Coding Standards Document

[**NODE JS**](#_bzvfccrejry0) **4**

[**1. Project Structure Practices**](#_9ccu0qj2p317) **4**

[1.1 Structure your solution by components](#_q2tcokunh1q9) 4

[1.2 Layer your components](#_7f3lg714m6bs) 4

[1.3 Wrap common utilities as npm packages](#_fzt2rvwqytas) 4

[1.5 Use environment aware, secure and hierarchical config](#_9c0zg2u0ldkk) 5

[**2. Error Handling Practices**](#_ly0t5qeq4t9p) **5**

[2.1 Use Async-Await or promises for async error handling](#_3m6wmzt9obma) 5

[2.2 Use only the built-in Error object](#_i13cx0y7v5bi) 5

[2.3 Distinguish operational vs programmer errors](#_4556jjvbh7xg) 5

[2.4 Handle errors centrally, not within an Express middleware](#_cov8nc7es016) 6

[2.5 Document API errors using Swagger or GraphQL](#_3phb1og1onms) 6

[2.6 Exit the process gracefully](#_omgvqxq68j5r) 6

[2.7 Use a mature logger to increase error visibility](#_9cczddi8skob) 6

[2.8 Test error flows using your favorite test framework](#_ff5rmkob6ge2) 6

[2.9 Discover errors and downtime using APM products](#_49qcfd4wohsi) 6

[2.10 Catch unhandled promise rejections](#_u0qluujfk9jp) 7

[2.11 Fail fast, validate arguments using a dedicated library](#_k0uj20izg175) 7

[**3. Code Style Practices**](#_du3a0xvmrg57) **7**

[3.1 Use ESLint](#_oa60d8x2za1p) 7

[3.2 Node.js specific plugins](#_gkiy7iwnxaay) 7

[3.3 Start a Codeblocks Curly Braces on the Same Line](#_wfdf3qw1ewki) 7

[3.4 Separate your statements properly](#_9elb7vys1xcr) 8

[3.5 Name your functions](#_b4sdm08wbp6e) 8

[3.6 Use naming conventions for variables, constants, functions, and classes](#_rvongc842si1) 8

[3.7 Prefer const over let. Ditch the var](#_rstzps8ulo4v) 8

[3.8 Require modules first, not inside functions](#_mb4zhra3pbtk) 8

[3.9 Require modules by folders, as opposed to the files directly](#_6xugm868iz17) 9

[3.10 Use the === operator](#_p6kijaioqmrr) 9

[3.11 Use Async Await, avoid callbacks](#_k8wcqfh63dac) 9

[3.12 Use arrow function expressions (=>)](#_s20axlqk3esl) 9

[**4. Testing And Overall Quality Practices**](#_rtrbsxw4hrv4) **9**

[4.1 At the very least, write API (component) testing](#_hl63crbm6z4e) 9

[4.2 Include 3 parts in each test name](#_75riql4vgkqo) 10

[4.3 Structure tests by the AAA pattern](#_xujeeynl2hmd) 10

[4.4 Detect code issues with a linter](#_1vga29u94ec2) 10

[4.5 Avoid global test fixtures and seeds, add data per-test](#_310ty65shvmi) 10

[4.6 Constantly inspect for vulnerable dependencies](#_h57j7cixvc3) 10

[4.7 Tag your tests](#_rva3ls76luoz) 10

[4.8 Check your test coverage, it helps to identify wrong test patterns](#_okud47t5xm2) 11

[4.9 Inspect for outdated packages](#_bt86ojjnose2) 11

[4.10 Use production-like env for e2e testing](#_i5drbeplqaci) 11

[4.11 Refactor regularly using static analysis tools](#_uf7ae3wvosxe) 11

[4.12 Carefully choose your CI platform (Jenkins vs CircleCI vs Travis vs Rest of the world)](#_faw5ja3qcwnm) 12

[**5. Going To Production Practices**](#_qk28zeljq4lm) **12**

[5.1. Monitoring](#_hx5gwog7a6s2) 12

[5.2. Increase transparency using smart logging](#_b3l5x5hqt4j) 12

[5.3. Delegate anything possible (e.g. gzip, SSL) to a reverse proxy](#_3gzi7ofz0dqx) 13

[5.4. Lock dependencies](#_mqxjp65vgm3t) 13

[5.5. Guard process uptime using the right tool](#_l853dsa5p3hh) 13

[5.6. Utilize all CPU cores](#_2wkadqrtcnzg) 13

[5.7. Create a ‘maintenance endpoint’](#_iq91vq1ksk1g) 14

[5.8. Discover errors and downtime using APM products](#_rg0w7dpkd0n7) 14

[5.9. Make your code production-ready](#_8ag6tcmos28v) 14

[5.10. Measure and guard the memory usage](#_r08aotooa1z7) 15

[5.11. Get your frontend assets out of Node](#_jy6elmvrpyrw) 15

[5.12. Be stateless, kill your servers almost every day](#_p311mayu3q6y) 15

[5.13. Use tools that automatically detect vulnerabilities](#_df10h3rlawu1) 15

[5.14. Assign a transaction id to each log statement](#_oh957bplr0lz) 16

[5.15. Set NODE\_ENV](#_s82guaxk0xa) 16

[5.16. Design automated, atomic and zero-downtime deployments](#_5t9tc1cxvhwk) 16

[5.17. Use an LTS release of Node.js](#_g2b7ahenqgqk) 16

[5.18. Don’t route logs within the app](#_cb2i6u4tursm) 16

[**6. Security Best Practices**](#_fgfcffth7up4) **17**

[6.1. Embrace linter security rules](#_lw0i8pjqj4v7) 17

[6.2. Limit concurrent requests using a middleware](#_hu1hhemrpk0) 17

[6.3 Extract secrets from config files or use packages to encrypt them](#_ut56dttnh24r) 17

