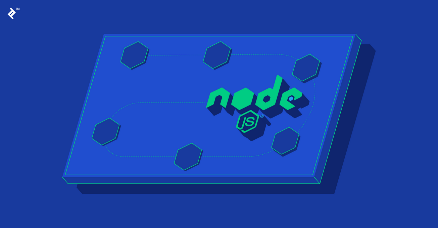
**Why the Hell Would I Use Node.js? A Case-by-case Tutorial**

Node.js shines in real-time web apps that employ push technology over WebSocket. Node’s real-time, two-way connections—where the client and server can each initiate communication—enable the freer exchange of data.

authors are vetted experts in their fields and write on topics in which they have demonstrated experience. All of our content is peer reviewed and validated by Toptal experts in the same field.





[**View full profile**](https://www.toptal.com/resume/tomislav-capan)

By [**Tomislav Capan**](https://www.toptal.com/resume/tomislav-capan)

**Verified Expert**in Engineering

Tomislav is an AWS Certified Solution Architect, developer, and technical consultant with more than 10 years of experience. Tomislav has a master’s degree in computing.

EXPERTISE

[JavaScript](https://www.toptal.com/javascript)[Express.js](https://www.toptal.com/express-js)[Socket.IO](https://www.toptal.com/socket-io)

PREVIOUSLY AT

SHARE THIS ARTICLE

Read the [Spanish](https://www.toptal.com/nodejs/por-que-demonios-usaria-node-js-un-tutorial-caso-por-caso) version of this article translated by **Isabella Rolz**.

**Editor’s note: The English version of this article was updated on 10/03/2022 by our editorial team. It has been modified to include recent sources and to align with our current editorial standards.**

JavaScript’s popularity has brought with it a lot of changes. The things we do on the web nowadays were hard to imagine just several years ago.

Before we dig into [Node.js (“Node”) solutions](https://www.toptal.com/nodejs), consider that applying [JavaScript across the stack](https://www.toptal.com/javascript/guide-to-full-stack-javascript-initjs) to unify the language and data format (JSON), would facilitate the optimal reuse developer resources. As this is more a benefit of JavaScript than Node.js specifically, we won’t elaborate further.

With all of its advantages, Node.js plays a critical role in the technology stack of many high-profile companies who depend on its unique benefits. This Node.js tutorial addresses how to realize these advantages and why you might—or might not—use Node.js.

What Is Node.js?

Node.js is composed of Google’s V8 JavaScript engine, the libUV platform abstraction layer, and a core library that is written in JavaScript. Additionally, Node.js is based on the open web stack (HTML, CSS, and JS), and operates over the standard port 80.

Node.js provides developers a comprehensive tool for working in the non-blocking, event-driven I/O paradigm. Ryan Dahl, the creator of Node.js was “inspired by applications like Gmail” and—in creating Node.js—aimed to create real-time websites with push capability.

After over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with real-time, two-way connections.

Why Use Node.js?

Node.js shines in real-time web applications employing push technology over WebSocket. After over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with real-time, two-way connections, where both the client and server can initiate communication, allowing them to exchange data more freely. This is in stark contrast to the typical web response paradigm, where the client always initiates communication.

One might argue that we’ve had this technology for years in the form of Flash and Java Applets. In reality, however, those were just sandboxed environments that used the web as a transport protocol to be delivered to the client. Plus, Flash and Java Applets were run in isolation and often operated over nonstandard ports, which may have required extra permissions.

How Does Node.js Work?

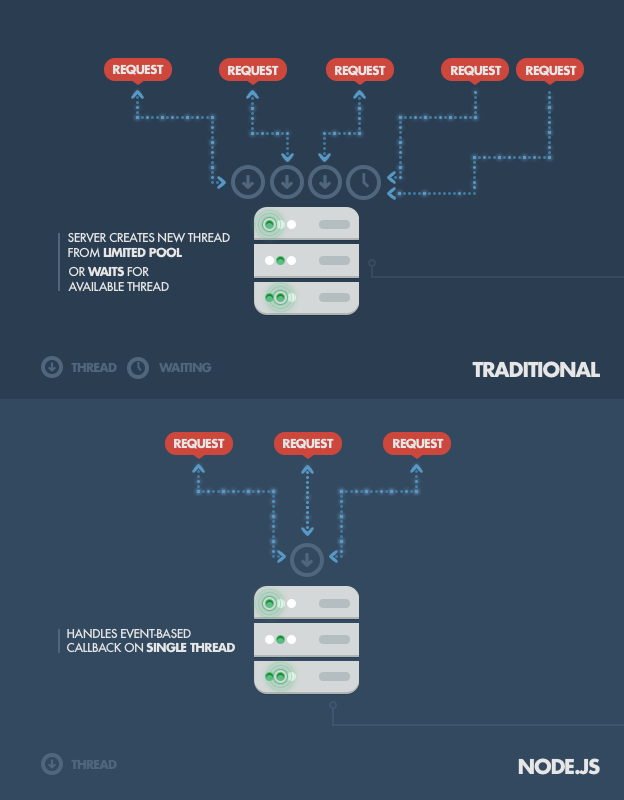
Node really shines in building fast, scalable network applications. This is due to its capability of handling a huge number of simultaneous connections with high throughput.

Node.js uses non-blocking, event-driven I/O to remain lightweight and efficient in the face of data-intensive real-time applications that run across distributed devices.

Node.js is a platform that fills a particular need, and understanding this is absolutely essential. For example, you wouldn’t use Node.js to perform CPU-intensive operations. Nearly all of Node’s advantages are annulled if it’s used for heavy computation.

Node.js is a platform that fills a particular need. It is *not* a silver bullet, or a platform that will dominate the web development world.

How Node.js works under the hood is interesting. Compared to traditional web-serving techniques where each connection (request) spawns a new thread (taking up system RAM and eventually maxing out at the amount of RAM available), Node.js operates on a single thread, using nonblocking I/O calls. This allows Node to support tens of thousands of concurrent connections held in the event loop.



Per Michael Abernethy’s 2011 article [“Just what is Node.js?”](https://kyulingcompany.files.wordpress.com/2012/06/os-nodejs-pdf.pdf), take a thread with an accompanying 2 MB of memory, running on a system with 8 GB of RAM, and providing a theoretical maximum of 4,000 concurrent connections. Add to this the cost of [context-switching between threads](http://www.slideshare.net/marcusf/nonblocking-io-event-loops-and-nodejs), and you get a common scenario in traditional web-serving techniques. Node.js avoids all this, achieving high scalability levels.

