

standard / standard

Public

JavaScript Style Guide, with linter & automatic code fixer

[standardjs.com](#)

MIT license

28k stars 2.4k forks

Star

Watch

<> Code

Issues 87

Pull requests 12

Actions

Security

Insights

master

...

fawazahmed0

on Feb 6

View code



# JavaScript Standard Style

chat

10 online

Test External

failing

Test Internal

passing

Old test

passing

npm

v17.0.0

downloads

10M/month

code style

standard

Sponsored by

Socket

Wormhole

[English](#) • [Español \(Latinoamérica\)](#) • [Français](#) • [Bahasa Indonesia](#) • [Italiano \(Italian\)](#) • [日本語 \(Japanese\)](#) • [한국어 \(Korean\)](#) • [Português \(Brasil\)](#) • [简体中文 \(Simplified Chinese\)](#) • [繁體中文 \(Taiwanese Mandarin\)](#)

## JavaScript style guide, linter, and formatter

---

This module saves you (and others!) time in three ways:

- **No configuration.** The easiest way to enforce code quality in your project. No decisions to make. No `.eslintrc` files to manage. It just works.
- **Automatically format code.** Just run `standard --fix` and say goodbye to messy or inconsistent code.
- **Catch style issues & programmer errors early.** Save precious code review time by eliminating back-and-forth between reviewer & contributor.

Give it a try by running `npx standard --fix` right now!

## Table of Contents

---

- Quick start
  - [Install](#)
  - [Usage](#)
  - [What you might do if you're clever](#)
- FAQ
  - [Why should I use JavaScript Standard Style?](#)
  - [Who uses JavaScript Standard Style?](#)
  - [Are there text editor plugins?](#)
  - [Is there a readme badge?](#)
  - [I disagree with rule X, can you change it?](#)
  - [But this isn't a real web standard!](#)
  - [Is there an automatic formatter?](#)
  - [How do I ignore files?](#)
  - [How do I disable a rule?](#)
  - [I use a library that pollutes the global namespace. How do I prevent "variable is not defined" errors?](#)
  - [How do I use experimental JavaScript \(ES Next\) features?](#)
  - [Can I use a JavaScript language variant, like Flow or TypeScript?](#)
  - [What about Mocha, Jest, Jasmine, QUnit, etc?](#)
  - [What about Web Workers and Service Workers?](#)
  - [What is the difference between warnings and errors?](#)

- [Can I check code inside of Markdown or HTML files?](#)
- [Is there a Git pre-commit hook?](#)
- [How do I make the output all colorful and pretty?](#)
- [Is there a Node.js API?](#)
- [How do I contribute to StandardJS?](#)

## Install

---

The easiest way to use JavaScript Standard Style is to install it globally as a Node command line program. Run the following command in Terminal:

```
$ npm install standard --global
```

Or, you can install `standard` locally, for use in a single project:

```
$ npm install standard --save-dev
```

*Note: To run the preceding commands, [Node.js](#) and [npm](#) must be installed.*

## Usage

---

After you've installed `standard`, you should be able to use the `standard` program. The simplest use case would be checking the style of all JavaScript files in the current working directory:

```
$ standard
Error: Use JavaScript Standard Style
  lib/torrent.js:950:11: Expected '===' and instead saw '=='.
```

If you've installed `standard` locally, run with `npx` instead:

```
$ npx standard
```

You can optionally pass in a directory (or directories) using the glob pattern. Be sure to quote paths containing glob patterns so that they are expanded by `standard` instead of your shell:

```
$ standard "src/util/**/*.js" "test/**/*.js"
```

**Note:** by default `standard` will look for all files matching the patterns: `**/*.js` , `**/*.jsx` .

## What you might do if you're clever

---

1. Add it to `package.json`

```
{
  "name": "my-cool-package",
  "devDependencies": {
    "standard": "*"
  },
  "scripts": {
    "test": "standard && node my-tests.js"
  }
}
```

2. Style is checked automatically when you run `npm test`

```
$ npm test
Error: Use JavaScript Standard Style
  lib/torrent.js:950:11: Expected '===' and instead saw '=='.
```

3. Never give style feedback on a pull request again!

## Why should I use JavaScript Standard Style?

---

The beauty of JavaScript Standard Style is that it's simple. No one wants to maintain multiple hundred-line style configuration files for every module/project they work on. Enough of this madness!