[6.4. Prevent query injection vulnerabilities with ORM/ODM libraries](#_2dl7r7gl9ffs) 17

[6.5. Adjust the HTTP response headers for enhanced security](#_8rugglxd1eov) 18

[6.6. Constantly and automatically inspect for vulnerable dependencies](#_h1cw3j2l1lmo) 18

[6.7. Avoid using the Node.js crypto library for handling passwords, use Bcrypt](#_6qlhgg1yer4m) 18

[6.8. Escape HTML, JS and CSS output](#_lgmdk99dywsd) 18

[6.9. Validate incoming JSON schemas](#_itr6f4flxlse) 18

[6.10. Support blacklisting JWTs](#_11xeyf2e26f9) 19

[6.11. Prevent brute-force attacks against authorization](#_vq3o7ujb3777) 19

[6.12. Run Node.js as non-root user](#_ai2s1bx1i7ky) 19

[6.13. Limit payload size using a reverse-proxy or a middleware](#_kus22lce98xy) 19

[6.14. Avoid JavaScript eval statements](#_10fsmlnxai2o) 19

[6.15. Prevent evil RegEx from overloading your single thread execution](#_gd9kem5hg31s) 20

[6.16. Avoid module loading using a variable](#_edghlerofl8c) 20

[6.17. Run unsafe code in a sandbox](#_ygm9ytxoslfi) 20

[6.18. Take extra care when working with child processes](#_8ysmkeittw9j) 21

[6.19. Hide error details from clients](#_h5qlc83kbyyx) 21

[6.20. Configure 2FA for npm or Yarn](#_b3y9bze0ak8v) 21

[6.21. Modify session middleware settings](#_945uemwgrsz) 21

[6.22. Avoid DOS attacks by explicitly setting when a process should crash](#_j7jzh47jtmz2) 22

[6.23. Prevent unsafe redirects](#_uccp5xfzrz8k) 22

[6.24. Avoid publishing secrets to the npm registry](#_332v619qoj6c) 22

[**REACT**](#_lsplh536z7gy) **22**

[Composition](#_7mmi3l8ssh4u) 22

[Common Abstraction](#_t7q8934w801a) 23

[Escape Hatches](#_f5im7ipjcqxi) 23

[Stability](#_l75y9xe7wcnt) 24

[Interoperability](#_gus00bwi79v7) 25

[Scheduling](#_6io0z7vht6jv) 25

[Developer Experience](#_95zp8rm9dehk) 26

[Debugging](#_muy76f1b3v93) 26

[Configuration](#_eb6b79ti92ui) 27

[Beyond the DOM](#_zeitmxb0sdw0) 27

[Implementation](#_lnrsx9j29kli) 28

[Optimized for Tooling](#_6wzkhs7mi6no) 28

[Dogfooding](#_rd8v5s50g3o3) 28

# NODE JS

# 1. Project Structure Practices

## 1.1 Structure your solution by components

* The worst large application pitfall is maintaining a huge code base with hundreds of dependencies – such a monolith slows down developers as they try to incorporate new features. Instead, partition your code into components, each gets its own folder or a dedicated codebase, and ensure that each unit is kept small and simple.
* The ultimate solution is to develop small software: divide the whole stack into self-contained components that don’t share files with others, each constitutes very few files (e.g. API, service, data access, test, etc.) so that it’s very easy to reason about it.

## 1.2 Layer your components

* Each component should contain ‘layers’ – a dedicated object for the web, logic, and data access code. This not only draws a clean separation of concerns but also significantly eases mocking and testing the system. Though this is a very common pattern, API developers tend to mix layers by passing the web layer objects (Express req, res) to business logic and data layers – this makes your application dependent on and accessible by Express only.

## 1.3 Wrap common utilities as npm packages

* In a large app that constitutes a large codebase, cross-cutting-concern utilities like logger, encryption and alike, should be wrapped by your own code and exposed as private npm packages. This allows sharing them among multiple codebases and projects.
* Avoid the habit of defining the entire Express app in a single huge file – separate your ‘Express’ definition to at least two files: the API declaration (app.js) and the networking concerns (WWW). For even better structure, locate your API declaration within components

## 1.5 Use environment aware, secure and hierarchical config

* A perfect and flawless configuration setup should ensure

1. keys can be read from file AND from environment variable
2. secrets are kept outside committed code
3. config is hierarchical for easier findability. There are a few packages that can help tick most of those boxes like rc, nconf and config.

# 2. Error Handling Practices

## 2.1 Use Async-Await or promises for async error handling

* Handling async errors in callback style is probably the fastest way to the pyramid of doom. Use a reputable promise library or async-await instead which enables a much more compact and familiar code syntax like try-catch.

## 2.2 Use only the built-in Error object

* Many throw errors as a string or as some custom type – this complicates the error handling logic and the interoperability between modules. Whether you reject a promise, throw an exception or emit an error – using only the built-in Error object will increase uniformity and prevent loss of information.

## 2.3 Distinguish operational vs programmer errors

* Operational errors (e.g. API received an invalid input) refer to known cases where the error impact is fully understood and can be handled thoughtfully. On the other hand, programmer error (e.g. trying to read an undefined variable) refers to unknown code failures that dictate to gracefully restart the application

## 2.4 Handle errors centrally, not within an Express middleware

* Error handling logic such as mail to admin and logging should be encapsulated in a dedicated and centralized object that all endpoints (e.g. Express middleware, cron jobs, unit-testing) call when an error comes in.

## 2.5 Document API errors using Swagger or GraphQL

* Let your API callers know which errors might come in return so they can handle these thoughtfully without crashing. For RESTful APIs, this is usually done with documentation frameworks like Swagger. If you’re using GraphQL, you can utilize your schema and comments as well.

