

# Configuration Files



## Tip

This page explains how to use flat config files. For the deprecated eslintrc format, [\*\*see the deprecated documentation\*\*](#).

You can put your ESLint project configuration in a configuration file. You can include built-in rules, how you want them enforced, plugins with custom rules, shareable configurations, which files you want rules to apply to, and more.

## Configuration File

The ESLint configuration file may be named any of the following:

- `eslint.config.js`
- `eslint.config.mjs`
- `eslint.config.cjs`
- `eslint.config.ts` (requires **additional setup**)
- `eslint.config.mts` (requires **additional setup**)
- `eslint.config.cts` (requires **additional setup**)

It should be placed in the root directory of your project and export an array of **configuration objects**. Here's an example:

```
1 // eslint.config.js
2 import { defineConfig } from "eslint/config";
3
```

```

4
5   export default defineConfig([
6     {
7       rules: {
8         semi: "error",
9         "prefer-const": "error",
10      },
11    },
12  ]);

```

In this example, the `defineConfig()` helper is used to define a configuration array with just one configuration object. The configuration object enables two rules: `semi` and `prefer-const`. These rules are applied to all of the files ESLint processes using this config file.

If your project does not specify `"type": "module"` in its `package.json` file, then `eslint.config.js` must be in CommonJS format, such as:

```

1   // eslint.config.js
2   const { defineConfig } = require("eslint/config");
3
4   module.exports = defineConfig([
5     {
6       rules: {
7         semi: "error",
8         "prefer-const": "error",
9       },
10    },
11  ]);

```

## Configuration Objects

Each configuration object contains all of the information ESLint needs to execute on a set of files. Each configuration object is made up of these properties:

- `name` - A name for the configuration object. This is used in error messages and config inspector to help identify which configuration object is being used. (**Naming Convention**)
- `files` - An array of glob patterns indicating the files that the configuration object should apply to. If not specified, the configuration object applies to all files matched by any other configuration object.
- `ignores` - An array of glob patterns indicating the files that the configuration object should not apply to. If not specified, the configuration object applies to all files matched by `files`. If `ignores` is used without any other keys in the configuration object, then the patterns act as **global ignores** and it gets applied to every configuration object.
- `extends` - An array of strings, configuration objects, or configuration arrays that contain additional configuration to apply.
- `languageOptions` - An object containing settings related to how JavaScript is configured for linting.
  - `ecmaVersion` - The version of ECMAScript to support. May be any year (i.e., 2022) or version (i.e., 5). Set to `"latest"` for the most recent supported version. (default: `"latest"`)
  - `sourceType` - The type of JavaScript source code. Possible values are `"script"` for traditional script files, `"module"` for ECMAScript modules (ESM), and `"commonjs"` for CommonJS files. (default: `"module"` for `.js` and `.mjs` files; `"commonjs"` for `.cjs` files)
  - `globals` - An object specifying additional objects that should be added to the global scope during linting.
  - `parser` - An object containing a `parse()` method or a `parseForESLint()` method. (default: [espreet](https://github.com/eslint/js/tree/main/packages/espreet) (<https://github.com/eslint/js/tree/main/packages/espreet>))
  - `parserOptions` - An object specifying additional options that are passed directly to the `parse()` or `parseForESLint()` method on the parser. The available options are parser-dependent.
- `linterOptions` - An object containing settings related to the linting process.

- `noInlineConfig` - A Boolean value indicating if inline configuration is allowed.
- `reportUnusedDisableDirectives` - A severity string indicating if and how unused disable and enable directives should be tracked and reported. For legacy compatibility, `true` is equivalent to `"warn"` and `false` is equivalent to `"off"`. (default: `"warn"`).
- `reportUnusedInlineConfigs` - A severity string indicating if and how unused inline configs should be tracked and reported. (default: `"off"`)
- `processor` - Either an object containing `preprocess()` and `postprocess()` methods or a string indicating the name of a processor inside of a plugin (i.e., `"pluginName/processorName"`).
- `plugins` - An object containing a name-value mapping of plugin names to plugin objects. When `files` is specified, these plugins are only available to the matching files.
- `rules` - An object containing the configured rules. When `files` or `ignores` are specified, these rule configurations are only available to the matching files.
- `settings` - An object containing name-value pairs of information that should be available to all rules.