There is, of course, the question of sharing a single thread among all client requests, a potential pitfall of [writing Node.js applications](https://www.toptal.com/json/jwt-nodejs-security).

First, heavy computation could choke up Node’s single thread and cause problems for all clients, as incoming requests are blocked until said computation is completed.

Second, developers need to be vigilant and prevent exceptions from bubbling up to the core (top) Node.js event loop, as this would cause the Node.js instance to terminate, effectively crashing the program.

To prevent the flow of exceptions, we pass errors back to the caller as callback parameters (instead of “throwing,” as we do in some other environments). If an unhandled exception manages to surface, we can use the [Forever](https://www.npmjs.com/package/forever) module or external tools such as [upstart and monit](http://www.thegeekstuff.com/2016/09/deploy-nodejs/) and [just upstart](https://jalada.co.uk/simple-upstart-script-to-keep-a-node-process-alive/) to monitor the Node.js process and perform the necessary [recovery of a crashed instance](https://shapeshed.com/uncaught-exceptions-in-node/). Note that these tools do not address the recovery of the current state of the user session.

npm: The Node Package Manager

Built-in support for [package management using npm](https://www.toptal.com/javascript/a-guide-to-npm-the-node-package-manager) is included in every Node.js installation. The idea behind npm modules is similar to that of [Ruby Gems](https://rubygems.org/): It is a set of publicly available, reusable components, easily installed via an online repository, with version and dependency management.

[npm Inc.](https://npmjs.com/) shares a list of packaged modules that are also accessible via its npm CLI tool. The module ecosystem is open to all to publish their own module, which would be added to the npm repository.

Some useful npm modules include:

|  |  |
| --- | --- |
| [express](http://expressjs.com/), Express.js, or simply Express | A Sinatra-inspired web development framework for Node.js, and the *de facto* standard for the majority of Node.js applications. |
| [hapi](https://mamaya-spark.gitbooks.io/hapi-guide/content/) | A modular and simple-to-use configuration-centric framework for building web and services applications. |
| [connect](https://github.com/senchalabs/connect#readme) | An extensible HTTP server framework for Node.js, providing a collection of high performance plugins known as *middleware*; serves as a base foundation for Express. |
| [socket.io](http://socket.io/) and [sockjs](https://github.com/sockjs" \t "_blank) | A server-side component of two common WebSocket components. |
| [pug](https://github.com/pugjs/pug) (formerly Jade) | A templating engine inspired by HAML, a default in Express.js. |
| [mongodb](https://www.npmjs.com/package/mongodb) and [mongojs](https://github.com/gett/mongojs" \t "_blank) | MongoDB wrappers to provide the API for MongoDB object databases in Node.js. |
| [redis](https://github.com/redis/node-redis) | The Redis client library. |
| [lodash, underscore, lazy.js](https://npmcompare.com/compare/lazy.js,lodash,underscore) | The JavaScript utility belt. Underscore initiated the game but got overthrown by one of its two counterparts, mainly due to lazy.js' [better performance](http://philosopherdeveloper.com/posts/introducing-lazy-js.html) and modular implementation. |
| [forever](https://npmjs.org/package/forever) | A utility for ensuring that a given node script runs continuously; keeps your Node.js process up in production in the face of any unexpected failures. |
| [bluebird](https://npmmirror.com/package/bluebird/v/3.3.1) | A full-featured Promises/A+ implementation with exceptionally good performance. |
| [moment.js](https://momentjs.com/) | A JavaScript date library for parsing, validating, manipulating, and formatting dates. |

Where to Use Node.js

Chat

Chat is a typical real-time, multi-user application—from IRC (back in the day)—to modern implementations in Node.js with WebSocket.

The chat application is lightweight, high traffic, and data intensive (but low processing /computation). It runs across distributed devices and is the sweet-spot example for Node.js.

Simple, yet covering most of the paradigms you’ll ever use in a typical Node.js application, chat is a great use case for learning.

Let’s depict how chat works: Say we have a single chat room where users can exchange messages in one-to-many (actually all) fashion. Let’s also say there are three users connected to our message board.

On the server side, a simple Express.js application implements:

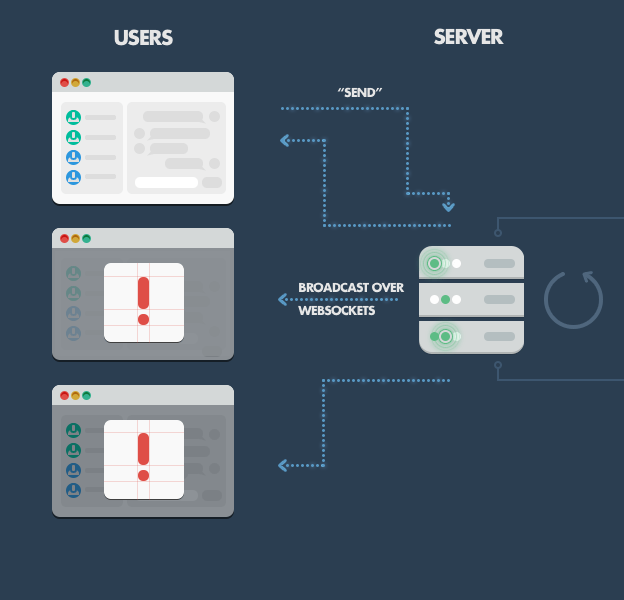
1. A GET / request handler which serves the webpage containing a message board and a ‘Send’ button to initialize new message input, and
2. A WebSocket server that listens for new messages emitted by WebSocket clients.

On the client side, we have an HTML page with a couple of handlers set up:

1. A handler for the ‘Send’ button click event, which picks up the input message and sends it down the WebSocket.
2. A handler that listens for new incoming messages on the WebSocket client (i.e., user-originated messages that the server wants the client to display).

When a client posts a message, here’s what happens:

1. The browser catches the ‘Send’ button click through a JavaScript handler. It picks up the value from the input field (i.e., the message text), and emits a WebSocket message using the WebSocket client connected to our server (initialized on web page initialization).
2. The server-side component of the WebSocket connection receives the message and forwards it to all other connected clients, using the broadcast method.
3. All clients receive the new message as a push message, via a WebSocket client-side component running within the web page. The clients then pick up the message content and update the web page in place, by appending the new message to the board.