## 2.6 Exit the process gracefully

* When an unknown error occurs – there is uncertainty about the application healthiness. Common practice suggests restarting the process carefully using a process management tool like Forever or PM2.

## 2.7 Use a mature logger to increase error visibility

* A set of mature logging tools like Winston, Bunyan, Log4js or Pino, will speed-up error discovery and understanding. So forget about console.log.

## 2.8 Test error flows using your favorite test framework

* Whether professional automated QA or plain manual developer testing – Ensure that your code not only satisfies positive scenarios but also handles and returns the right errors. Testing frameworks like Mocha & Chai can handle this easily.

## 2.9 Discover errors and downtime using APM products

* Monitoring and performance products (a.k.a APM) proactively gauge your codebase or API so they can automagically highlight errors, crashes and slow parts that you were missing.

## 2.10 Catch unhandled promise rejections

* Any exception thrown within a promise will get swallowed and discarded unless a developer didn’t forget to explicitly handle. Even if your code is subscribed to process.uncaughtException! Overcome this by registering to the event process.unhandledRejection.

## 2.11 Fail fast, validate arguments using a dedicated library

* This should be part of your Express best practices – Assert API input to avoid nasty bugs that are much harder to track later. The validation code is usually tedious unless you are using a very cool helper library like Joi.

# 3. Code Style Practices

## 3.1 Use ESLint

* ESLint is the de-facto standard for checking possible code errors and fixing code style, not only to identify nitty-gritty spacing issues but also to detect serious code anti-patterns like developers throwing errors without classification. Though ESLint can automatically fix code styles, other tools like prettier and beautify are more powerful in formatting the fix and work in conjunction with ESLint.

## 3.2 Node.js specific plugins

* On top of ESLint standard rules that cover vanilla JavaScript, add Node.js specific plugins like eslint-plugin-node, eslint-plugin-mocha and eslint-plugin-node-security.

## 3.3 Start a Codeblocks Curly Braces on the Same Line

* The opening curly braces of a code block should be on the same line as the opening statement.

## 3.4 Separate your statements properly

* Use ESLint to gain awareness about separation concerns. Prettier or Standardjs can automatically resolve these issues.

## 3.5 Name your functions

* Name all functions, including closures and callbacks. Avoid anonymous functions. This is especially useful when profiling a node app. Naming all functions will allow you to easily understand what you’re looking at when checking a memory snapshot.

## 3.6 Use naming conventions for variables, constants, functions, and classes

* Use lowerCamelCase when naming constants, variables, and functions and UpperCamelCase (capital first letter as well) when naming classes. This will help you to easily distinguish between plain variables/functions, and classes that require instantiation. Use descriptive names, but try to keep them short.

## 3.7 Prefer const over let. Ditch the var

* Using const means that once a variable is assigned, it cannot be reassigned. Preferring const will help you to not be tempted to use the same variable for different uses, and make your code clearer. If a variable needs to be reassigned, in a for loop, for example, use let to declare it. Another important aspect of let is that a variable declared using it is only available in the block scope in which it was defined. var is function scoped, not block scope, and shouldn’t be used in ES6 now that you have const and let at your disposal.

## 3.8 Require modules first, not inside functions

* Require modules at the beginning of each file, before and outside of any functions. This simple best practice will not only help you easily and quickly tell the dependencies of a file right at the top but also avoids a couple of potential problems.

## 3.9 Require modules by folders, as opposed to the files directly

* When developing a module/library in a folder, place an index.js file that exposes the module’s internals so every consumer will pass through it. This serves as an ‘interface’ to your module and eases future changes without breaking the contract.

## 3.10 Use the === operator

* Prefer the strict equality operator === over the weaker abstract equality operator ==. == will compare two variables after converting them to a common type. There is no type conversion in ===, and both variables must be of the same type to be equal.

## 3.11 Use Async Await, avoid callbacks

* Node 8 LTS now has full support for Async-await. This is a new way of dealing with asynchronous code which supersedes callbacks and promises. Async-await is non-blocking, and it makes asynchronous code look synchronous. The best gift you can give to your code is using async-await which provides a much more compact and familiar code syntax like try-catch.

## 3.12 Use arrow function expressions (=>)

* Though it’s recommended to use async-await and avoid function parameters when dealing with older APIs that accept promises or callbacks – arrow functions make the code structure more compact and keep the lexical context of the root function (i.e. this).

# 4. Testing And Overall Quality Practices

## 4.1 At the very least, write API (component) testing

* Most projects just don’t have any automated testing due to short timetables or often the ‘testing project’ ran out of control and was abandoned. For that reason, prioritize and start with API testing which is the easiest way to write and provides more coverage than unit testing (you may even craft API tests without code using tools like Postman. Afterward, should you have more resources and time, continue with advanced test types like unit testing, DB testing, performance testing, etc.

## 4.2 Include 3 parts in each test name

* Make the test speak at the requirements level so it’s self-explanatory also to QA engineers and developers who are not familiar with the code internals. State in the test name what is being tested (unit under test), under what circumstances and what is the expected result.

## 4.3 Structure tests by the AAA pattern

* Structure your tests with 3 well-separated sections: Arrange, Act & Assert (AAA). The first part includes the test setup, then the execution of the unit under test and finally the assertion phase. Following this structure guarantees that the reader spends no brain CPU on understanding the test plan.

## 4.4 Detect code issues with a linter

* Use a code linter to check the basic quality and detect anti-patterns early. Run it before any test and add it as a pre-commit git-hook to minimize the time needed to review and correct any issue.

## 4.5 Avoid global test fixtures and seeds, add data per-test

* To prevent tests coupling and easily reason about the test flow, each test should add and act on its own set of DB rows. Whenever a test needs to pull or assume the existence of some DB data – it must explicitly add that data and avoid mutating any other records.

