**NodeJs Best Practices -** [Maram Al-Quraan](https://medium.com/@quraanmaram?source=post_page-----5793375c3ad8--------------------------------) Apr 21. 2022

To take benefit from the power of any technology you need to use it in its best practices. **NodeJs** is most popular for its **asynchronous** **event-driven**,**non-blocking I/O**processing. It gets most of this concurrency and asynchronism from Javascript’s **single-threaded** **event loop** model.

So in this blog, we will be going through some of the important best practices for programming performance using node.js

*The internet is full of tutorials, documentation, blogs, and videos covering the basics of web development. But usually, information (and importance) about best practices is something that we learn along the way, as we build more apps, as we fail and succeed along the way.*

**→ Project Structure**

* **Divide your Solution by Components:**

We should divide the entire codebase into smaller components so that each module gets its own folder, and certain that each module is kept simple and small.

Timeline

Description automatically generated with low confidence

* **Separate business logic and API routes**

Text

Description automatically generated

* **Layering Components**

Separate component code into layers: web, services, and Data Access Layer.

* **Wrap common utilities as npm packages**

Combine the same code used repeatedly multiple times into a single private package file and use that package at various places within our app.

**→**

**Logging and Error Handling Practices**

* **Use Async-Await or promises for async error handling.**
* **Handling Errors Centrally.**

Every logic that handles errors like logging performance, and sending emails regarding errors should be written in such a way so that all APIs, night jobs, and unit testing can debug messages and call this method whenever any error occurs.

* **Use only the built-in Error object.**

The Built-in error object makes a uniform approach to handling errors within our source code and other open-source JSON packages.

// throwing an Error from typical function, whether sync or async  
if(!productToAdd)  
 throw new Error('How can I add new product when no value provided?');  
  
// 'throwing' an Error from EventEmitter  
const myEmitter = new MyEmitter();  
myEmitter emit('error', new Error('whoops!'));  
  
// 'throwing' an Error from a Promise  
 const addProduct = async (productToAdd) => {  
 try {  
 const existingProduct = await DAL.getProduct (productToAdd.id);  
 if (existingProduct !== null) {  
 throw new Error('Product already exists!');  
 }  
 } catch (err) {  
 // ...  
 }  
}

* **Using a good logger to increase the error visibility.**  
  set of good logging tools like [Pino](https://github.com/pinojs/pino) or [Log4js](https://www.npmjs.com/package/log4js), will speed up error discovery and understanding. So forget about console.log
* **Catch unhandled promise rejections.**
* **Test your application.**

→ Following are the most popular testing Libraries for Node.js applications (Mocha, Jest, Jasmine, and AVA)

**→ Code Style Practices**

* **Use Linting Packages**

There are many linting tools available, **[ESLint](https://medium.com/wiki/spaces/MP/pages/2240741386" \t "_blank)** is one the most popular linting package which is used to check possible errors in code otherwise you can also [check code styles](https://www.tatvasoft.com/blog/importance-code-quality/) to meet best practices standards**.**

* **Start a Codeblock’s Curly Braces on the Same Line.** ([“Why do results vary based on curly brace placement?” (StackOverflow)](https://stackoverflow.com/questions/3641519/why-do-results-vary-based-on-curly-brace-placement)
* **Name your functions.**
* **Use naming conventions for variables, constants, functions, and classes.**
* **Add Required Modules at the Beginning**,

Avoid Inside Functions. By doing this we can easily identify dependencies of the entire file and avoid some of the potential performance issues.

* **Use Static Analysis Tools**

Tools like SonarQube and Code Climate can do a static analysis that helps to improve code quality, and performance and keeps our code manageable. We can add these tools to the CI-CD pipeline which causes build failure when they detect any areas where we can improve code quality so as to boost performance.

***More guidelines:***

[*https://developer.mozilla.org/en-US/docs/MDN/Guidelines/Code\_guidelines/JavaScript*](https://developer.mozilla.org/en-US/docs/MDN/Guidelines/Code_guidelines/JavaScript)

**→ Security Best Practices**

* **Embrace linter security rules.**

We can use linter plugins like [eslint-plugin-security](https://github.com/nodesecurity/eslint-plugin-security" \t "_blank) to catch code issues while we are coding our node.js app. Linting plugs-ins, ensures we eliminate vulnerable code while developing.

* **Inspect Vulnerable Dependencies**

We can use tools like NPM audit or [snyk.io](http://snyk.io/) to check vulnerable dependencies.

* **Limit concurrent requests**.
* **Configuration files and Environment Variables** :light\_bulb\_on: ***Config it up***

As security best practices, we should keep our app-level keys easily readable from file and environment variables. We should also keep secrets outside the committed code and make a config file hierarchy for easier accessibility. To meet all this, a perfect and flawless configuration setup is required. There are a few node.js development project structures that are available that can help to do this like RC, nconf, and config.

* **Adjust the HTTP response headers.**

Your application should be using secure headers to prevent attackers from using common attacks like cross-site scripting (XSS), clickjacking, and other malicious attacks. These can be configured easily using modules like [helmet](https://www.npmjs.com/package/helmet).

* **Validate incoming JSON schemas.**

We should validate all the request parameters and body parameters to meet the expected schema before executing the actual logic.

* **Prevent query injection vulnerabilities.**

To prevent SQL/NoSQL injection and other malicious attacks, always make use of an ORM/ODM or a database library that escapes data or supports named or indexed parameterized queries, and takes care of validating user input for expected types.

* **Hide error details from clients**

Ensure not to return the entire Error object to the client, which might contain some sensitive application details such as server file paths, third-party modules in use, and other internal workflows of the application which could be exploited by an attacker could be leaked from information found in a stack trace.

[***Read More: Common security best practices***](https://github.com/goldbergyoni/nodebestpractices/blob/master/sections/security/commonsecuritybestpractices.md)

**→ Performance Best Practices**

* **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. Here’s an example:

// This callback will run quickly for small n and more slowly for large 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}`);  
 }  
 }

* **Clean up dependencies before production.**
* **Plan for efficient caching**.
* **Lock Dependencies.**

**References:**  
1- <https://dev.to/dhiwise/10-best-practices-every-nodejs-developer-must-follow-362m>  
2- <https://github.com/goldbergyoni/nodebestpractices>  
3- <https://www.tatvasoft.com/blog/node-js-best-practices/>  
4- <https://scoutapm.com/blog/nodejs-architecture-and-12-best-practices-for-nodejs-development>  
5- <https://nodejs.org/en/docs/guides/dont-block-the-event-loop/>  
6- <https://blog.logrocket.com/the-perfect-architecture-flow-for-your-next-node-js-project/>