Here is a [simple example of real-time chat with NodeJS, Socket.io and ExpressJS](https://code.tutsplus.com/tutorials/real-time-chat-with-nodejs-socketio-and-expressjs--net-31708).

For a more powerful solution, you might use a simple cache based on the Redis store. Or in an even more advanced solution, use a message queue to handle the routing of messages to clients, and a more robust delivery mechanism. The queue may cover for temporary connection losses or store messages for registered clients while they’re offline.

Regardless of the solution you choose, Node.js operates under the same basic principles: reacting to events, handling many concurrent connections, and maintaining fluidity in the user experience.

API on Top of an Object DB

Node.js is a natural fit for exposing the data from object DBs (e.g., MongoDB). JSON-stored data allows Node.js to function without impedance mismatch and data conversion.

For instance, if you’re using [Rails](https://www.toptal.com/ruby-on-rails/after-two-decades-of-programming-i-use-rails/), you would convert from JSON to binary models, then expose them back as JSON over the HTTP when the data is consumed by Backbone.js, Angular.js, etc.—or even plain jQuery AJAX calls. With Node.js, you can expose JSON objects with a [REST API](https://www.redhat.com/en/topics/api/what-is-a-rest-api) for the client to consume.

If you are using MongoDB, you needn’t worry about converting between JSON and whatever else when reading or writing from the database. Thus, you can avoid the need for multiple conversions by using a uniform data serialization format across the client, server, and database.

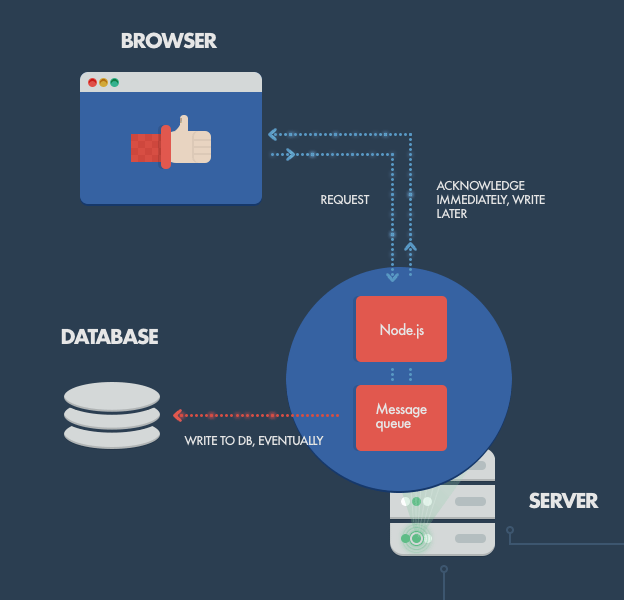
Queued Inputs

Node allows you the flexibility to push database write-offs to the side. But there are even more reasons to use Node.js.

If you’re receiving a large amount of concurrent data, your database can become a bottleneck. Node.js can easily handle the concurrent connections. Because—in this case—database access is a blocking operation, we run into trouble. The solution is to acknowledge the client’s behavior before the data is truly written to the database.

This approach enables the system to maintain responsiveness under a heavy load. It is particularly useful when the client doesn’t require firm confirmation of a successful data write, when logging or writing user-tracking data, processed in batches, to be used at a later time, or for [operations that don’t need to be reflected instantly](https://www.allthingsdistributed.com/2007/12/eventually_consistent.html), like updating a Facebook “likes” count.

Data is queued through some kind of caching or message queuing infrastructure like [RabbitMQ](https://www.rabbitmq.com/) or [ZeroMQ](https://zeromq.org/" \t "_blank). It is then digested by a separate database batch-write process or computation-intensive processing backend service, written in a better-performing platform for such a task.



In short: With Node you can push the database writes off to the side to deal with later.

Data Streaming

Why not use Node.js in data streaming? In more traditional web platforms, HTTP requests and responses are treated like isolated events—although they’re actually streams. We can use this observation to build some cool Node.js features.

For example, we can process files while they’re still being uploaded. As data comes in through a stream, we can process it in parallel during that upload process. This is true for [real-time audio or video encoding](https://transloadit.com/blog/2010/12/realtime-encoding-over-150x-faster/) and proxying between various data sources.

Proxy

Node.js is easily employed as a server-side proxy, where it can handle a large amount of simultaneous connections in a nonblocking manner. It’s useful for proxying different services with varying response times, or collecting data from multiple source points.

As an example, let’s consider a server-side application communicating with third-party resources, pulling in data from different sources, or storing assets (like images and videos) to third-party cloud services.

Using Node in place of a dedicated proxy server might be helpful if your proxying infrastructure is non-existent, or if you need a solution for local development. By this, I mean that you could build a client-side app with a Node.js development server for assets and proxying/stubbing API requests. In production, you’d handle such interactions with a dedicated proxy service (like nginx or HAProxy).

Brokerage/Stock Trader’s Dashboard

At the application level, brokers’ trading software is another example where desktop software dominates, but could be easily replaced with a real-time web solution. Brokers’ trading software tracks stock prices, performs calculations and technical analysis, and renders graphs and charts.

Why not use Node.js to write a real-time web-based solution for brokers? Then, brokers could easily switch workstations or work locations. We may soon meet our brokers on the beach in Florida or Ibiza or Bali.

Application Monitoring Dashboard

Imagine how you could grow your business if you could see what your visitors were doing in real time. With Node’s real-time, two-way sockets, you can gain this ability.

Node with WebSocket fits perfectly for tracking website visitors and visualizing their interactions in real time.

Reasons to use Node.js for a monitoring dashboard include gathering real-time stats from users, or introducing targeted interactions with your visitors by opening a communication channel at a specific point in your funnel. [CANDDi](http://canddi.com/" \t "_blank) productizes this idea.

System Monitoring Dashboard

Now, let’s visit the infrastructure side of things. Imagine, for example, a SaaS provider that wants to offer users a service monitoring page, like GitHub’s status page. With Node.js’s event-loop, we can create a powerful web-based dashboard that checks services’ statuses in an [asynchronous](https://www.toptal.com/javascript/asynchronous-javascript-async-await-tutorial) manner, pushing data to clients using WebSocket.

Both internal (intracompany) and public services’ statuses can be reported live and in real time using this technology. Push a little further and try to imagine a [network operations center (NOC)](https://en.wikipedia.org/wiki/Network_operations_center) that monitors the applications of a telecommunications operator, cloud/network/hosting provider, or some financial institution. The applications would run on the open web stack backed by Node.js and WebSocket.