## 4.6 Constantly inspect for vulnerable dependencies

* Even the most reputable dependencies such as Express have known vulnerabilities. This can get easily tamed using community and commercial tools such as npm audit and snyk.io that can be invoked from your CI on every build.

## 4.7 Tag your tests

* Different tests must run on different scenarios: quick smoke, IO-less, tests should run when a developer saves or commits a file, full end-to-end tests usually run when a new pull request is submitted, etc. This can be achieved by tagging tests with keywords like #cold #api #sanity so you can grep with your testing harness and invoke the desired subset. For example, this is how you would invoke only the sanity test group with Mocha: mocha –grep ‘sanity’.

## 4.8 Check your test coverage, it helps to identify wrong test patterns

* Code coverage tools like Istanbul/NYC are great for various reasons: it helps to identify a decrease in testing coverage, and last but not least it highlights testing mismatches: by looking at colored code coverage reports you may notice, for example, code areas that are never tested like catch clauses (meaning that tests only invoke the happy paths and not how the app behaves on errors). Set it to fail builds if the coverage falls under a certain threshold.

## 4.9 Inspect for outdated packages

* Use your preferred tool (e.g. ‘npm outdated’ or npm-check-updates to detect installed packages which are outdated, inject this check into your CI pipeline and even make a build fail in a severe scenario. For example, a severe scenario might be when an installed package is 5 patch commits behind (e.g. local version is 1.3.1 and repository version is 1.3.8) or it is tagged as deprecated by its author – kill the build and prevent deploying this version.

## 4.10 Use production-like env for e2e testing

* End to end (e2e) testing which includes live data used to be the weakest link of the CI process as it depends on multiple heavy services like DB. Use an environment which is as close to your real production as possible like a-continue.

## 4.11 Refactor regularly using static analysis tools

* Using static analysis tools helps by giving objective ways to improve code quality and keeps your code maintainable. You can add static analysis tools to your CI build to fail when it finds code smells. Its main selling points over plain linting are the ability to inspect quality in the context of multiple files (e.g. detect duplications), perform advanced analysis (e.g. code complexity) and follow the history and progress of code issues. Two examples of tools you can use are Sonarqube and Code Climate.

## 4.12 Carefully choose your CI platform (Jenkins vs CircleCI vs Travis vs Rest of the world)

* Your continuous integration platform (CICD) will host all the quality tools (e.g test, lint) so it should come with a vibrant ecosystem of plugins. Jenkins used to be the default for many projects as it has the biggest community along with a very powerful platform at the price of a complex setup that demands a steep learning curve. Nowadays, it has become much easier to set up a CI solution using SaaS tools like CircleCI and others. These tools allow crafting a flexible CI pipeline without the burden of managing the whole infrastructure. Eventually, it’s a trade-off between robustness and speed – choose your side carefully.

# 5. Going To Production Practices

## 5.1. Monitoring

* At the very basic level, monitoring means you can easily identify when bad things happen at production. For example, by getting notified by email or Slack. Start with defining the core set of metrics that must be watched to ensure a healthy state – CPU, server RAM, Node process RAM (less than 1.4GB), the number of errors in the last minute, number of process restarts, average response time. Then go over some advanced features you might fancy and add to your wishlist. Some examples of a luxury monitoring feature: DB profiling, cross-service measuring (i.e. measure business transaction), front-end integration, expose raw data to custom BI clients, Slack notifications and many others.

## 5.2. Increase transparency using smart logging

* Logs can be a dumb warehouse of debug statements or the enabler of a beautiful dashboard that tells the story of your app. Plan your logging platform from day 1: how logs are collected, stored and analyzed to ensure that the desired information (e.g. error rate, following an entire transaction through services and servers, etc) can really be extracted.

## 5.3. Delegate anything possible (e.g. gzip, SSL) to a reverse proxy

* Node is awfully bad at doing CPU intensive tasks like gzipping, SSL termination, etc. You should use ‘real’ middleware services like nginx, HAproxy or cloud vendor services instead.

## 5.4. Lock dependencies

* Your code must be identical across all environments, but amazingly npm lets dependencies drift across environments by default – when you install packages at various environments it tries to fetch packages’ latest patch version. Overcome this by using npm config files, .npmrc, that tell each environment to save the exact (not the latest) version of each package. Alternatively, for finer-grained control use npm shrinkwrap. \*Update: as of NPM5, dependencies are locked by default. The new package manager in town, Yarn, also got us covered by default.

## 5.5. Guard process uptime using the right tool

* The process must go on and get restarted upon failures. For simple scenarios, process management tools like PM2 might be enough but in today’s ‘dockerized’ world, cluster management tools should be considered as well.
* Running dozens of instances without a clear strategy and too many tools together (cluster management, docker, PM2) might lead to DevOps chaos.

## 5.6. Utilize all CPU cores

* At its basic form, a Node app runs on a single CPU core while all others are left idling. It’s your duty to replicate the Node process and utilize all CPUs – For small-medium apps you may use Node Cluster or PM2. For a larger app consider replicating the process using some Docker cluster (e.g. K8S, ECS) or deployment scripts that are based on Linux init system (e.g. systemd).

## 5.7. Create a ‘maintenance endpoint’

* Expose a set of system-related information, like memory usage and REPL, etc in a secured API. Although it’s highly recommended to rely on the standard and battle-tests tools, some valuable information and operations are easier done using code.

## 5.8. Discover errors and downtime using APM products

* Application monitoring and performance products (a.k.a APM) proactively gauge codebase and API so they can auto-magically go beyond traditional monitoring and measure the overall user-experience across services and tiers. For example, some APM products can highlight a transaction that loads too slow on the end-users side while suggesting the root cause.