## Specifying `files` and `ignores`

### ⓘ Tip

Patterns specified in `files` and `ignores` use [minimatch](https://www.npmjs.com/package/minimatch) (<https://www.npmjs.com/package/minimatch>) syntax and are evaluated relative to the location of the `eslint.config.js` file. If using an alternate config file via the `--config` command line option, then all patterns are evaluated relative to the current working directory.

You can use a combination of `files` and `ignores` to determine which files the configuration object should apply to and which not. By default, ESLint lints files that match the patterns `**/*.js`, `**/*.cjs`, and `**/*.mjs`. Those files are always matched unless you explicitly exclude them using **global ignores**. Because

config objects that don't specify `files` or `ignores` apply to all files that have been matched by any other configuration object, they will apply to all JavaScript files. For example:

```
1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3
4  export default defineConfig([
5    {
6      rules: {
7        semi: "error",
8      },
9    },
10  ]);
```

With this configuration, the `semi` rule is enabled for all files that match the default files in ESLint. So if you pass `example.js` to ESLint, the `semi` rule is applied. If you pass a non-JavaScript file, like `example.txt`, the `semi` rule is not applied because there are no other configuration objects that match that filename. (ESLint outputs an error message letting you know that the file was ignored due to missing configuration.)

## Excluding files with `ignores`

You can limit which files a configuration object applies to by specifying a combination of `files` and `ignores` patterns. For example, you may want certain rules to apply only to files in your `src` directory:

```
1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3
4  export default defineConfig([
5    {
6      files: ["src/**/*.js"],
7    },
8  ]);
```

```

7         rules: {
8             semi: "error",
9         },
10     },
11 ];

```

Here, only the JavaScript files in the `src` directory have the `semi` rule applied. If you run ESLint on files in another directory, this configuration object is skipped. By adding `ignores`, you can also remove some of the files in `src` from this configuration object:

```

1  import { defineConfig } from "eslint/config";
2
3  export default defineConfig([
4      {
5          files: ["src/**/*.js"],
6          ignores: ["**/*.config.js"],
7          rules: {
8              semi: "error",
9          },
10     },
11 ]);

```

This configuration object matches all JavaScript files in the `src` directory except those that end with `.config.js`. You can also use negation patterns in `ignores` to exclude files from the ignore patterns, such as:

```

import { defineConfig } from "eslint/config";

export default defineConfig([
    {
        files: ["src/**/*.js"],
        ignores: ["**/*.config.js", "!**/eslint.con",
        rules: {
            semi: "error",

```

```

9         },
10     },
11 ];

```

Here, the configuration object excludes files ending with `.config.js` except for `eslint.config.js`. That file still has `semi` applied.

Non-global `ignores` patterns can only match file names. A pattern like `"dir-to-exclude/"` will not ignore anything. To ignore everything in a particular directory, a pattern like `"dir-to-exclude/**"` should be used instead.

If `ignores` is used without `files` and there are other keys (such as `rules`), then the configuration object applies to all linted files except the ones excluded by `ignores`, for example.

```

1  import { defineConfig } from "eslint/config";
2
3  export default defineConfig([
4      {
5          ignores: ["**/*.config.js"],
6          rules: {
7              semi: "error",
8          },
9      },
10 ]);

```

This configuration object applies to all JavaScript files except those ending with `.config.js`. Effectively, this is like having `files` set to `**/*`. In general, it's a good idea to always include `files` if you are specifying `ignores`.

Note that when `files` is not specified, negated `ignores` patterns do not cause any matching files to be linted automatically. ESLint only lints files that are matched either by default or by a `files` pattern that is not `*` and does not end with `/*` or `**`.



Tip

Use the **config inspector** (<https://github.com/eslint/config-inspector>) (`--inspect-config` in the CLI) to test which config objects apply to a specific file.