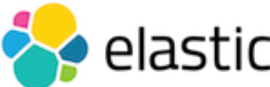






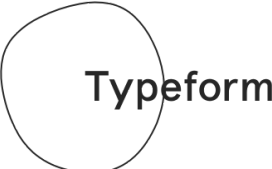


















This module saves you (and others!) time in three ways:

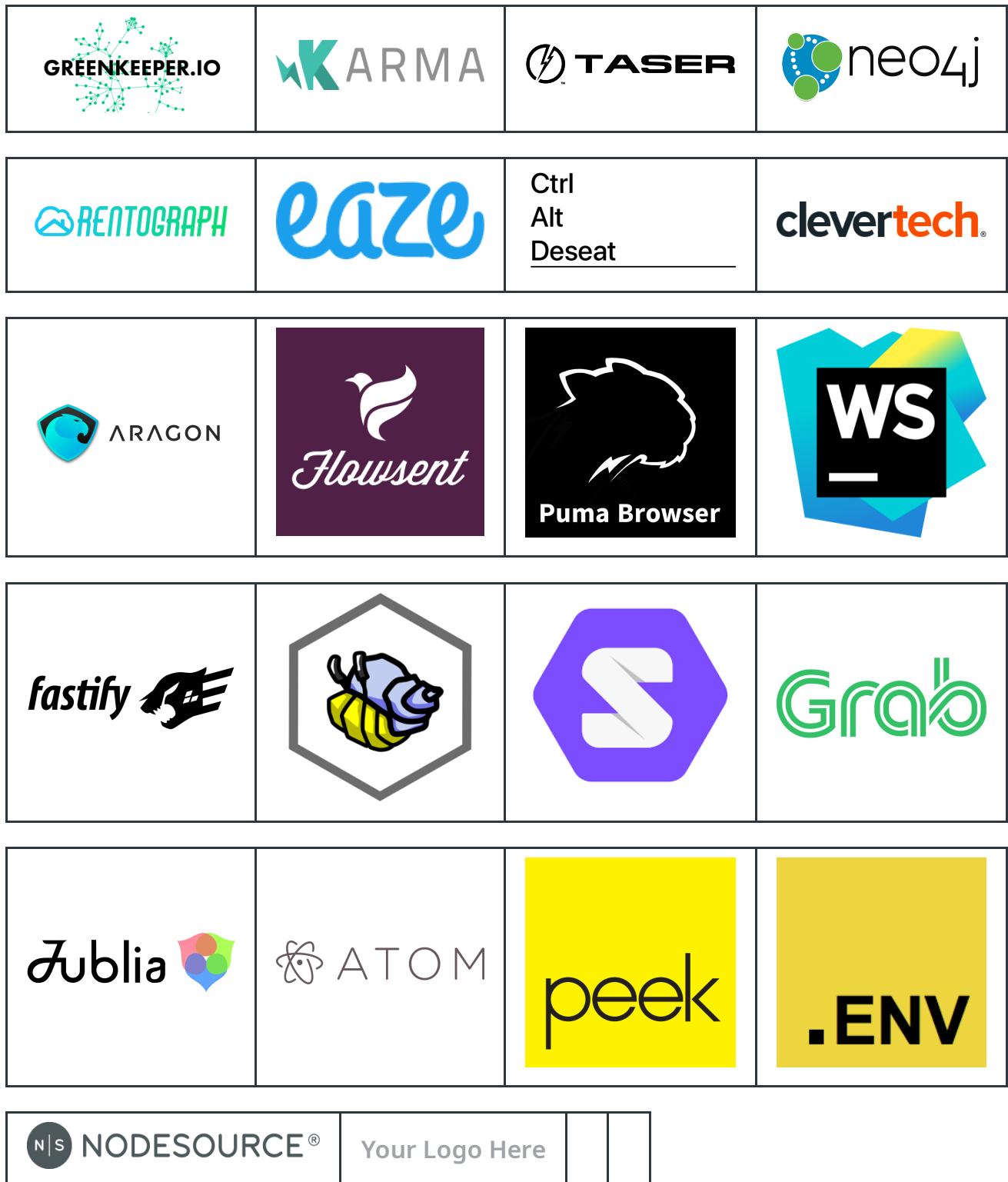
- **No configuration.** The easiest way to enforce consistent style in your project. Just drop it in.
- **Automatically format code.** Just run `standard --fix` and say goodbye to messy or inconsistent code.
- **Catch style issues & programmer errors early.** Save precious code review time by eliminating back-and-forth between reviewer & contributor.