## 5.9. Make your code production-ready

* Code with the end in mind, plan for production from day 1.
* Following is a list of development tips that greatly affect production maintenance and stability:

1. The twelve-factor guide
   * Get familiar with the Twelve factors guide
2. Be stateless
   * Save no data locally on a specific web server (see separate bullet – ‘Be Stateless’)
3. Cache
   * Utilize cache heavily, yet never fail because of cache mismatch
4. Test memory
   * gauge memory usage and leaks as part of your development flow, tools such as ‘memwatch’ can greatly facilitate this task
5. Name functions
   * Minimize the usage of anonymous functions (i.e. inline callback) as a typical memory profiler will provide memory usage per method name
6. Use CI tools
   * Use CI tool to detect failures before sending to production. For example, use ESLint to detect reference errors and undefined variables. Use –trace-sync-io to identify code that uses synchronous APIs (instead of the async version)
7. Log wisely
   * Include in each log statement contextual information, hopefully in JSON format so log aggregators tools such as Elastic can search upon those properties (see separate bullet – ‘Increase visibility using smart logs’). Also, include transaction-id that identifies each request and allows to correlate lines that describe the same transaction (see separate bullet – ‘Include Transaction-ID’)
8. Error management
   * Error handling is the Achilles’ heel of Node.js production sites – many Node processes are crashing because of minor errors while others hang on alive in a faulty state instead of crashing. Setting your error handling strategy is absolutely critical.

## 5.10. Measure and guard the memory usage

* The v8 engine has soft limits on memory usage (1.4GB) and there are known paths to leak memory in Node’s code – thus watching Node’s process memory is a must. In small apps, you may gauge memory periodically using shell commands but in medium-large apps consider baking your memory watch into a robust monitoring system.

## 5.11. Get your frontend assets out of Node

* Serve frontend content using dedicated middleware (nginx, S3, CDN) because Node performance really gets hurt when dealing with many static files due to its single-threaded model.

## 5.12. Be stateless, kill your servers almost every day

* Store any type of data (e.g. user sessions, cache, uploaded files) within external data stores. Consider ‘killing’ your servers periodically or use ‘serverless’ platform (e.g. AWS Lambda) that explicitly enforces a stateless behavior.

## 5.13. Use tools that automatically detect vulnerabilities

* Even the most reputable dependencies such as Express have known vulnerabilities (from time to time) that can put a system at risk. This can be easily tamed using community and commercial tools that constantly check for vulnerabilities and warn (locally or at GitHub), some can even patch them immediately.

## 5.14. Assign a transaction id to each log statement

* Assign the same identifier, transaction-id: {some value}, to each log entry within a single request. Then when inspecting errors in logs, easily conclude what happened before and after. Unfortunately, this is not easy to achieve in Node due to its async nature.

## 5.15. Set NODE\_ENV

* Set the environment variable NODE\_ENV to ‘production’ or ‘development’ to flag whether production optimizations should get activated – many npm packages determine the current environment and optimize their code for production.

## 5.16. Design automated, atomic and zero-downtime deployments

* Research shows that teams who perform many deployments lower the probability of severe production issues. Fast and automated deployments that don’t require risky manual steps and service downtime significantly improve the deployment process. You should probably achieve this using Docker combined with CI tools as they became the industry standard for streamlined deployment.

## 5.17. Use an LTS release of Node.js

* Ensure you are using an LTS version of Node.js to receive critical bug fixes, security updates and performance improvements.

## 5.18. Don’t route logs within the app

* Log destinations should not be hard-coded by developers within the application code, but instead should be defined by the execution environment the application runs in. Developers should write logs to stdout using a logger utility and then let the execution environment (container, server, etc.) pipe the stdout stream to the appropriate destination (i.e. Splunk, Graylog, ElasticSearch, etc.).

# 6. Security Best Practices

## 6.1. Embrace linter security rules

* Make use of security-related linter plugins such as eslint-plugin-security to catch security vulnerabilities and issues as early as possible, preferably while they’re being coded. This can help to catch security weaknesses like using eval, invoking a child process or importing a module with a string literal (e.g. user input).

## 6.2. Limit concurrent requests using a middleware

* DOS attacks are very popular and relatively easy to conduct. Implement rate limiting using an external service such as cloud load balancers, cloud firewalls, nginx, rate-limiter-flexible package, or (for smaller and less critical apps) a rate-limiting middleware (e.g. express-rate-limit).

## 6.3 Extract secrets from config files or use packages to encrypt them

* Never store plain-text secrets in configuration files or source code. Instead, make use of secret-management systems like Vault products, Kubernetes/Docker Secrets, or using environment variables. As a last resort, secrets stored in source control must be encrypted and managed (rolling keys, expiring, auditing, etc). Make use of pre-commit/push hooks to prevent committing secrets accidentally.

## 6.4. Prevent query injection vulnerabilities with ORM/ODM libraries

* 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. Never just use JavaScript template strings or string concatenation to inject values into queries as this opens your application to a wide spectrum of vulnerabilities. All the reputable Node.js data access libraries (e.g. Sequelize, Knex, mongoose) have built-in protection against injection attacks.

## 6.5. Adjust the HTTP response headers for enhanced security

* 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 helmets.

## 6.6. Constantly and automatically inspect for vulnerable dependencies

* With the npm ecosystem it is common to have many dependencies for a project. Dependencies should always be kept in check as new vulnerabilities are found. Use tools like npm audit or snyk to track, monitor and patch vulnerable dependencies. Integrate these tools with your CI setup so you catch a vulnerable dependency before it makes it to production.

## 6.7. Avoid using the Node.js crypto library for handling passwords, use Bcrypt

* Passwords or secrets (API keys) should be stored using a secure hash + salt function like bcrypt, that should be a preferred choice over its JavaScript implementation due to performance and security reasons.