Don’t try to build hard real-time systems in Node (i.e., systems requiring consistent response times). [Erlang](https://www.erlang.org/) is probably a better choice</a> for that class of application.

Where to Use Node.js, but Cautiously

Server-side Web Applications

With Node.js with Express.js, you can create classic web applications on the server side. While possible, this request-response paradigm in which Node.js would carry rendered HTML is not an ideal use case. There are arguments to be made for and against this approach. Here are some facts to consider:

Pros:

* You can significantly ease the development of an application that does not require CPU-intensive computation, by using Javascript to build it top to bottom, even down to the database level—if you use JSON storage Object DB (e.g., MongoDB).
* Crawlers receive a fully rendered HTML response, which is far more SEO friendly than, say, a Single Page Application or a WebSocket app that runs on top of Node.js.

Cons:

* Any CPU-intensive computation will block Node.js responsiveness, so a threaded platform is a better approach. Alternatively, you could try scaling out the computation.
* Using Node.js with a relational database can be painful. If you’re trying to perform relational operations, consider going with an environment such as Rails, Django, or ASP.Net MVC.

An alternative to CPU-intensive computations is to create a highly scalable MQ-backed environment with back-end processing to keep Node as a front-facing “clerk” to handle client requests asynchronously.

Where Not to Use Node.js

There are situations where Node may not be the best tool for the job.

Server-side Web Application With a Relational Database Application

Ruby on Rails was once the clear choice as a tool to access relational databases like PostgreSQL, MySQL, and Microsoft SQL Server. This was because relational DB tools for Node.js were still in their early stages while, in contrast, Rails automatically provided data access setup right out of the box, together with DB schema migrations support tools, and other Gems (pun intended). Rails and its peer frameworks have mature and proven Active Record or Data Mapper data access layer implementations.

It’s possible and not uncommon to use Node solely on the front end, while keeping your Rails back end with its easy access to a relational DB.

But things have changed. [Sequelize](http://sequelize.org/" \t "_blank), [TypeORM](https://github.com/typeorm/typeorm" \t "_blank), and [Bookshelf](https://github.com/bookshelf/bookshelf) have come a long way toward becoming mature ORM solutions. It might also be worth checking out [Join Monster](https://github.com/join-monster/join-monster) if you’re looking to generate SQL from GraphQL queries.

Heavy Server-side Computation and/or Processing

Node.js is not the best platform to handle heavy computation. No, you definitely don’t want to build a Fibonacci computation server in Node.js.

In general, any CPU-intensive operation annuls all the throughput benefits Node offers with its event-driven, nonblocking I/O model. This is because incoming requests are blocked while the thread is occupied with your number-crunching—assuming you’re trying to run computations in the same Node instance used to respond to requests.

Since Node.js is single-threaded and uses only a single CPU core, it would require considerable effort to develop a [cluster module](http://nodejs.org/api/cluster.html) in order to deliver concurrency on a multicore server. Alternatively, you can run several Node.js server instances pretty easily behind a [reverse proxy via nginx](https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/).

With clustering, you should still offload all heavy computation to background processes. Ensure that you use an appropriate environment for the background processes, and that they communicate via a message queue server like RabbitMQ.

While you may run background processes on the main server, this approach may not scale well once the load increases. You may distribute background processing services to separate worker servers without the need to configure the loads of front-facing web servers.

With Node.js—as opposed to most other platforms—you enjoy that high reqs/sec throughput we talked about, as each request is a small task that Node handles quickly and efficiently.

Why Choose Node.js?

We discussed Node.js from theory to practice, beginning with its purpose, and ending with its sweet spots and pitfalls.

Problems with Node almost always originate from the fact that *blocking operations are the root of all evil*—and 99% of Node misuses are a direct consequence.

In Node, blocking operations are the root of all evil—99% of Node misuses are a direct consequence.

If your use case does not contain CPU-intensive operations, nor accesses blocking resources, you can exploit the benefits of Node.js and enjoy fast and scalable network applications. Welcome to the real-time web.

Further Reading on the Toptal Blog:

* [The Top 10 Most Common Mistakes That Node.js Developers Make](https://www.toptal.com/nodejs/top-10-common-nodejs-developer-mistakes)
* [Creating a Secure REST API in Node.js](https://www.toptal.com/nodejs/secure-rest-api-in-nodejs)
* [Express, Koa, Meteor, Sails.js: A Node.js Framework Comparison](https://www.toptal.com/nodejs/nodejs-frameworks-comparison)
* [Is It Time to Use Node 8?](https://www.toptal.com/nodejs/is-it-time-to-use-node-8)
* [Smart Node.js Form Validation](https://www.toptal.com/nodejs/smart-node-js-form-validation)
* [How to Use JWT and Node.js for Better App Security](https://www.toptal.com/json/jwt-nodejs-security)

UNDERSTANDING THE BASICS

* What is Node.js?

Node.js is a server-side, open-source, JavaScript runtime environment. Node uses Google’s V8 engine—libUV—to deliver cross-platform compatibility and a core library. Notably, Node.js does not expose a global window object, since it does not run within a browser.

* What is Node.js used for?
* What is a web framework?
* Is Node.js a framework?
* Is Node.js a programming language?
* Why is Node.js popular?
* What is the difference between Node.js and Angular/AngularJS?
* Why is Node.js bad?

TAGS

* [JavaScript](https://www.toptal.com/developers/blog/tags/javascript)
* [I/O](https://www.toptal.com/developers/blog/tags/io)
* [Node.js](https://www.toptal.com/developers/blog/tags/node-js)

Further Reading on the Toptal Blog:

* [The Top 10 Most Common Mistakes That Node.js Developers Make](https://www.toptal.com/nodejs/top-10-common-nodejs-developer-mistakes)
* [Creating a Secure REST API in Node.js](https://www.toptal.com/nodejs/secure-rest-api-in-nodejs)
* [Express, Koa, Meteor, Sails.js: A Node.js Framework Comparison](https://www.toptal.com/nodejs/nodejs-frameworks-comparison)
* [Is It Time to Use Node 8?](https://www.toptal.com/nodejs/is-it-time-to-use-node-8)
* [Smart Node.js Form Validation](https://www.toptal.com/nodejs/smart-node-js-form-validation)
* [How to Use JWT and Node.js for Better App Security](https://www.toptal.com/json/jwt-nodejs-security)

UNDERSTANDING THE BASICS

* What is Node.js?