Adopting `standard` style means ranking the importance of code clarity and community conventions higher than personal style. This might not make sense for 100% of projects and development cultures, however open source can be a hostile place for newbies. Setting up clear, automated contributor expectations makes a project healthier.

For more info, see the conference talk "[Write Perfect Code with Standard and ESLint](#)". In this talk, you'll learn about linting, when to use `standard` versus `eslint`, and how `prettier` compares to `standard`.

## Who uses JavaScript Standard Style?



In addition to companies, many community members use `standard` on packages that are [too numerous](#) to list here.

`standard` is also the top-starred linter in GitHub's [Clean Code Linter](#) showcase.

## Are there text editor plugins?

First, install `standard`. Then, install the appropriate plugin for your editor:

### Sublime Text

Using **Package Control**, install **SublimeLinter** and **SublimeLinter-contrib-standard**.

For automatic formatting on save, install **StandardFormat**.

## Atom

Install **linter-js-standard**.

Alternatively, you can install **linter-js-standard-engine**. Instead of bundling a version of `standard` it will automatically use the version installed in your current project. It will also work out of the box with other linters based on **standard-engine**.

For automatic formatting, install **standard-formatter**. For snippets, install **standardjs-snippets**.

## Visual Studio Code

Install **vscode-standard**. (Includes support for automatic formatting.)

For JS snippets, install: **vscode-standardjs-snippets**. For React snippets, install **vscode-react-standard**.

## Vim

Install **ale**. And add these lines to your `.vimrc` file.

```
let g:ale_linters = {  
  \   'javascript': ['standard'],  
  \  
  \  
let g:ale_fixers = {'javascript': ['standard']}
```

This sets `standard` as your only linter and fixer for javascript files and so prevents conflicts with `eslint`. For linting and automatic fixing on save, add these lines to `.vimrc`:

```
let g:ale_lint_on_save = 1  
let g:ale_fix_on_save = 1
```

Alternative plugins to consider include **neomake** and **syntastic**, both of which have built-in support for `standard` (though configuration may be necessary).

## Emacs

Install **Flycheck** and check out the **manual** to learn how to enable it in your projects.

## Brackets

Search the extension registry for "**Standard Code Style**" and click "Install".

## WebStorm (PhpStorm, IntelliJ, RubyMine, JetBrains, etc.)

WebStorm [recently announced native support](#) for `standard` directly in the IDE.

If you still prefer to configure `standard` manually, [follow this guide](#). This applies to all JetBrains products, including PhpStorm, IntelliJ, RubyMine, etc.

## Is there a readme badge?

---

Yes! If you use `standard` in your project, you can include one of these badges in your readme to let people know that your code is using the standard style.



```
[![JavaScript Style Guide](https://cdn.rawgit.com/standard/standard/master/badge
```



code style **standard**

```
[![JavaScript Style Guide](https://img.shields.io/badge/code_style-standard-brightgreen
```



## I disagree with rule X, can you change it?

---

No. The whole point of `standard` is to save you time by avoiding [bikeshedding](#) about code style. There are lots of debates online about tabs vs. spaces, etc. that will never be resolved. These debates just distract from getting stuff done. At the end of the day you have to 'just pick something', and that's the whole philosophy of `standard` -- it's a bunch of sensible 'just pick something' opinions. Hopefully, users see the value in that over defending their own opinions.

There are a couple of similar packages for anyone who does not want to completely accept `standard` :

- [semistandard](#) - standard, with semicolons
- [standardx](#) - standard, with custom tweaks



If you really want to configure hundreds of ESLint rules individually, you can always use `eslint` [eslint-config-standard](#) to layer your changes on top. [standard-eject](#) can help you migrate from `standard` to `eslint` and `eslint-config-standard`

Pro tip: Just use `standard` and move on. There are actual real problems that you could spend your time solving! :P

## But this isn't a real web standard!

---

Of course it's not! The style laid out here is not affiliated with any official web standards groups, which is why this repo is called `standard/standard` and not `ECMA/standard`.

The word "standard" has more meanings than just "web standard" :-). For example:

- This module helps hold our code to a high *standard of quality*.
- This module ensures that new contributors follow some basic *style standards*.