## 6.8. Escape HTML, JS and CSS output

* Untrusted data that is sent down to the browser might get executed instead of just being displayed, this is commonly referred to as a cross-site-scripting (XSS) attack. Mitigate this by using dedicated libraries that explicitly mark the data as pure content that should never get executed (i.e. encoding, escaping).

## 6.9. Validate incoming JSON schemas

* Validate the incoming requests’ body payload and ensure it meets expectations, fail fast if it doesn’t. To avoid tedious validation coding within each route you may use lightweight JSON-based validation schemas such as jsonschema or joi.

## 6.10. Support blacklisting JWTs

* When using JSON Web Tokens (for example, with Passport.js), by default there’s no mechanism to revoke access from issued tokens. Once you discover some malicious user activity, there’s no way to stop them from accessing the system as long as they hold a valid token. Mitigate this by implementing a blacklist of untrusted tokens that are validated on each request.

## 6.11. Prevent brute-force attacks against authorization

* A simple and powerful technique is to limit authorization attempts using two metrics:
* The first is a number of consecutive failed attempts by the same user unique ID/name and IP address.The second is a number of failed attempts from an IP address over some long period of time.

## 6.12. Run Node.js as non-root user

* There is a common scenario where Node.js runs as a root user with unlimited permissions. For example, this is the default behavior in Docker containers. It’s recommended to create a non-root user and either bake it into the Docker image (examples given below) or run the process on this user’s behalf by invoking the container with the flag “-u username”.

## 6.13. Limit payload size using a reverse-proxy or a middleware

* The bigger the body payload is, the harder your single thread works in processing it. This is an opportunity for attackers to bring servers to their knees without tremendous amount of requests (DOS/DDOS attacks). Mitigate this limiting the body size of incoming requests on the edge (e.g. firewall, ELB) or by configuring express body parser to accept only small-size payloads.

## 6.14. Avoid JavaScript eval statements

* eval is evil as it allows executing custom JavaScript code during run time. This is not just a performance concern but also an important security concern due to malicious JavaScript code that may be sourced from user input. Another language feature that should be avoided is new Function constructor. setTimeout and setInterval should never be passed dynamic JavaScript code either.

## 6.15. Prevent evil RegEx from overloading your single thread execution

* Regular Expressions, while being handy, pose a real threat to JavaScript applications at large, and the Node.js platform in particular. A user input for text to match might require an outstanding amount of CPU cycles to process. RegEx processing might be inefficient to an extent that a single request that validates 10 words can block the entire event loop for 6 seconds and set the CPU on fire. For that reason, prefer third-party validation packages like validator.js instead of writing your own Regex patterns, or make use of safe-regex to detect vulnerable regex patterns.

## 6.16. Avoid module loading using a variable

* Avoid requiring/importing another file with a path that was given as parameter due to the concern that it could have originated from user input. This rule can be extended for accessing files in general (i.e. fs.readFile()) or other sensitive resource access with dynamic variables originating from user input. Eslint-plugin-security linter can catch such patterns and warn early enough.

## 6.17. Run unsafe code in a sandbox

* When tasked to run external code that is given at run-time (e.g. plugin), use any sort of ‘sandbox’ execution environment that isolates and guards the main code against the plugin. This can be achieved using a dedicated process (e.g. cluster.fork()), serverless environment or dedicated npm packages that act as a sandbox.
* As a rule of thumb, one should run his own JavaScript files only. Theories aside, real-world scenarios demand to execute JavaScript files that are being passed dynamically at run-time. For example, consider a dynamic framework like webpack that accepts custom loaders and executes those dynamically during build time. In the existence of some malicious plugin we wish to minimize the damage and maybe even let the flow terminate successfully – this requires to run the plugins in a sandbox environment that is fully isolated in terms of resources, crashes and the information we share with it.
* Three main options can help in achieving this isolation:

1. a dedicated child process – this provides a quick information isolation but demand to tame the child process, limit its execution time and recover from errors
2. a cloud serverless framework ticks all the sandbox requirements but deployment and invoking a FaaS function dynamically is not a walk in the park
3. Some npm libraries, like sandbox and vm2 allow execution of isolated code in 1 single line of code. Though this latter option wins in simplicity it provides a limited protection

## 6.18. Take extra care when working with child processes

* Avoid using child processes when possible and validate and sanitize input to mitigate shell injection attacks if you still have to. Prefer using child\_process.execFile which by definition will only execute a single command with a set of attributes and will not allow shell parameter expansion.

## 6.19. Hide error details from clients

* An integrated express error handler hides the error details by default. However, great are the chances that you implement your own error handling logic with custom Error objects (considered by many as a best practice). If you do so, ensure not to return the entire Error object to the client, which might contain some sensitive application details.

## 6.20. Configure 2FA for npm or Yarn

* Any step in the development chain should be protected with MFA (multi-factor authentication), npm/Yarn are a sweet opportunity for attackers who can get their hands on some developer’s password. Using developer credentials, attackers can inject malicious code into libraries that are widely installed across projects and services. Maybe even across the web if published in public. Enabling 2-factor-authentication in npm leaves almost zero chances for attackers to alter your package code.

## 6.21. Modify session middleware settings

* Each web framework and technology has its known weaknesses - telling an attacker which web framework we use is a great help for them. Using the default settings for session middlewares can expose your app to module- and framework-specific hijacking attacks in a similar way to the X-Powered-By header. Try hiding anything that identifies and reveals your tech stack (E.g. Node.js, express).

## 6.22. Avoid DOS attacks by explicitly setting when a process should crash

* The Node process will crash when errors are not handled. Many best practices even recommend to exit even though an error was caught and got handled. Express, for example, will crash on any asynchronous error - unless you wrap routes with a catch clause. This opens a very sweet attack spot for attackers who recognize what input makes the process crash and repeatedly send the same request. There’s no instant remedy for this but a few techniques can mitigate the pain: Alert with critical severity anytime a process crashes due to an unhandled error, validate the input and avoid crashing the process due to invalid user input, wrap all routes with a catch and consider not to crash when an error originated within a request (as opposed to what happens globally).

