manner of puppeteering. The idea is to limit the number of moving pieces and control things that "don't matter". "mocks" and "stubs" are technically different kinds of "test doubles". For the curious mind, a stub is a replacement that does nothing (a no-op) but track its invocation. A mock is a stub that also has a fake implementation (the `when 'a', do 'b'`). Within this doc, the difference is unimportant, and stubs are referred to as mocks.

Tests should be deterministic: runnable in any order, any number of times, and always produce the same result. Proper setup and mocking make this possible.

Node.js provides many ways to mock various pieces of code.

This articles deals with the following types of tests:

| type | description | example | mock candidates | | :- | :- | :- | :- | | unit | the smallest bit of code you can isolate | `const sum = (a, b) => a + b` | own code, external code, external system | | component | a unit + dependencies | `const arithmetic = (op = sum, a, b) => ops[op](a, b)` | external code, external system | | integration | components fitting together | - | external code, external system | | end-to-end (e2e) | app + external data stores, delivery, etc | A fake user (ex a Playwright agent) literally using an app connected to real external systems. | none (do not mock) |

There are different schools of thought about when to mock and when not to mock, the broad strokes of which are outlined below.

## When and not to mock

There are 3 main mock candidates:

- Own code
- External code
- External system

### Own code

This is what your project controls.

```
import foo from './foo.mjs';

export function main() {
  const f = foo();
}
```

Here, `foo` is an "own code" dependency of `main`.

#### Why

For a true unit test of `main`, `foo` should be mocked: you're testing that `main` works, not that `main` + `foo` work (that's a different test).

#### Why not

Mocking `foo` can be more trouble than worth, especially when `foo` is simple, well-tested, and rarely updated.

Not mocking `foo` can be better because it's more authentic and increases coverage of `foo` (because `main`'s tests will also verify `foo`). This can, however, create noise: when `foo` breaks, a bunch of other tests will also break, so tracking down the problem is more tedious: if only the 1 test for the item ultimately responsible for the issue is failing, that's very easy to spot; whereas 100 tests failing creates a needle-in-a-haystack to find the real problem.

### External code

This is what your project does not control.

```
import bar from 'bar';

export function main() {
  const f = bar();
}
```

Here, `bar` is an external package, e.g. an npm dependency.

Uncontroversially, for unit tests, this should always be mocked. For component and integration tests, whether to mock depends on what this is.

**Why**

Verifying that code that your project does not maintain works is not the goal of a unit test (and that code should have its own tests).

**Why not**

Sometimes, it's just not realistic to mock. For example, you would almost never mock a large framework such as react or angular (the medicine would be worse than the ailment).

### External system

These are things like databases, environments (Chromium or Firefox for a web app, an operating system for a node app, etc), file systems, memory store, etc.

Ideally, mocking these would not be necessary. Aside from somehow creating isolated copies for each case (usually very impractical due to cost, additional execution time, etc), the next best option is to mock. Without mocking, tests sabotage each other:

```
import { db } from 'db';

export function read(key, all = false) {
  validate(key, val);

  if (all) return db.getAll(key);

  return db.getOne(key);
}

export function save(key, val) {
  validate(key, val);

  return db.upsert(key, val);
}
```

```
import assert from 'node:assert/strict';
import { describe, it } from 'node:test';

import { db } from 'db';

import { save } from './storage.mjs';

describe('storage', { concurrency: true }, () => {
  it('should retrieve the requested item', async () => {
    await db.upsert('good', 'item'); // give it something to read
    await db.upsert('erroneous', 'item'); // give it a chance to fail

    const results = await read('a', true);

    assert.equal(results.length, 1); // ensure read did not retrieve erroneous item

    assert.deepEqual(results[0], { key: 'good', val: 'item' });
  });

  it('should save the new item', async () => {
    const id = await save('good', 'item');

    assert.ok(id);

    const items = await db.getAll();

    assert.equal(items.length, 1); // ensure save did not create duplicates

    assert.deepEqual(items[0], { key: 'good', val: 'item' });
  });
});
```

In the above, the first and second cases (the `it()` statements) can sabotage each other because they are run concurrently and mutate the same store (a race condition): `save()`'s insertion can cause the otherwise valid `read()`'s test to fail its assertion on items found (and `read()`'s can do the same thing to `save()`'s).

## What to mock

### Modules + units

This leverages [`mock`] from the Node.js test runner.

```
import assert from 'node:assert/strict';
import { before, describe, it, mock } from 'node:test';


describe('foo', { concurrency: true }, () => {
  let barMock = mock.fn();
  let foo;

  before(async () => {
    const barNamedExports = await import('./bar.mjs')
      // discard the original default export
      .then(({ default, ...rest }) => rest);

    // It's usually not necessary to manually call restore() after each
    // nor reset() after all (node does this automatically).
    mock.module('./bar.mjs', {
      defaultExport: barMock
      // Keep the other exports that you don't want to mock.
      namedExports: barNamedExports,
    });

    // This MUST be a dynamic import because that is the only way to ensure the
    // import starts after the mock has been set up.
    ({ foo } = await import('./foo.mjs'));
  });

  it('should do the thing', () => {
    barMock.mockImplementationOnce(function bar_mock() {/* … */});

    assert.equal(foo(), 42);
  });
});
```

### APIs

A little-known fact is that there is a builtin way to mock `fetch`. [`undici`] is the Node.js implementation of `fetch`. It's shipped with `node`, but not currently exposed by `node` itself, so it must be installed (ex `npm install undici`).

```
import assert from 'node:assert/strict';
import { beforeEach, describe, it } from 'node:test';
import { MockAgent, setGlobalDispatcher } from 'undici';

import endpoints from './endpoints.mjs';

describe('endpoints', { concurrency: true }, () => {
  let agent;
  beforeEach(() => {
    agent = new MockAgent();
    setGlobalDispatcher(agent);
  });

  it('should retrieve data', async () => {
    const endpoint = 'foo';
    const code = 200;
    const data = {
      key: 'good',
      val: 'item',
    };
    agent
      .get('example.com')
      .intercept({
        path: endpoint,
        method: 'GET',
      })
      .reply(code, data);

    assert.deepEqual(await endpoints.get(endpoint), {
      code,
      data,
    });
  });

  it('should save data', async () => {
    const endpoint = 'foo/1';
    const code = 201;
    const data = {
      key: 'good',
      val: 'item',
    };

    agent
      .get('example.com')
      .intercept({
        path: endpoint,
        method: 'PUT',
      })
      .reply(code, data);

    assert.deepEqual(await endpoints.save(endpoint), {
      code,
      data,
    });
  });
});
```

### Time

Like Doctor Strange, you too can control time. You would usually do this just for convenience to avoid artificially protracted test runs (do you really want to wait 3 minutes for that `setTimeout()` to trigger?). You may also want to travel through time. This leverages `mock.timers` from the Node.js test runner.

Note the use of time-zone here (`Z` in the time-stamps). Neglecting to include a consistent time-zone will likely lead to unexpected restults.

```
import assert from 'node:assert/strict';
import { describe, it, mock } from 'node:test';

import ago from './ago.mjs';

describe('whatever', { concurrency: true }, () => {
  it('should choose "minutes" when that\'s the closet unit', () => {
    mock.timers.enable({ now: new Date('2000-01-01T00:02:02Z') });

    const t = ago('1999-12-01T23:59:59Z');

    assert.equal(t, '2 minutes ago');
  });
});
```

This is especially useful when comparing against a static fixture (that is checked into a repository), such as in snapshot testing.