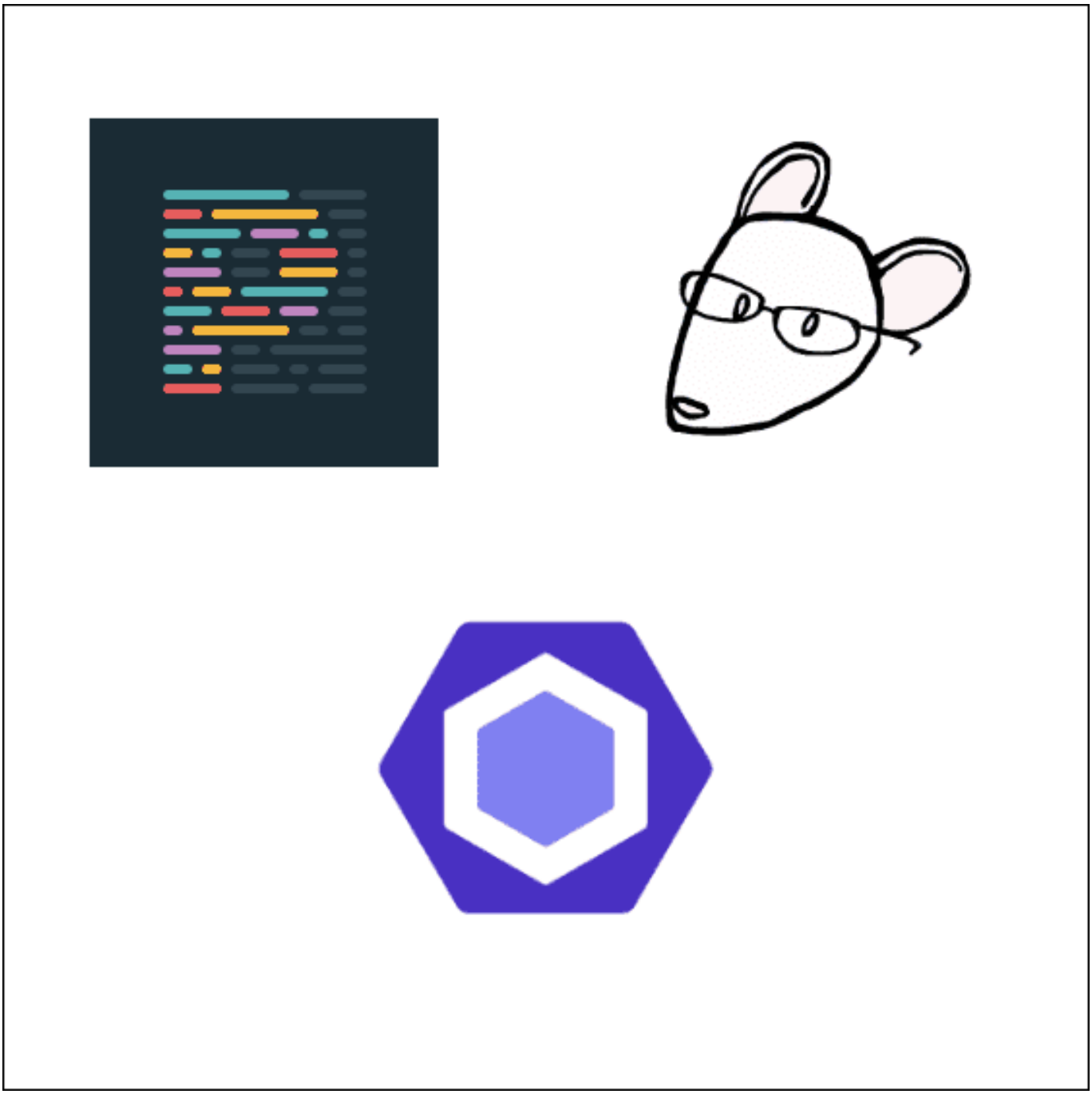
Theodo

Why You Should Use ESLint, Prettier & EditorConfig

August 21, 2019 François Hendriks 7 min read



This post is the first of a series of articles on how to empower your dev environment with **ESLint**, **Prettier**, and **EditorConfig**. If you already know ESLint, Prettier, EditorConfig and how you can benefit from using