## 6.23. Prevent unsafe redirects

* Redirects that do not validate user input can enable attackers to launch phishing scams, steal user credentials, and perform other malicious actions.

## 6.24. Avoid publishing secrets to the npm registry

Precautions should be taken to avoid the risk of accidentally publishing secrets to public npm registries. An .npmignore file can be used to blacklist specific files or folders, or the files array in package.json can act as a whitelist.

# REACT

## Composition

* The key feature of React is composition of components. Components written by different people should work well together. It is important to us that you can add functionality to a component without causing rippling changes throughout the codebase.
* For example, it should be possible to introduce some local state into a component without changing any of the components using it. Similarly, it should be possible to add some initialization and teardown code to any component when necessary.
* There is nothing “bad” about using state or lifecycle methods in components. Like any powerful feature, they should be used in moderation, but we have no intention to remove them. On the contrary, we think they are integral parts of what makes React useful. We might enable more functional patterns in the future, but both local state and lifecycle methods will be a part of that model.
* Components are often described as “just functions” but in our view they need to be more than that to be useful. In React, components describe any composable behavior, and this includes rendering, lifecycle, and state. Some external libraries like Relay augment components with other responsibilities such as describing data dependencies. It is possible that those ideas might make it back into React too in some form.

## Common Abstraction

* In general we resist adding features that can be implemented in userland. We don’t want to bloat your apps with useless library code. However, there are exceptions to this.
* For example, if React didn’t provide support for local state or lifecycle methods, people would create custom abstractions for them. When there are multiple abstractions competing, React can’t enforce or take advantage of the properties of either of them. It has to work with the lowest common denominator.
* This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don’t do it lightly. When we do it, it’s because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle methods, cross-browser event normalization are good examples of this.
* We always discuss such improvement proposals with the community. You can find some of those discussions by the “big picture” label on the React issue tracker.

## Escape Hatches

* React is pragmatic. It is driven by the needs of the products written at Facebook. While it is influenced by some paradigms that are not yet fully mainstream such as functional programming, staying accessible to a wide range of developers with different skills and experience levels is an explicit goal of the project.
* If we want to deprecate a pattern that we don’t like, it is our responsibility to consider all existing use cases for it and educate the community about the alternatives before we deprecate it. If some pattern that is useful for building apps is hard to express in a declarative way, we will provide an imperative API for it. If we can’t figure out a perfect API for something that we found necessary in many apps, we will provide a temporary subpar working API as long as it is possible to get rid of it later and it leaves the door open for future improvements.

## Stability

* We value API stability. At Facebook, we have more than 50 thousand components using React. Many other companies, including Twitter and Airbnb, are also heavy users of React. This is why we are usually reluctant to change public APIs or behavior.
* However we think stability in the sense of “nothing changes” is overrated. It quickly turns into stagnation. Instead, we prefer the stability in the sense of “It is heavily used in production, and when something changes, there is a clear (and preferably automated) migration path.”
* When we deprecate a pattern, we study its internal usage at Facebook and add deprecation warnings. They let us assess the impact of the change. Sometimes we back out if we see that it is too early, and we need to think more strategically about getting the codebases to the point where they are ready for this change.
* If we are confident that the change is not too disruptive and the migration strategy is viable for all use cases, we release the deprecation warning to the open source community. We are closely in touch with many users of React outside of Facebook, and we monitor popular open source projects and guide them in fixing those deprecations.
* Given the sheer size of the Facebook React codebase, successful internal migration is often a good indicator that other companies won’t have problems either. Nevertheless sometimes people point out additional use cases we haven’t thought of, and we add escape hatches for them or rethink our approach.
* We don’t deprecate anything without a good reason. We recognize that sometimes deprecations warnings cause frustration but we add them because deprecations clean up the road for the improvements and new features that we and many people in the community consider valuable.
* For example, we added a warning about unknown DOM props in React 15.2.0. Many projects were affected by this. However fixing this warning is important so that we can introduce the support for custom attributes to React. There is a reason like this behind every deprecation that we add.
* When we add a deprecation warning, we keep it for the rest of the current major version, and change the behavior in the next major version. If there is a lot of repetitive manual work involved, we release a codemod script that automates most of the change. Codemods enable us to move forward without stagnation in a massive codebase, and we encourage you to use them as well.
* You can find the codemods that we released in the react-codemod repository.

## Interoperability

* We place high value in interoperability with existing systems and gradual adoption. Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can start using React for a small feature rather than rewrite their code to bet on it.
* This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

## Scheduling

* Even when your components are described as functions, when you use React you don’t call them directly. Every component returns a description of what needs to be rendered, and that description may include both user-written components like <LikeButton> and platform-specific components like <div>. It is up to React to “unroll” <LikeButton> at some point in the future and actually apply changes to the UI tree according to the render results of the components recursively.
* This is a subtle distinction but a powerful one. Since you don’t call that component function but let React call it, it means React has the power to delay calling it if necessary. In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start delaying some updates to avoid dropping frames.
* This is a common theme in Re
* act design. Some popular libraries implement the “push” approach where computations are performed when the new data is available. React, however, sticks to the “pull” approach where computations can be delayed until necessary.
* React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.
* If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.
* To be clear, we are not taking advantage of this right now. However the freedom to do something like this is why we prefer to have control over scheduling, and why setState() is asynchronous. Conceptually, we think of it as “scheduling an update”.
* The control over scheduling would be harder for us to gain if we let the user directly compose views with a “push” based paradigm common in some variations of Functional Reactive Programming. We want to own the “glue” code.
* It is a key goal for React that the amount of the user code that executes before yielding back into React is minimal. This ensures that React retains the capability to schedule and split work in chunks according to what it knows about the UI.
* There is an internal joke in the team that React should have been called “Schedule” because React does not want to be fully “reactive”.