## Is there an automatic formatter?

---

Yes! You can use `standard --fix` to fix most issues automatically.

`standard --fix` is built into `standard` for maximum convenience. Most problems are fixable, but some errors (like forgetting to handle errors) must be fixed manually.

To save you time, `standard` outputs the message "Run `standard --fix` to automatically fix some problems" when it detects problems that can be fixed automatically.

## How do I ignore files?

---

Certain paths (`node_modules/`, `coverage/`, `vendor/`, `*.min.js`, and files/folders that begin with `.` like `.git/`) are automatically ignored.

Paths in a project's root `.gitignore` file are also automatically ignored.

Sometimes you need to ignore additional folders or specific minified files. To do that, add a `standard.ignore` property to `package.json`:

```
"standard": {  
  "ignore": [  
    "**/out/",  
    "/lib/select2/",  
    "/lib/ckeditor/",
```

```
"tmp.js"  
]  
}
```

## How do I disable a rule?

---

In rare cases, you'll need to break a rule and hide the error generated by `standard`.

JavaScript Standard Style uses [ESLint](#) under-the-hood and you can hide errors as you normally would if you used ESLint directly.

Disable **all** rules on a specific line:

```
file = 'I know what I am doing' // eslint-disable-line
```

Or, disable **only** "no-use-before-define" rule:

```
file = 'I know what I am doing' // eslint-disable-line no-use-before-define
```

Or, disable the "no-use-before-define" rule for **multiple lines**:

```
/* eslint-disable no-use-before-define */  
console.log('offending code goes here...')  
console.log('offending code goes here...')  
console.log('offending code goes here...')  
/* eslint-enable no-use-before-define */
```

## I use a library that pollutes the global namespace. How do I prevent "variable is not defined" errors?

---

Some packages (e.g. `mocha`) put their functions (e.g. `describe`, `it`) on the global object (poor form!). Since these functions are not defined or `require`'d anywhere in your code, `standard` will warn that you're using a variable that is not defined (usually, this rule is really useful for catching typos!). But we want to disable it for these global variables.

To let `standard` (as well as humans reading your code) know that certain variables are global in your code, add this to the top of your file:

```
/* global myVar1, myVar2 */
```

If you have hundreds of files, it may be desirable to avoid adding comments to every file. In this case, run:

```
$ standard --global myVar1 --global myVar2
```

Or, add this to `package.json` :

```
{
  "standard": {
    "globals": [ "myVar1", "myVar2" ]
  }
}
```

*Note: `global` and `globals` are equivalent.*

## How do I use experimental JavaScript (ES Next) features?

---

`standard` supports the latest ECMAScript features, ES8 (ES2017), including language feature proposals that are in "Stage 4" of the proposal process.

To support experimental language features, `standard` supports specifying a custom JavaScript parser. Before using a custom parser, consider whether the added complexity is worth it.

To use a custom parser, first install it from npm:

```
npm install @babel/eslint-parser --save-dev
```

Then run:

```
$ standard --parser @babel/eslint-parser
```

Or, add this to `package.json` :

```
{
  "standard": {
    "parser": "@babel/eslint-parser"
  }
}
```

# Can I use a JavaScript language variant, like Flow or TypeScript?

---

`standard` supports the latest ECMAScript features. However, Flow and TypeScript add new syntax to the language, so they are not supported out-of-the-box.

For TypeScript, an official variant `ts-standard` is supported and maintained that provides a very similar experience to `standard`.

For other JavaScript language variants, `standard` supports specifying a custom JavaScript parser as well as an ESLint plugin to handle the changed syntax. Before using a JavaScript language variant, consider whether the added complexity is worth it.

## TypeScript

`ts-standard` is the officially supported variant for TypeScript. `ts-standard` supports all the same rules and options as `standard` and includes additional TypeScript specific rules. `ts-standard` will even lint regular `javascript` `tsconfig.json`

```
npm install ts-standard --save-dev
```

Then run (where `tsconfig.json` is located in the working directory):

```
$ ts-standard
```

Or, add this to `package.json` :

```
{
  "ts-standard": {
    "project": "./tsconfig.json"
  }
}
```

*Note: To include additional files in linting such as test files, create a `tsconfig.eslint.json` file to use instead.*

If you really want to configure hundreds of ESLint rules individually, you can always use eslint directly with `eslint-config-standard-with-typescript` to layer your changes on top.

