Language Options

Specifying Environments

An environment provides predefined global variables. The available environments are:

- `browser` - browser global variables.
- `node` - Node.js global variables and Node.js scoping.
- `commonjs` - CommonJS global variables and CommonJS scoping (use this for browser-only code that uses Browserify/WebPack).
- `shared-node-browser` - Globals common to both Node.js and Browser.
- `es6` - enable all ECMAScript 6 features except for modules (this automatically sets the `ecmaVersion` parser option to 6).
- `es2016` - adds all ECMAScript 2016 globals and automatically sets the `ecmaVersion` parser option to 7.
- `es2017` - adds all ECMAScript 2017 globals and automatically sets the `ecmaVersion` parser option to 8.
- `es2018` - adds all ECMAScript 2018 globals and automatically sets the `ecmaVersion` parser option to 9.
- `es2019` - adds all ECMAScript 2019 globals and automatically sets the `ecmaVersion` parser option to 10.
- `es2020` - adds all ECMAScript 2020 globals and automatically sets the `ecmaVersion` parser option to 11.
- `es2021` - adds all ECMAScript 2021 globals and automatically sets the `ecmaVersion` parser option to 12.
- `es2022` - adds all ECMAScript 2022 globals and automatically sets the `ecmaVersion` parser option to 13.
- `worker` - web workers global variables.
- `amd` - defines `require()` and `define()` as global variables as per the amd spec.
- `mocha` - adds all of the Mocha testing global variables.
- `jasmine` - adds all of the Jasmine testing global variables for version 1.3 and 2.0.
- `jest` - Jest global variables.
- `phantomjs` - PhantomJS global variables.
- `protractor` - Protractor global variables.
- `qunit` - QUnit global variables.
- `jquery` - jQuery global variables.
- `prototypejs` - Prototype.js global variables.
- `shelljs` - ShellJS global variables.
- `meteor` - Meteor global variables.
- `mongo` - MongoDB global variables.
- `applescript` - AppleScript global variables.
- `nashorn` - Java 8 Nashorn global variables.
- `serviceworker` - Service Worker global variables.
- `atomtest` - Atom test helper globals.
- `embertest` - Ember test helper globals.
- `webextensions` - WebExtensions globals.
- `greasemonkey` - GreaseMonkey globals.

These environments are not mutually exclusive, so you can define more than one at a time.

Environments can be specified inside of a file, in configuration files or using the `--env` command line flag.

Using configuration comments

To specify environments using a comment inside of your JavaScript file, use the following format:

```
/* eslint-env node, mocha */
```

This enables Node.js and Mocha environments.

Using configuration files

To specify environments in a configuration file, use the `env` key and specify which environments you want to enable by setting each to `true`. For example, the following enables the browser and Node.js environments:

```
{
    "env": {
        "browser": true,
        "node": true
    }
}
```

Or in a `package.json` file

```
{
    "name": "mypackage",
    "version": "0.0.1",
    "eslintConfig": {
        "env": {
            "browser": true,
            "node": true
        }
    }
}
```

And in YAML:

```
---
  env:
    browser: true
    node: true
```

Using a plugin

If you want to use an environment from a plugin, be sure to specify the plugin name in the `plugins` array and then use the unprefixed plugin name, followed by a slash, followed by the environment name. For example:

```
{
    "plugins": ["example"],
    "env": {
        "example/custom": true
    }
}
```

Or in a `package.json` file

```
{
    "name": "mypackage",
    "version": "0.0.1",
    "eslintConfig": {
        "plugins": ["example"],
        "env": {
            "example/custom": true
        }
    }
}
```

Specifying Globals

Some of ESLint's core rules rely on knowledge of the global variables available to your code at runtime. Since these can vary greatly between different environments as well as be modified at runtime, ESLint makes no assumptions about what global variables exist in your execution environment. If you would like to use rules that require knowledge of what global variables are available, you can define global variables in your configuration file or by using configuration comments in your source code.

Using configuration comments

To specify globals using a comment inside of your JavaScript file, use the following format:

```
/* global var1, var2 */
```

This defines two global variables, `var1` and `var2`. If you want to optionally specify that these global variables can be written to (rather than only being read), then you can set each with a `"writable"` flag:

```
/* global var1:writable, var2:writable */
```

Using configuration files

To configure global variables inside of a configuration file, set the `globals` configuration property to an object containing keys named for each of the global variables you want to use. For each global variable key, set the corresponding value equal to `"writable"` to allow the variable to be overwritten or `"readonly"` to disallow overwriting. For example:

```
{
    "globals": {
        "var1": "writable",
        "var2": "readonly"
    }
}
```

And in YAML:

```
---
  globals:
    var1: writable
    var2: readonly
```

These examples allow `var1` to be overwritten in your code, but disallow it for `var2`.

Globals can be disabled with the string `"off"`. For example, in an environment where most ES2015 globals are available but `Promise` is unavailable, you might use this config:

```
{
    "env": {
        "es6": true
    },
    "globals": {
        "Promise": "off"
    }
}
```

For historical reasons, the boolean value `false` and the string value `"readable"` are equivalent to `"readonly"`. Similarly, the boolean value `true` and the string value `"writeable"` are equivalent to `"writable"`. However, the use of older values is deprecated.

Specifying Parser Options

ESLint allows you to specify the JavaScript language options you want to support. By default, ESLint expects ECMAScript 5 syntax. You can override that setting to enable support for other ECMAScript versions as well as JSX by using parser options.

Please note that supporting JSX syntax is not the same as supporting React. React applies specific semantics to JSX syntax that ESLint doesn't recognize. We recommend using eslint-plugin-react if you are using React and want React semantics. By the same token, supporting ES6 syntax is not the same as supporting new ES6 globals (e.g., new types such as `Set`). For ES6 syntax, use `{ "parserOptions": { "ecmaVersion": 6 } }`; for new ES6 global variables, use `{ "env": { "es6": true } }`. `{ "env": { "es6": true } }` enables ES6 syntax automatically, but `{ "parserOptions": { "ecmaVersion": 6 } }` does not enable ES6 globals automatically.

Parser options are set in your `.eslintrc.*` file by using the `parserOptions` property. The available options are:

- `ecmaVersion` - set to 3, 5 (default), 6, 7, 8, 9, 10, 11, 12, 13, or 14 to specify the version of ECMAScript syntax you want to use. You can also set to 2015 (same as 6), 2016 (same as 7), 2017 (same as 8), 2018 (same as 9), 2019 (same as 10), 2020 (same as 11), 2021 (same as 12), 2022 (same as 13), or 2023 (same as 14) to use the year-based naming. You can also set "latest" to use the most recently supported version.
- `sourceType` - set to `"script"` (default) or `"module"` if your code is in ECMAScript modules.
- `allowReserved` - allow the use of reserved words as identifiers (if `ecmaVersion` is 3).
- `ecmaFeatures` - an object indicating which additional language features you'd like to use:
  - `globalReturn` - allow `return` statements in the global scope
  - `impliedStrict` - enable global strict mode (if `ecmaVersion` is 5 or greater)
  - `jsx` - enable JSX

Here's an example `.eslintrc.json` file:

```
{
    "parserOptions": {
        "ecmaVersion": "latest",
        "sourceType": "module",
        "ecmaFeatures": {
            "jsx": true
        }
    },
    "rules": {
        "semi": "error"
    }
}
```

Setting parser options helps ESLint determine what is a parsing error. All language options are `false` by default.

Edit this page

Table of Contents