## Developer Experience

* Providing a good developer experience is important to us.
* For example, we maintain React DevTools which let you inspect the React component tree in Chrome and Firefox. We have heard that it brings a big productivity boost both to the Facebook engineers and to the community.
* We also try to go an extra mile to provide helpful developer warnings. For example, React warns you in development if you nest tags in a way that the browser doesn’t understand, or if you make a common typo in the API. Developer warnings and the related checks are the main reason why the development version of React is slower than the production version.
* The usage patterns that we see internally at Facebook help us understand what the common mistakes are, and how to prevent them early. When we add new features, we try to anticipate the common mistakes and warn about them.
* We are always looking out for ways to improve the developer experience. We love to hear your suggestions and accept your contributions to make it even better.

## Debugging

* When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.
* If you see something wrong on the screen, you can open React DevTools, find the component responsible for rendering, and then see if the props and state are correct. If they are, you know that the problem is in the component’s render() function, or some function that is called by render(). The problem is isolated.
* If the state is wrong, you know that the problem is caused by one of the setState() calls in this file. This, too, is relatively simple to locate and fix because usually there are only a few setState() calls in a single file.
* If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first “poisoned the well” by passing bad props down.
* This ability to trace any UI to the data that produced it in the form of current props and state is very important to React. It is an explicit design goal that state is not “trapped” in closures and combinators, and is available to React directly.
* While the UI is dynamic, we believe that synchronous render() functions of props and state turn debugging from guesswork into a boring but finite procedure. We would like to preserve this constraint in React even though it makes some use cases, like complex animations, harder.

## Configuration

* We find global runtime configuration options to be problematic.
* For example, it is occasionally requested that we implement a function like React.configure(options) or React.register(component). However this poses multiple problems, and we are not aware of good solutions to them.
* What if somebody calls such a function from a third-party component library? What if one React app embeds another React app, and their desired configurations are incompatible? How can a third-party component specify that it requires a particular configuration? We think that global configuration doesn’t work well with composition. Since composition is central to React, we don’t provide global configuration in code.
* We do, however, provide some global configuration on the build level. For example, we provide separate development and production builds. We may also add a profiling build in the future, and we are open to considering other build flags.

## Beyond the DOM

* We see the value of React in the way it allows us to write components that have fewer bugs and compose together well. DOM is the original rendering target for React but React Native is just as important both to Facebook and the community.
* Being renderer-agnostic is an important design constraint of React. It adds some overhead in the internal representations. On the other hand, any improvements to the core translate across platforms.
* Having a single programming model lets us form engineering teams around products instead of platforms. So far the tradeoff has been worth it for us.

## Implementation

* We try to provide elegant APIs where possible. We are much less concerned with the implementation being elegant. The real world is far from perfect, and to a reasonable extent we prefer to put the ugly code into the library if it means the user does not have to write it. When we evaluate new code, we are looking for an implementation that is correct, performant and affords a good developer experience. Elegance is secondary.
* We prefer boring code to clever code. Code is disposable and often changes. So it is important that it doesn’t introduce new internal abstractions unless absolutely necessary. Verbose code that is easy to move around, change and remove is preferred to elegant code that is prematurely abstracted and hard to change.

## Optimized for Tooling

* Some commonly used APIs have verbose names. For example, we use componentDidMount() instead of didMount() or onMount(). This is intentional. The goal is to make the points of interaction with the library highly visible.
* In a massive codebase like Facebook, being able to search for uses of specific APIs is very important. We value distinct verbose names, and especially for the features that should be used sparingly. For example, dangerouslySetInnerHTML is hard to miss in a code review.
* Optimizing for search is also important because of our reliance on codemods to make breaking changes. We want it to be easy and safe to apply vast automated changes across the codebase, and unique verbose names help us achieve this. Similarly, distinctive names make it easy to write custom lint rules about using React without worrying about potential false positives.
* JSX plays a similar role. While it is not required with React, we use it extensively at Facebook both for aesthetic and pragmatic reasons.
* In our codebase, JSX provides an unambiguous hint to the tools that they are dealing with a React element tree. This makes it possible to add build-time optimizations such as hoisting constant elements, safely lint and codemod internal component usage, and include JSX source location into the warnings.

## Dogfooding

* We try our best to address the problems raised by the community. However we are likely to prioritize the issues that people are also experiencing internally at Facebook. Perhaps counter-intuitively, we think this is the main reason why the community can bet on React.
* Heavy internal usage gives us the confidence that React won’t disappear tomorrow. React was created at Facebook to solve its problems. It brings tangible business value to the company and is used in many of its products. Dogfooding it means that our vision stays sharp and we have a focused direction going forward.
* This doesn’t mean that we ignore the issues raised by the community. For example, we added support for web components and SVG to React even though we don’t rely on either of them internally. We are actively listening to your pain points and address them to the best of our ability. The community is what makes React special to us, and we are honored to contribute back.
* After releasing many open source projects at Facebook, we have learned that trying to make everyone happy at the same time produced projects with poor focus that didn’t grow well. Instead, we found that picking a small audience and focusing on making them happy brings a positive net effect. That’s exactly what we did with React, and so far solving the problems encountered by Facebook product teams has translated well to the open source community.
* The downside of this approach is that sometimes we fail to give enough focus to the things that Facebook teams don’t have to deal with, such as the “getting started” experience. We are acutely aware of this, and we are thinking of how to improve in a way that would benefit everyone in the community without making the same mistakes we did with open source projects before.