Engineer

ESLint

a rule-driven linter for JavaScript

What it is

ESLint is an open-source static analysis tool that reads JavaScript source code, tests it against a configured set of rules and reports each place the code breaks one, repairing the simpler cases itself when asked. On the command line it is the `eslint` command, installed from npm as a development dependency of the project it checks. Nicholas C. Zakas started it in June 2013, and the project still introduces itself as a configurable JavaScript linter.

It never runs the program. A parser turns each file into an abstract syntax tree in the ESTree format, and every rule is a small module that asks to be called at particular kinds of node: each function, each comparison, each `await`. The rule looks at what it is handed and, if something is wrong, reports a message and a location. Some rules also carry a fix, an edit the project only accepts when it corrects the problem without changing what the program does, and `eslint --fix` applies those to the files. Others offer suggestions instead, which may change behaviour and are therefore applied one at a time, by a person, in an editor.

Nearly everything around that core can be swapped. Plugins are npm packages that add rules, shareable configurations, processors or support for other languages. A different parser lets ESLint read syntax the default one does not understand, such as TypeScript. Processors pull JavaScript out of other files, for instance the code blocks in a Markdown document, and formatters change how the report is printed. The project's own account of its philosophy is that every rule stands alone, can be switched off, and can be set to warn or to fail on its own terms.

All of this is set in one JavaScript file at the root of the project, usually `eslint.config.js` or its `.mjs` and `.cjs` variants, which exports an array of objects. Each object names the files it applies to and the rules, globals and parser options that hold for them, and later objects override earlier ones. That layout, called flat config, became the default in ESLint v9.0.0 on 5 April 2024. ESLint v10.0.0, released on 6 February 2026, removed the older `.eslintrc` format and the `.eslintignore` file outright.

Use the concept in a decision

Put “What is ESLint?” to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

A linter earns its place at one moment: when it stops a change from merging. ESLint makes that call through its exit code, and the exit code follows severity. Every rule is set to `"off"`, `"warn"` or `"error"`. An error makes the command exit with 1. A warning is printed and the command still exits with 0, unless `--max-warnings` sets a ceiling, and that ceiling is disabled by default. Exit code 2 is reserved for a broken configuration or an internal crash.

So the decision a team really makes with ESLint is what a warning means. A warning that never fails a build is advice, and advice in a terminal gets scrolled past once there are forty lines of it. Teams that want the linter to hold a line either set every rule they care about to `"error"` or run the gate with `--max-warnings 0`. Either way a healthy branch lints clean, and a single new line is news someone reads.

The payoff is in what JavaScript lets through. The language will load a file that refers to a variable nobody declared, defines the same key twice in an object literal, compares a value with `NaN` using `===`, or leaves code after a `return` that can never run. These problems can survive loading and remain unnoticed, or cause wrong behaviour when the code runs, including in production. The rules in ESLint’s recommended set catch all four before review, which leaves the reviewer free to read for design.

There are costs. A configuration accumulates opinions, and a rule that fires on correct code teaches people to stop reading the output, which is worse than having no rule. A linter is also only as good as the moment it runs: one that runs only in an editor, on some machines, checks some of the code some of the time.

What it looks like in practice

ESLint was one of six linters behind a zero-warnings bar set across six languages, where it checked TypeScript beside golangci-lint for Go, SQLFluff for SQL, Ruff for Python, ShellCheck for shell scripts and markdownlint for Markdown. What made the bar hold was the severity question above, settled once for everyone: every linter diagnostic is an error, there is no “warn” tier to hide in, and inline suppressions are banned. Pre-commit and pre-push hooks run the linters and the tests; failures in the covered rule and test set block the corresponding commit or push.

The same repository then needed rules no published package ships, the ones about how that particular codebase is meant to work. Fifteen custom ESLint rules hold the house patterns in TypeScript, alongside 274 check scripts, 268 of them registered into the commit hooks. One of those validators checks that nobody quietly added an inline suppression. The guards have 146 tests of their own, because a broken guard passes everything and nobody finds out, and the page names the price plainly: committing is slow.

This site holds its own build scripts to the same standard on a smaller scale. Its `eslint.config.mjs` starts from `js.configs.recommended` and adds only rules the code already passed when they were measured; the comments record a re-measurement on 2026-10-15 across all 199 non-deprecated rules that enabled 21 more whose subject is correctness rather than taste. Every rule is an error. `reportUnusedDisableDirectives`, a warning by default, is raised to an error too, so a disable comment that has stopped suppressing anything fails the run. The scripts declare Node and browser globals together, because the function bodies they hand to `page.evaluate` run inside a browser, a few lines away from code that runs in Node.

Where it goes wrong

Most ESLint trouble is the linter doing exactly what its configuration says, which is not always what the team believes it says.

The build is green and the terminal is full of warnings. Warnings leave the exit code at 0, so continuous integration passes while the list grows by a few lines a week, and after a while a real defect sits in it unread. Decide which rules matter, set them to `"error"`, switch the rest off, and add `--max-warnings 0` to the gate so that a warning can no longer accumulate.

A disable comment outlives its reason. A line was silenced for a good reason once; the code beneath it changed, the comment stayed, and now it hides the next problem on that line or nothing at all. Set `linterOptions.reportUnusedDisableDirectives` to `"error"` so directives that suppress nothing fail the run, and ask for a reason after `--` on the ones that remain, so a reviewer can judge them:

```
// eslint-disable-next-line no-console -- printing is this CLI's output
console.log(report);
```

An upgrade changes which files are checked. After moving to v10, paths that `.eslintignore` used to skip are suddenly reported, or one folder starts following different rules. Version 10 no longer honours `.eslintrc.*` or `.eslintignore`, and it begins its search for `eslint.config.*` in each linted file's own directory, so a stray config in a subfolder now governs that subfolder. Move ignores into a global `ignores` entry, convert the old file with `npx @eslint/migrate-config .eslintrc.json` and review what it writes, then list every `eslint.config.*` in the tree and remove the ones nobody meant to keep.

Browser code is reported as undefined. A Node script that hands functions to a browser gets `no-undef` errors on `document` and `window`, and someone turns `no-undef` off, which also hides genuine typos. Declare every environment the code really runs in through `languageOptions.globals`, for the files that run there.

A custom rule passes everything. A rule whose visitor names a node type that never occurs, or whose condition is inverted, simply never reports, and silence looks exactly like compliance. Test each custom rule with ESLint's `RuleTester`, giving it code that must pass and code that must fail, so a rule that stops reporting fails its own test before it fails the codebase.

Further reading

- Core Concepts — rules, fixes, plugins, parsers and processors, as the project defines them.
- Configuration Files — the flat config array, file patterns, global ignores and how objects merge.
- Configure Rules — severities, rule options and the comments that disable a rule for a line.
- Command Line Interface Reference — every flag, `--max-warnings` among them, and what each exit code means.
- ESLint v10.0.0 released — the removal of eslintrc and the new configuration lookup, in the release notes.
- Custom Rules — writing a rule against the syntax tree and proving it with `RuleTester`.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is ESLint?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/eslint/