## Flow

To use Flow, you need to run `standard @babel/eslint-parser` as the parser and `eslint-plugin-flowtype` as a plugin.

```
npm install @babel/eslint-parser eslint-plugin-flowtype --save-dev
```

Then run:

```
$ standard --parser @babel/eslint-parser --plugin flowtype
```

Or, add this to `package.json` :

```
{
  "standard": {
    "parser": "@babel/eslint-parser",
    "plugins": [ "flowtype" ]
  }
}
```

*Note: `plugin` and `plugins` are equivalent.*

## What about Mocha, Jest, Jasmine, QUnit, etc?

---

To support mocha in test files, add this to the top of the test files:

```
/* eslint-env mocha */
```

Or, run:

```
$ standard --env mocha
```

Where `mocha` can be one of `jest`, `jasmine`, `qunit`, `phantomjs`, and so on. To see a full list, check ESLint's [specifying environments](#) documentation. For a list of what globals are available for these environments, check the [globals](#) npm module.

*Note: `env` and `envs` are equivalent.*

## What about Web Workers and Service Workers?

---

Add this to the top of web worker files:

```
/* eslint-env worker */
```

This lets `standard` (as well as humans reading the code) know that `self` is a global in web worker code.

For Service workers, add this instead:

```
/* eslint-env serviceworker */
```

## What is the difference between warnings and errors?

---

`standard` treats all rule violations as errors, which means that `standard` will exit with a non-zero (error) exit code.

However, we may occasionally release a new major version of `standard` which changes a rule that affects the majority of `standard` users (for example, transitioning from `var` `let` / `const` ). We do this only when we think the advantage is worth the cost and only when the rule is [auto-fixable](#).

In these situations, we have a "transition period" where the rule change is only a "warning". Warnings don't cause `standard` to return a non-zero (error) exit code. However, a warning message will still print to the console. During the transition period, using `standard --fix` will update your code so that it's ready for the next major version.

The slow and careful approach is what we strive for with `standard` . We're generally extremely conservative in enforcing the usage of new language features. We want using `standard` to be light and fun and so we're careful about making changes that may get in your way. As always, you can [disable a rule](#) at any time, if necessary.

## Can I check code inside of Markdown or HTML files?

---

To check code inside Markdown files, use [standard-markdown](#) .

Alternatively, there are ESLint plugins that can check code inside Markdown, HTML, and many other types of language files:

To check code inside Markdown files, use an ESLint plugin:

```
$ npm install eslint-plugin-markdown
```

Then, to check JS that appears inside code blocks, run:

```
$ standard --plugin markdown '**/*.md'
```

To check code inside HTML files, use an ESLint plugin:

```
$ npm install eslint-plugin-html
```

Then, to check JS that appears inside `<script>` tags, run:

```
$ standard --plugin html '**/*.html'
```

## Is there a Git pre-commit hook?

☰ README.md

---

repo. Never give style feedback on a pull request again!

You even have a choice...

### Install your own hook

JavaScript files staged for commit pass standard code style

```
r() {
  rsion of "xargs -r". The -r flag is a GNU extension that
  rgs from running if there are no input files.
```

```
-r -d '$\n' path; then
```

```
" | cat - | xargs "$@"
```

```
-only --cached --relative | grep '\.jsx\?$$' | sed 's/^[^[:alnum:]]/\&/g' | xargs-r
]]; then
```

```
ipt Standard Style errors were detected. Aborting commit.'
```

### Use a pre-commit hook

The [pre-commit](#) library allows hooks to be declared within a `.pre-commit-config.yaml` configuration file in the repo, and therefore more easily maintained across a team.

Users of pre-commit can simply add `standard` to their `.pre-commit-config.yaml` file, which will automatically fix `.js`, `.jsx`, `.mjs` and `.cjs` files:

```
- repo: https://github.com/standard/standard
  rev: master
  hooks:
    - id: standard
```

Alternatively, for more advanced styling configurations, use `standard` within the `eslint` hook:

```
- repo: https://github.com/pre-commit/mirrors-eslint
  rev: master
  hooks:
    - id: eslint
      files: \.[jt]sx?$ # *.js, *.jsx, *.ts and *.tsx
      types: [file]
      additional_dependencies:
        - eslint@latest
        - eslint-config-standard@latest
        # and whatever other plugins...
```

## How do I make the output all colorful and pretty?

---

The built-in output is simple and straightforward, but if you like shiny things, install `snazzy`:

```
$ npm install snazzy
```

And run:

```
$ standard | snazzy
```

There's also `standard-tap`, `standard-json`, `standard-reporter`, and `standard-summary`.