## Specifying files with arbitrary extensions

To lint files with extensions other than the default `.js`, `.cjs` and `.mjs`, include them in `files` with a pattern in the format of `"**/*.extension"`. Any pattern will work except if it is `*` or if it ends with `/*` or `**`. For example, to lint TypeScript files with `.ts`, `.cts` and `.mts` extensions, you would specify a configuration object like this:

```
1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3
4  export default defineConfig([
5    {
6      files: ["**/*.ts", "**/*.cts", "**/*.mts"],
7    },
8    // ...other config
9  ]);
```

## Specifying files without extension

Files without an extension can be matched with the pattern `!(*)`. For example:

```
1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3
4  export default defineConfig([
5    {
6      files: ["**/!(*)"],
7    },
8  ]);
```



```
8      },
9      // ...other config
    });
```

The above config lints files without extension besides the default `.js`, `.cjs` and `.mjs` extensions in all directories.

ⓘ Tip

Filenames starting with a dot, such as `.gitignore`, are considered to have only an extension without a base name. In the case of `.gitignore`, the extension is `gitignore`, so the file matches the pattern `**/.gitignore` but not `**/*.gitignore`.

## Globally ignoring files with `ignores`

Depending on how the `ignores` property is used, it can behave as non-global `ignores` or as global `ignores`.

- When `ignores` is used without any other keys (besides `name`) in the configuration object, then the patterns act as global `ignores`. This means they apply to every configuration object (not only to the configuration object in which it is defined). Global `ignores` allows you not to have to copy and keep the `ignores` property synchronized in more than one configuration object.
- If `ignores` is used with other properties in the same configuration object, then the patterns act as non-global `ignores`. This way `ignores` applies only to the configuration object in which it is defined.

Global and non-global `ignores` have some usage differences:

- patterns in non-global `ignores` only match the files (`dir/filename.js`) or files within directories (`dir/**`)
- patterns in global `ignores` can match directories (`dir/`) in addition to the patterns that non-global `ignores` supports.

For all uses of `ignores`:

- The patterns you define are added after the default ESLint patterns, which are `["**/node_modules/", ".git/"]`.
- The patterns always match files and directories that begin with a dot, such as `.foo.js` or `.fixtures`, unless those files are explicitly ignored. The only dot directory ignored by default is `.git`.

```

1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3
4  // Example of global ignores
5  export default defineConfig([
6    {
7      ignores: [".config/", "dist/", "tsconfig.json"],
8    },
9    { ... }, // ... other configuration object, inheriting
10   { ... }, // ... other configuration object inheriting
11  ]);
12
13 // Example of non-global ignores
14 export default defineConfig([
15   {
16     ignores: [".config/**", "dir1/script1.js"],
17     rules: { ... } // the presence of this property
18   },
19   {
20     ignores: ["other-dir/**", "dist/script2.js"],
21     rules: { ... } // the presence of this property
22   },
23  ]);

```

To avoid confusion, use the `globalIgnores()` helper function to clearly indicate which ignores are meant to be global. Here's the previous example rewritten to use `globalIgnores()`:

```

1  // eslint.config.js
2  import { defineConfig, globalIgnores } from "eslint/
3
4  // Example of global ignores
5  export default defineConfig([
6      globalIgnores([".config/", "dist/", "tsconfig.js
7      { ... }, // ... other configuration object, inher
8      { ... }, // ... other configuration object inher
9  ]);
10
11 // Example of non-global ignores
12 export default defineConfig([
13     {
14         ignores: [".config/**", "dir1/script1.js"],
15         rules: { ... } // the presence of this proper
16     },
17     {
18         ignores: ["other-dir/**", "dist/script2.js"],
19         rules: { ... } // the presence of this proper
20     },
21 ]);

```

For more information and examples on configuring rules regarding `ignores`, see [\*\*Ignore Files\*\*](#).