Node.js is a server-side, open-source, JavaScript runtime environment. Node uses Google’s V8 engine—libUV—to deliver cross-platform compatibility and a core library. Notably, Node.js does not expose a global window object, since it does not run within a browser.

* What is Node.js used for?

Because Node.js is single-threaded, we use it primarily for non-blocking, event-driven servers. We can also use Node.js for traditional websites and back-end API services, as it was designed with real-time, push-based architectures in mind.

* What is a web framework?

Web frameworks like Angular and React are libraries that help organize and generate the front-end code that runs in a web browser. Web frameworks reuse code for common operations, thereby reducing development time. Some web frameworks are full stack.

* Is Node.js a framework?

No, Node.js is an environment. Back-end frameworks run within Node.js. Popular, compatible frameworks include Express.js (also referred to as Express) for HTTP servers and Socket.IO for WebSocket servers.

* Is Node.js a programming language?

Node.js is not a programming language. The “.js” at the end of its name indicates that JavaScript is its associated programming language. Anything that can transpile to JavaScript—like TypeScript, Haxe, or CoffeeScript—can also be used with Node.js.

* Why is Node.js popular?

Aside from its being highly effective, Node.js is popular because of its huge, active, open-source, JavaScript-based ecosystem.

* What is the difference between Node.js and Angular/AngularJS?

The Node.js runtime environment executes JavaScript code on the server, whereas Angular is a JavaScript framework that is executed on the client (i.e., within a web browser).

* Why is Node.js bad?

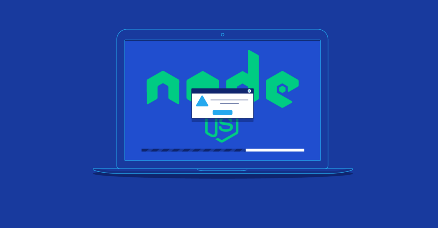
Node.js isn’t bad. Its technology is widely used for many types of servers. However, because it’s single-threaded, Node.js is not ideal for web servers that double as computational servers—such heavy computation would block the server’s responsiveness.

**The Top 10 Most Common Mistakes That Node.js Developers Make**

Node.js, the cross-platform runtime environment, has seen its fair share of praise and criticism for what it offers to developers. Often, reasons for such criticisms are rooted in how developers use/misuse the language.

This article explores ten common mistakes that Node.js developers make and how to avoid them.

authors are vetted experts in their fields and write on topics in which they have demonstrated experience. All of our content is peer reviewed and validated by Toptal experts in the same field.





[**View full profile**](https://www.toptal.com/resume/mahmud-ridwan)

By [**Mahmud Ridwan**](https://www.toptal.com/resume/mahmud-ridwan)

**Verified Expert**in Engineering

Mahmud is a software developer with many years of experience and a knack for efficiency, scalability, and stable solutions.

EXPERTISE

[Node.js](https://www.toptal.com/nodejs)[JavaScript](https://www.toptal.com/javascript)

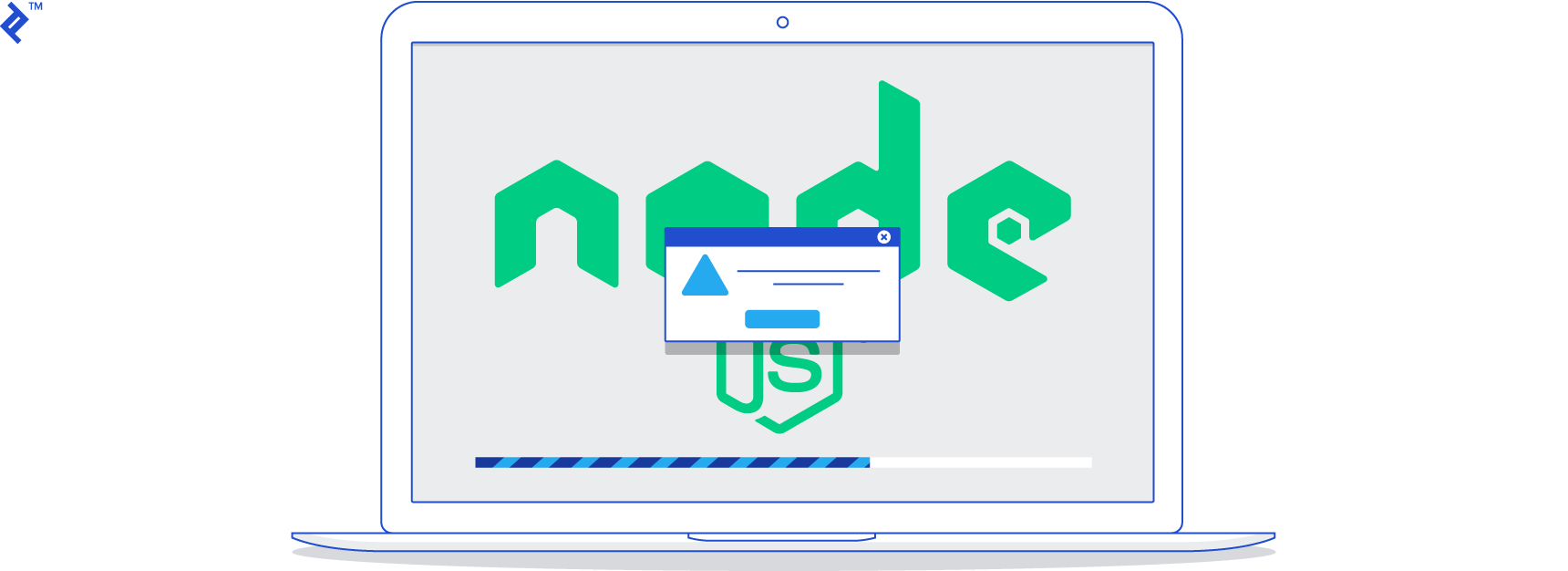
YEARS OF EXPERIENCE

**13**

SHARE THIS ARTICLE

Since the moment Node.js was unveiled to the world, it has seen a fair share of both [praise and criticism](https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js). The debate still continues, and may not end anytime soon. What we often overlook in these debates is that every programming language and platform is criticized based on certain issues, which are created by how we use the platform. Regardless of how difficult Node.js makes writing safe code, and how easy it makes writing highly concurrent code, the platform has been around for quite a while and has been used to build a huge number of robust and sophisticated web services. These web services scale well, and have proven their stability through their endurance of time on the Internet.