## Is there a Node.js API?

---

Yes!

```
async standard.lintText(text, [opts])
```

Lint the provided source `text`. An `opts` object may be provided:

```
{
  // unique to lintText
```



```

filename: '',          // path of file containing the text being linted

// common to lintText and lintFiles
cwd: '',              // current working directory (default: process.cwd())
fix: false,           // automatically fix problems
extensions: [],       // file extensions to lint (has sane defaults)
globals: [],          // custom global variables to declare
plugins: [],          // custom eslint plugins
envs: [],             // custom eslint environment
parser: '',           // custom js parser (e.g. babel-eslint)
usePackageJson: true, // use options from nearest package.json?
useGitIgnore: true    // use file ignore patterns from .gitignore?
}

```

All options are optional, though some ESLint plugins require the `filename` option.

Additional options may be loaded from a `package.json` if it's found for the current working directory. See below for further details.

Returns a `Promise` resolving to the `results` or rejected with an `Error`.

The `results` object will contain the following properties:

```

const results = {
  results: [
    {
      filePath: '',
      messages: [
        { ruleId: '', message: '', line: 0, column: 0 }
      ],
      errorCount: 0,
      warningCount: 0,
      output: '' // fixed source code (only present with {fix: true} option)
    }
  ],
  errorCount: 0,
  warningCount: 0
}

```

## async standard.lintFiles(files, [opts])

Lint the provided `files` globs. An `opts` object may be provided:

```

{
  // unique to lintFiles
  ignore: [],          // file globs to ignore (has sane defaults)

  // common to lintText and lintFiles

```

```

cwd: '',           // current working directory (default: process.cwd())
fix: false,        // automatically fix problems
extensions: [],    // file extensions to lint (has sane defaults)
globals: [],       // custom global variables to declare
plugins: [],       // custom eslint plugins
envs: [],          // custom eslint environment
parser: '',        // custom js parser (e.g. babel-eslint)
usePackageJson: true, // use options from nearest package.json?
useGitIgnore: true  // use file ignore patterns from .gitignore?
}

```

Additional options may be loaded from a `package.json` if it's found for the current working directory. See below for further details.

Both `ignore` and `files` patterns are resolved relative to the current working directory.

Returns a `Promise` resolving to the `results` or rejected with an `Error` (same as above).

## How do I contribute to StandardJS?

---

Contributions are welcome! Check out the [issues](#) or the [PRs](#), and make your own if you want something that you don't see there.

Want to chat? Join contributors on IRC in the `#standard` channel on freenode.

Here are some important packages in the `standard` ecosystem:

- **standard** - this repo
  - **standard-engine** - cli engine for arbitrary eslint rules
  - **eslint-config-standard** - eslint rules for standard
  - **eslint-config-standard-jsx** - eslint rules for standard (JSX)
  - **eslint** - the linter that powers standard
- **snazzy** - pretty terminal output for standard
- **standard-www** - code for <https://standardjs.com>
- **semistandard** - standard, with semicolons (if you must)
- **standardx** - standard, with custom tweaks

There are also many [editor plugins](#), a list of [npm packages that use standard](#), and an awesome list of [packages in the standard ecosystem](#).

## Security Policies and Procedures

---

The `standard` team and community take all security bugs in `standard` seriously. Please see our [security policies and procedures](#) document to learn how to report issues.

# License

MIT. Copyright (c) [Feross Aboukhadijeh](#).

## Releases 4

 **v17.0.0** Latest  
on Apr 20, 2022

[+ 3 releases](#)

## Sponsor this project



feross Feross Aboukhadijeh





standard Standard JS





[tidelift.com/funding/github/npm/standard](https://tidelift.com/funding/github/npm/standard)

[Learn more about GitHub Sponsors](#)

## Packages

No packages published

## Used by 195k

 + 194,620

## Contributors 175



[+ 164 contributors](#)

## Languages

● JavaScript 81.9%   ● Shell 18.1%