## Cascading Configuration Objects

When more than one configuration object matches a given filename, the configuration objects are merged with later objects overriding previous objects when there is a conflict. For example:

```

1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3

```

```

4
5   export default defineConfig([
6       {
7           files: ["**/*.js"],
8           languageOptions: {
9               globals: {
10                  MY_CUSTOM_GLOBAL: "readonly",
11              },
12          },
13      },
14      {
15          files: ["tests/**/*.js"],
16          languageOptions: {
17              globals: {
18                  it: "readonly",
19                  describe: "readonly",
20              },
21          },
22      },
23  ]);

```

Using this configuration, all JavaScript files define a custom global object defined called `MY_CUSTOM_GLOBAL` while those JavaScript files in the `tests` directory have `it` and `describe` defined as global objects in addition to `MY_CUSTOM_GLOBAL`. For any JavaScript file in the `tests` directory, both configuration objects are applied, so `languageOptions.globals` are merged to create a final result.

## Configuring Linter Options

Options specific to the linting process can be configured using the `linterOptions` object. These effect how linting proceeds and does not affect how the source code of the file is interpreted.

## Disabling Inline Configuration

Inline configuration is implemented using an `/*eslint*/` comment, such as `/*eslint semi: error*/`. You can disallow inline configuration by setting `noInlineConfig` to `true`. When enabled, all inline configuration is ignored. Here's an example:

```
1  // eslint.config.js
2  import { defineConfig } from "eslint/config";
3
4  export default defineConfig([
5    {
6      files: ["**/*.js"],
7      linterOptions: {
8        noInlineConfig: true,
9      },
10   },
11  ]);
```

## Reporting Unused Disable Directives

Disable and enable directives such as `/*eslint-disable*/`, `/*eslint-enable*/` and `/*eslint-disable-next-line*/` are used to disable ESLint rules around certain portions of code. As code changes, it's possible for these directives to no longer be needed because the code has changed in such a way that the rule is no longer triggered. You can enable reporting of these unused disable directives by setting the `reportUnusedDisableDirectives` option to a severity string, as in this example:

```
// eslint.config.js
import { defineConfig } from "eslint/config";

export default defineConfig([
  {
```

```

6         files: ["**/*.js"],
7         linterOptions: {
8             reportUnusedDisableDirectives: "error",
9         },
10    },
11    });

```

This setting defaults to "warn".

You can override this setting using the `--report-unused-disable-directives` or the `--report-unused-disable-directives-severity` command line options.

For legacy compatibility, `true` is equivalent to "warn" and `false` is equivalent to "off".

## Reporting Unused Inline Configs

Inline config comments such as `/* eslint rule-name: "error" */` are used to change ESLint rule severity and/or options around certain portions of code. As a project's ESLint configuration file changes, it's possible for these directives to no longer be different from what was already set. You can enable reporting of these unused inline config comments by setting the `reportUnusedInlineConfigs` option to a severity string, as in this example:

```

// eslint.config.js
import { defineConfig } from "eslint/config";

export default defineConfig([
  {
    files: ["**/*.js"],
    linterOptions: {
      reportUnusedInlineConfigs: "error",
    },
  },
]);

```

```
]);
```

You can override this setting using the `--report-unused-inline-configs` command line option.

## Configuring Rules

You can configure any number of rules in a configuration object by add a `rules` property containing an object with your rule configurations. The names in this object are the names of the rules and the values are the configurations for each of those rules. Here's an example:

```
1 // eslint.config.js
2 import { defineConfig } from "eslint/config";
3
4 export default defineConfig([
5   {
6     rules: {
7       semi: "error",
8     },
9   },
10 ]);
```

This configuration object specifies that the `semi` rule should be enabled with a severity of `"error"`. You can also provide options to a rule by specifying an array where the first item is the severity and each item after that is an option for the rule. For example, you can switch the `semi` rule to disallow semicolons by passing `"never"` as an option:

```
1 // eslint.config.js
2 import { defineConfig } from "eslint/config";
3
4 export default defineConfig([
5   {
```

```

6         rules: {
7             semi: ["error", "never"],
8         },
9     },
10 ];

```

Each rule specifies its own options and can be any valid JSON data type. Please check the documentation for the rule you want to configure for more information about its available options.

For more information on configuring rules, see [Configure Rules](#).