However, like any other platform, Node.js is vulnerable to developer problems and issues. Some of these mistakes degrade performance, while others make Node.js appear straight out unusable for whatever you are trying to achieve. In this article, we will take a look at ten common mistakes that developers new to Node.js often make, and how they can be avoided to become a [Node.js pro](https://www.toptal.com/nodejs).



Mistake #1: Blocking the event loop

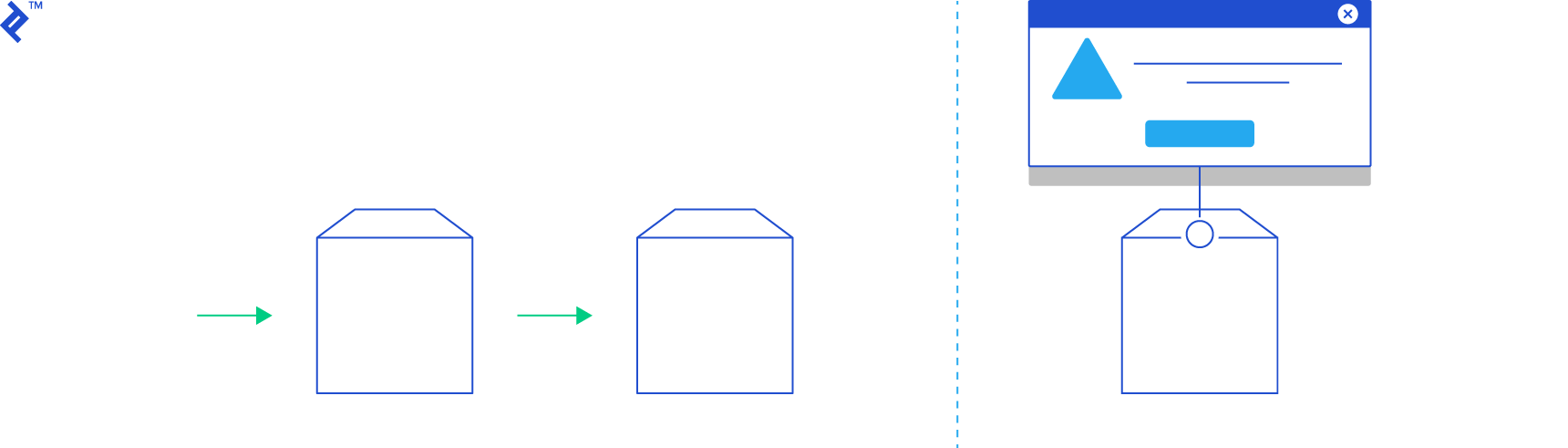
JavaScript in Node.js (just like in the browser) provides a single threaded environment. This means that no two parts of your application run in parallel; instead, concurrency is achieved through the handling of I/O bound operations asynchronously. For example, a request from Node.js to the database engine to fetch some document is what allows Node.js to focus on some other part of the application:

// Trying to fetch an user object from the database. Node.js is free to run other parts of the code from the moment this function is invoked..

db.User.get(userId, function(err, user) {

// .. until the moment the user object has been retrieved here

})



However, a piece of CPU-bound code in a Node.js instance with thousands of clients connected is all it takes to block the event loop, making all the clients wait. CPU-bound codes include attempting to sort a large array, running an extremely long loop, and so on. For example:

function sortUsersByAge(users) {

users.sort(function(a, b) {

return a.age < b.age ? -1 : 1

})

}

Invoking this “sortUsersByAge” function may be fine if run on a small “users” array, but with a large array, it will have a horrible impact on the overall performance. If this is something that absolutely must be done, and you are certain that there will be nothing else waiting on the event loop (for example, if this was part of a command-line tool that you are building with Node.js, and it wouldn’t matter if the entire thing ran synchronously), then this may not be an issue. However, in a Node.js server instance trying to serve thousands of users at a time, such a pattern can prove fatal.

If this array of users was being retrieved from the database, the ideal solution would be to fetch it already sorted directly from the database. If the event loop was being blocked by a loop written to compute the sum of a long history of financial transaction data, it could be deferred to some external worker/queue setup to avoid hogging the event loop.

As you can see, there is no silver-bullet solution to this kind of Node.js problem, rather each case needs to be addressed individually. The fundamental idea is to not do CPU intensive work within the front facing Node.js instances - the ones clients connect to concurrently.

Mistake #2: Invoking a Callback More Than Once

JavaScript has relied on callbacks since forever. In web browsers, events are handled by passing references to (often anonymous) functions that act like callbacks. In Node.js, callbacks used to be the only way asynchronous elements of your code communicated with each other - up until promises were introduced. Callbacks are still in use, and package developers still design their APIs around callbacks. One common Node.js issue related to using callbacks is calling them more than once. Typically, a function provided by a package to do something asynchronously is designed to expect a function as its last argument, which is called when the asynchronous task has been completed:

module.exports.verifyPassword = function(user, password, done) {

if(typeof password !== ‘string’) {

done(new Error(‘password should be a string’))

return

}

computeHash(password, user.passwordHashOpts, function(err, hash) {

if(err) {

done(err)

return

}

done(null, hash === user.passwordHash)

})

}

Notice how there is a return statement every time “done” is called, up until the very last time. This is because calling the callback doesn’t automatically end the execution of the current function. If the first “return” was commented out, passing a non-string password to this function will still result in “computeHash” being called. Depending on how “computeHash” deals with such a scenario, “done” may be called multiple times. Anyone using this function from elsewhere may be caught completely off guard when the callback they pass is invoked multiple times.

Being careful is all it takes to avoid this Node.js error. Some Node.js developers adopt a habit of adding a return keyword before every callback invocation:

if(err) {

return done(err)

}

In many asynchronous functions, the return value has almost no significance, so this approach often makes it easy to avoid such a problem.

Mistake #3: Deeply Nesting Callbacks

Deeply-nesting callbacks, often referred to as “callback hell”, is not a Node.js issue in itself. However, this can cause problems making code quickly spin out of control:

function handleLogin(..., done) {

db.User.get(..., function(..., user) {

if(!user) {

return done(null, ‘failed to log in’)

}

utils.verifyPassword(..., function(..., okay) {

if(okay) {

return done(null, ‘failed to log in’)

}

session.login(..., function() {

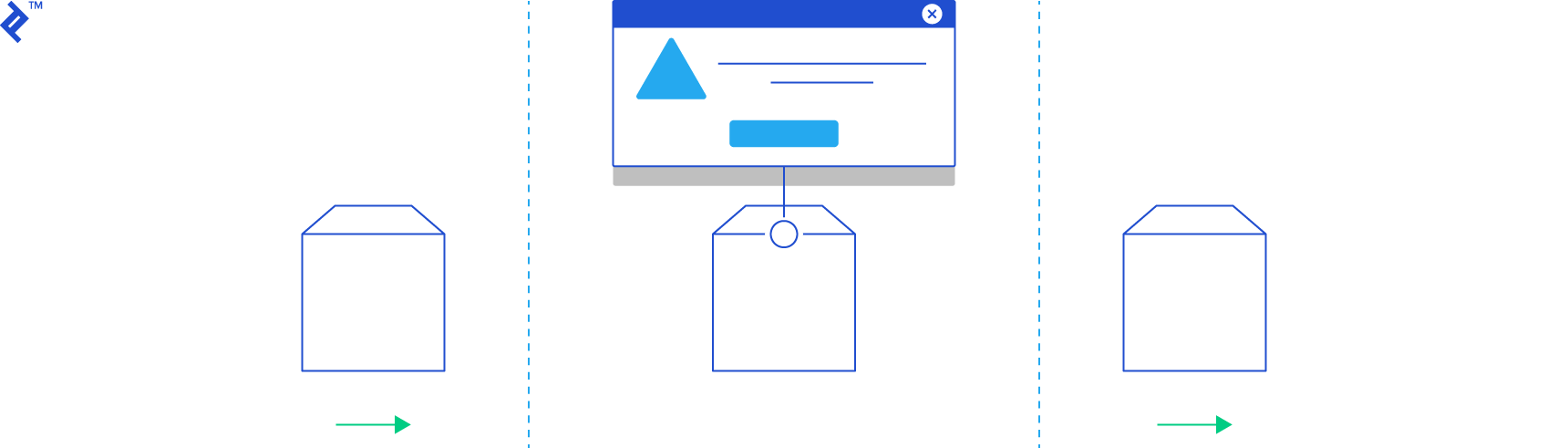
done(null, ‘logged in’)

})

})

})

}



The more complex the task, the worse this can get. By nesting callbacks in such a way, we easily end up with error-prone, hard to read, and hard to maintain code. One workaround is to declare these tasks as small functions, and then link them up. Although, one of the (arguably) cleanest solutions to this is to use a utility Node.js package that deals with [asynchronous JavaScript](https://www.toptal.com/javascript/asynchronous-javascript-programming) patterns, such as [Async.js](https://github.com/caolan/async):

function handleLogin(done) {

async.waterfall([

function(done) {

db.User.get(..., done)

},

function(user, done) {

if(!user) {

return done(null, ‘failed to log in’)

}

utils.verifyPassword(..., function(..., okay) {

done(null, user, okay)

})

},

function(user, okay, done) {

if(okay) {

return done(null, ‘failed to log in’)

}

session.login(..., function() {

done(null, ‘logged in’)

})

}

], function() {

// ...

})

}

Similar to “async.waterfall”, there are a number of other functions that Async.js provides to deal with different asynchronous patterns. For brevity, we used simpler examples here, but reality is often worse.

Mistake #4: Expecting Callbacks to Run Synchronously

Asynchronous programming with callbacks may not be something unique to JavaScript and Node.js, but they are responsible for its popularity. With other programming languages, we are accustomed to the predictable order of execution where two statements will execute one after another, unless there is a specific instruction to jump between statements. Even then, these are often limited to conditional statements, loop statements, and function invocations.

However, in JavaScript, with callbacks a particular function may not run well until the task it is waiting on is finished. The execution of the current function will run until the end without any stop:

function testTimeout() {

console.log(“Begin”)

setTimeout(function() {

console.log(“Done!”)

}, duration \* 1000)

console.log(“Waiting..”)

}

As you will notice, calling the “testTimeout” function will first print “Begin”, then print “Waiting..” followed by the the message “Done!” after about a second.

Anything that needs to happen after a callback has fired needs to be invoked from within it.

Mistake #5: Assigning to “exports”, Instead of “module.exports”

Node.js treats each file as a small isolated module. If your package has two files, perhaps “a.js” and “b.js”, then for “b.js” to access “a.js”’s functionality, “a.js” must export it by adding properties to the exports object:

// a.js

exports.verifyPassword = function(user, password, done) { ... }

When this is done, anyone requiring “a.js” will be given an object with the property function “verifyPassword”:

// b.js

require(‘a.js’) // { verifyPassword: function(user, password, done) { ... } }

However, what if we want to export this function directly, and not as the property of some object? We can overwrite exports to do this, but we must not treat it as a global variable then:

// a.js

module.exports = function(user, password, done) { ... }

Notice how we are treating “exports” as a property of the module object. The distinction here between “module.exports” and “exports” is very important, and is often a cause of frustration among new Node.js developers.

Mistake #6: Throwing Errors from Inside Callbacks

JavaScript has the notion of exceptions. Mimicking the syntax of almost all traditional languages with exception handling support, such as Java and C++, JavaScript can “throw” and catch exceptions in try-catch blocks:

function slugifyUsername(username) {

if(typeof username === ‘string’) {

throw new TypeError(‘expected a string username, got '+(typeof username))

}

// ...

}

try {

var usernameSlug = slugifyUsername(username)

} catch(e) {

console.log(‘Oh no!’)

}

However, try-catch will not behave as you might expect it to in asynchronous situations. For example, if you wanted to protect a large chunk of code with lots of asynchronous activity with one big try-catch block, it wouldn’t necessarily work:

try {

db.User.get(userId, function(err, user) {

if(err) {

throw err

}

// ...

usernameSlug = slugifyUsername(user.username)

// ...

})

} catch(e) {

console.log(‘Oh no!’)

}

If the callback to “db.User.get” fired asynchronously, the scope containing the try-catch block would have long gone out of context for it to still be able to catch those errors thrown from inside the callback.

This is how errors are handled in a different way in Node.js, and that makes it essential to follow the (err, …) pattern on all callback function arguments - the first argument of all callbacks is expected to be an error if one happens.