## Configuring Shared Settings

ESLint supports adding shared settings into configuration files. When you add a `settings` object to a configuration object, it is supplied to every rule. By convention, plugins namespace the settings they are interested in to avoid collisions with others. Plugins can use `settings` to specify the information that should be shared across all of their rules. This may be useful if you are adding custom rules and want them to have access to the same information. Here's an example:

```

// eslint.config.js
import { defineConfig } from "eslint/config";

export default defineConfig([
  {
    settings: {
      sharedData: "Hello",
    },
    plugins: {
      customPlugin: {
        rules: {
          "my-rule": {
            meta: {
              // custom rule's meta in

```



```

15         },
16         create(context) {
17             const sharedData = context;
18             return {
19                 // code
20             };
21         },
22     },
23 },
24 },
25 },
26 rules: {
27     "customPlugin/my-rule": "error",
28 },
29 },
30 ]);

```

## Extending Configurations

A configuration object uses `extends` to inherit all the traits of another configuration object or array (including rules, plugins, and language options) and can then modify all the options. The `extends` key is an array of values indicating which configurations to extend from. The elements of the `extends` array can be one of three values:


- a string that specifies the name of a configuration in a plugin
- a configuration object
- a configuration array



## Using Configurations from Plugins

ESLint plugins can export predefined configurations. These configurations are referenced using a string and follow the pattern `pluginName/configName`. The plugin must be specified in the `plugins` key first. Here's an example:

```
1  // eslint.config.js
2  import examplePlugin from "eslint-plugin-example";
3  import { defineConfig } from "eslint/config";
4
5  export default defineConfig([
6    {
7      files: ["**/*.js"],
8      plugins: {
9        example: examplePlugin,
10     },
11     extends: ["example/recommended"],
12   },
13   ]);
```



In this example, the configuration named `recommended` from `eslint-plugin-example` is loaded. The plugin configurations can also be referenced by name inside of the configuration array.

You can also insert plugin configurations directly into the `extends` array. For example:

```
// eslint.config.js
import pluginExample from "eslint-plugin-example";
import { defineConfig } from "eslint/config";

export default defineConfig([
  {
    files: ["**/*.js"],
```

```

8         plugins: {
9             example: pluginExample,
10         },
11         extends: [pluginExample.configs.recommended
12     ],
13     });

```

In this case, the configuration named `recommended` from `eslint-plugin-example` is accessed directly through the plugin object's `configs` property.



#### Important

It's recommended to always use a `files` key when you use the `extends` key to ensure that your configuration applies to the correct files. By omitting the `files` key, the extended configuration may end up applied to all files.

## Using Predefined Configurations

ESLint has two predefined configurations for JavaScript:

- `js/recommended` - enables the rules that ESLint recommends everyone use to avoid potential errors.
- `js/all` - enables all of the rules shipped with ESLint. This configuration is **not recommended** for production use because it changes with every minor and major version of ESLint. Use at your own risk.

To include these predefined configurations, install the `@eslint/js` package and then make any modifications to other properties in subsequent configuration objects:

```

1  // eslint.config.js
2  import js from "@eslint/js";
3  import { defineConfig } from "eslint/config";
4
5  export default defineConfig([
6      {
7
8      }
9  ]);

```

```

7         files: ["**/*.js"],
8         plugins: {
9             js,
10        },
11        extends: ["js/recommended"],
12        rules: {
13            "no-unused-vars": "warn",
14        },
15    },
16    });

```

Here, the `js/recommended` predefined configuration is applied first and then another configuration object adds the desired configuration for `no-unused-vars`.

For more information on how to combine predefined configs with your preferences, please see [Combine Configs](#).

## Using a Shareable Configuration Package

A sharable configuration is an npm package that exports a configuration object or array. This package should be installed as a dependency in your project and then referenced from inside of your `eslint.config.js` file. For example, to use a shareable configuration named `eslint-config-example`, your configuration file would look like this:

```

// eslint.config.js
import exampleConfig from "eslint-config-example";
import { defineConfig } from "eslint/config";

export default defineConfig([
    {
        files: ["**/*.js"],
        extends: [exampleConfig],
        rules: {
            "no-unused-vars": "warn",
        },
    },
]);

```

```
12     },
13   ] );
```

In this example, `exampleConfig` can be either an object or an array, and either way it can be inserted directly into the `extends` array.

For more information on how to combine shareable configs with your preferences, please see [Combine Configs](#).

## Configuration Naming Conventions

The `name` property is optional, but it is recommended to provide a name for each configuration object, especially when you are creating shared configurations. The name is used in error messages and the config inspector to help identify which configuration object is being used.

The name should be descriptive of the configuration object's purpose and scoped with the configuration name or plugin name using `/` as a separator. ESLint does not enforce the names to be unique at runtime, but it is recommended that unique names be set to avoid confusion.

For example, if you are creating a configuration object for a plugin named `eslint-plugin-example`, you might add `name` to the configuration objects with the `example/` prefix:

```
1   export default {
2     configs: {
3       recommended: {
4         name: "example/recommended",
5         rules: {
6           "no-unused-vars": "warn",
7         },
8       },
9       strict: {
10        name: "example/strict",
11      }
```

```

12         rules: {
13             "no-unused-vars": "error",
14         },
15     },
16 },
};

```

When exposing arrays of configuration objects, the `name` may have extra scoping levels to help identify the configuration object. For example:

```

1  export default {
2      configs: {
3          strict: [
4              {
5                  name: "example/strict/language-setup",
6                  languageOptions: {
7                      ecmaVersion: 2024,
8                  },
9              },
10             {
11                 name: "example/strict/sub-config",
12                 file: ["src/**/*.js"],
13                 rules: {
14                     "no-unused-vars": "error",
15                 },
16             },
17         ],
18     },
19 };

```



## Configuration File Resolution

When ESLint is run on the command line, it first checks the current working directory for `eslint.config.js`. If that file is found, then the search stops, otherwise it checks for `eslint.config.mjs`. If that file is found, then the search stops, otherwise it checks for `eslint.config.cjs`. If none of the files are found, it checks the parent directory for each file. This search continues until either a config file is found or the root directory is reached.

You can prevent this search for `eslint.config.js` by using the `-c` or `--config` option on the command line to specify an alternate configuration file, such as:

npm      yarn      pnpm      bun

```
1      npx eslint --config some-other-file.js **/*.js
```

In this case, ESLint does not search for `eslint.config.js` and instead uses `some-other-file.js`.

## Experimental Configuration File Resolution

### ⚠ Warning

This feature is experimental and its details may change before being finalized. This behavior will be the new lookup behavior starting in v10.0.0, but you can try it today using a feature flag.

You can use the `unstable_config_lookup_from_file` flag to change the way ESLint searches for configuration files. Instead of searching from the current working directory, ESLint will search for a configuration file by first starting in the directory of the file being linted and then searching up its ancestor directories until it finds a `eslint.config.js` file (or any other extension of configuration file). This behavior is better for monorepos, where each subdirectory may have its own configuration file.

To use this feature on the command line, use the `--flag` flag:

```
1 npx eslint --flag unstable_config_lookup_from_file
```

For more information about using feature flags, see **Feature Flags**.

## TypeScript Configuration Files

For Deno and Bun, TypeScript configuration files are natively supported; for Node.js, you must install the optional dev dependency [jiti](https://github.com/unjs/jiti) (<https://github.com/unjs/jiti>) in version 2.0.0 or later in your project (this dependency is not automatically installed by ESLint):

npm    yarn    pnpm    bun

```
1 npm install --save-dev jiti
```

You can then create a configuration file with a `.ts`, `.mts`, or `.cts` extension, and export an array of **configuration objects**.



### Important

ESLint does not perform type checking on your configuration file and does not apply any settings from `tsconfig.json`.

## Configuration File Precedence

If you have multiple ESLint configuration files, ESLint prioritizes JavaScript files over TypeScript files. The order of precedence is as follows:

1. `eslint.config.js`
2. `eslint.config.mjs`
3. `eslint.config.cjs`



4. `eslint.config.ts`

5. `eslint.config.mts`

6. `eslint.config.cts`

To override this behavior, use the `--config` or `-c` command line option to specify a different configuration file:

`npm`     `yarn`     `pnpm`     `bun`

```
1  npx eslint --config eslint.config.ts
```