Mistake #7: Assuming Number to Be an Integer Datatype

Numbers in JavaScript are floating points - there is no integer data type. You wouldn’t expect this to be a problem, as numbers large enough to stress the limits of float are not encountered often. That is exactly when mistakes related to this happen. Since floating point numbers can only hold integer representations up to a certain value, exceeding that value in any calculation will immediately start messing it up. As strange as it may seem, the following evaluates to true in Node.js:

Math.pow(2, 53)+1 === Math.pow(2, 53)

Unfortunately, the quirks with numbers in JavaScript doesn’t end here. Even though Numbers are floating points, operators that work on integer data types work here as well:

5 % 2 === 1 // true

5 >> 1 === 2 // true

However, unlike arithmetic operators, bitwise operators and shift operators work only on the trailing 32 bits of such large “integer” numbers. For example, trying to shift “Math.pow(2, 53)” by 1 will always evaluate to 0. Trying to do a bitwise-or of 1 with that same large number will evaluate to 1.

Math.pow(2, 53) / 2 === Math.pow(2, 52) // true

Math.pow(2, 53) >> 1 === 0 // true

Math.pow(2, 53) | 1 === 1 // true

You may rarely need to deal with large numbers, but if you do, there are plenty of big integer libraries that implement the important mathematical operations on large precision numbers, such as [node-bigint](https://www.npmjs.com/package/bignum).

Mistake #8: Ignoring the Advantages of Streaming APIs

Let’s say we want to build a small proxy-like web server that serves responses to requests by fetching the content from another web server. As an example, we shall build a small web server that serves Gravatar images:

var http = require('http')

var crypto = require('crypto')

http.createServer()

.on('request', function(req, res) {

var email = req.url.substr(req.url.lastIndexOf('/')+1)

if(!email) {

res.writeHead(404)

return res.end()

}

var buf = new Buffer(1024\*1024)

http.get('http://www.gravatar.com/avatar/'+crypto.createHash('md5').update(email).digest('hex'), function(resp) {

var size = 0

resp.on('data', function(chunk) {

chunk.copy(buf, size)

size += chunk.length

})

.on('end', function() {

res.write(buf.slice(0, size))

res.end()

})

})

})

.listen(8080)

In this particular example of a Node.js problem, we are fetching the image from Gravatar, reading it into a Buffer, and then responding to the request. This isn’t such a bad thing to do, given that Gravatar images are not too large. However, imagine if the size of the contents we are proxying were thousands of megabytes in size. A much better approach would have been this:

http.createServer()

.on('request', function(req, res) {

var email = req.url.substr(req.url.lastIndexOf('/')+1)

if(!email) {

res.writeHead(404)

return res.end()

}

http.get('http://www.gravatar.com/avatar/'+crypto.createHash('md5').update(email).digest('hex'), function(resp) {

resp.pipe(res)

})

})

.listen(8080)

Here, we fetch the image and simply pipe the response to the client. At no point do we need to read the entire content into a buffer before serving it.

Mistake #9: Using Console.log for Debugging Purposes

In Node.js, “console.log” allows you to print almost anything to the console. Pass an object to it and it will print it as a JavaScript object literal. It accepts any arbitrary number of arguments and prints them all neatly space-separated. There are a number of reasons why a developer may feel tempted to use this to debug his code; however, it is strongly recommended that you avoid “console.log” in real code. You should avoid writing “console.log” all over the code to debug it and then commenting them out when they are no longer needed. Instead, use one of the amazing libraries that are built just for this, such as [debug](https://www.npmjs.com/package/debug).

Packages like these provide convenient ways of enabling and disabling certain debug lines when you start the application. For example, with debug it is possible to prevent any debug lines from being printed to the terminal by not setting the DEBUG environment variable. Using it is simple:

// app.js

var debug = require('debug')('app')

debug('Hello, %s!', 'world')

To enable debug lines, simply run this code with the environment variable DEBUG set to “app” or “\*”:

DEBUG=app node app.js

Mistake #10: Not Using Supervisor Programs

Regardless of whether your Node.js code is running in production or in your local development environment, a supervisor program monitor that can orchestrate your program is an extremely useful thing to have. One practice often recommended by developers designing and implementing modern applications recommends that your code should fail fast. If an unexpected error occurs, do not try to handle it, rather let your program crash and have a supervisor restart it in a few seconds. The benefits of supervisor programs are not just limited to restarting crashed programs. These tools allow you to restart the program on crash, as well as restart them when some files change. This makes developing Node.js programs a much more pleasant experience.

There is a plethora of supervisor programs available for Node.js. For example:

* [pm2](https://www.npmjs.com/package/pm2)
* [forever](https://www.npmjs.com/package/forever)
* [nodemon](https://www.npmjs.com/package/nodemon)
* [supervisor](https://www.npmjs.com/package/supervisor)

All these tools come with their pros and cons. Some of them are good for handling multiple applications on the same machine, while others are better at log management. However, if you want to get started with such a program, all of these are fair choices.

Conclusion

As you can tell, some of these Node.js problems can have devastating effects on your program. Some may be the cause of frustration while you’re trying to implement the simplest of things in Node.js. Although Node.js has made it extremely easy for newcomers to get started, it still has areas where it is just as easy to mess up. Developers from other programming languages may be able to relate to some of these issues, but these mistakes are quite common among new [Node.js developers](https://www.toptal.com/nodejs/job-description). Fortunately, they are easy to avoid. I hope this short guide will help beginners to [write better code in Node.js](https://www.toptal.com/koa), and to develop stable and efficient software for us all.

Further Reading on the Toptal Blog:

* [The 10 Most Common JavaScript Issues Developers Face](https://www.toptal.com/javascript/10-most-common-javascript-mistakes)
* [Why the Hell Would I Use Node.js? A Case-by-case Tutorial](https://www.toptal.com/javascript/why-the-hell-would-i-use-node-js)
* [Using Express.js Routes for Promise-based Error Handling](https://www.toptal.com/express-js/routes-js-promises-error-handling)
* [Writing Testable Code in JavaScript: A Brief Overview](https://www.toptal.com/javascript/writing-testable-code-in-javascript)
* [Buggy C# Code: The 10 Most Common Mistakes in C# Programming](https://www.toptal.com/c-sharp/top-10-mistakes-that-c-sharp-programmers-make)
* [Next.js vs. React: A Comparative Tutorial](https://www.toptal.com/next-js/next-js-vs-react)