Theodo

Context

On one of our TypeScript-React projects, we decided to use ESLint and Prettier to both **lint and format** our code. We struggled for days on formatting conflicts between ESLint and Prettier formatting rules. Our code looked like this:

```
function printUser(
    firstName: string,
    lastName: string,
    number: number,
    street: string,
    code: number,
    city: string,
    country: string
): void {
    console.log(
        `${firstName} ${lastName} lives at ${number}, ${street}, ${code} in ${city}, ${country}`
    );
}

printUser(
    'John',
    'Doe',
    48,
    '998 Primrose Lane',
    53718,
    'Madison',
    'United States of America'
);
```

After a lot of investigating, we managed to make ESLint and Prettier coexist and work together. On top of this, we even decided to add EditorConfig to the mix! I then wrote a few articles on the subject to share my findings. Though solving the formatting issues took us some time, we did not abandon the idea of using ESLint, Prettier and EditorConfig together. The simple reason is that **while they all have a similar purpose, they each excel in their own field**.

ESLint, Prettier and EditorConfig

Our three tools all have **similar objectives**:

- make code more **consistent** in itself and across team members
- **detect problematic code patterns** that could lead to potential bugs

ESLint is by far the most popular JavaScript **linter**. It statically analyzes your code to help you detect formatting issues and find code inconsistencies.

Prettier, while similar to ESLint by **formatting** your code, does not check your code quality. It just serves as a code formatter. It does this job pretty well though by natively supporting JavaScript but also JSX, Flow, TypeScript, HTML, JSON, CSS and many more languages.

EditorConfig, on the other hand, does not lint or format your code. It merely **defines a standard code formatting style guide** among all the IDEs and editors used within a team of developers. For instance, if a team uses two main editors such as Sublime Text and Visual Studio Code, EditorConfig allows them to define a common indentation pattern (spaces or tabs) within a single file.

Customize the rules

Your linting and formatting experience will depend on your customization of ESLint, Prettier and EditorConfig rules.

ESLint's configuration is provided within a **.eslintrc.*** file (can be .eslintrc.json or .eslint.yaml ...) that can be used to define your own set of rules or to use plugins available for different frameworks (React, Vue, TypeScript). We will use this configuration file later in the article.

Unlike ESLint which requires a configuration file, Prettier is supposed to work with **no configuration at all**. The idea is to make teams of

Finally, the **.editorconfig** configuration file overrides virtually any IDE or editor configuration. This allows you to create a common multi-editor development environment for all the team members.

Auto fixing ability

ESLint and Prettier both provide an **auto-fixing feature** that allows them to make changes to your code whenever either one finds a fix to an error. This awesome feature can be integrated within your IDE or editor to correct and format your code whenever you save a file or paste some code. EditorConfig, on the other hand, will directly override your editor's configuration.

Wait a minute! Why use all three?

That is a good question! **ESLint** already does code auto-formatting, so why use **Prettier**? And while we are at it, **Prettier** already formats our code without the editor. So why even bother with **EditorConfig**?

To answer the first question, one should keep in mind that linters ultimately have two categories of rules:

- **Formatting rules**: Rules that prevent inconsistent and ugly looking code (eg: max-len, no-mixed-spaces-and-tabs, keyword-spacing, comma-style...)
- **Code-quality rules**: Rules that prevent useless or error making code (eg no-unused-vars, no-extra-bind, no-implicit-globals, prefer-promise-reject-errors...)

Example

To illustrate this, we create a simple JavaScript `main.js` file. We will then run **ESLint** and **Prettier** using the CLI for demonstration purposes.

```
function printUser(firstName, lastName, number, street,
    console.log(`${firstName} ${lastName} lives at ${num
}
printUser('John', 'Doe', 48, '998 Primrose Lane', 53718
```

Notice that we use a 4 sized tab indentation.

We add **ESLint** and **Prettier** to our project as dev dependencies so that we can compare both tools on the same file.

npm install eslint prettier --save-dev

We also add a simple .eslintrc.json configuration to add ESLint recommended rules. This also allows ES6 expressions (like the backticks) and node variables (for console.log).
To this configuration, we add some custom rules:

- Two spaces indentation
- Lines should not exceed 80 characters

```
{
  "extends": ["eslint:recommended"],
  "env": {
    "es6": true,
```

```
    "max-len": ["error", {"code": 80}],
    "indent": ["error", 2]
  }
}
```

Now we run ESLint in the terminal.

```
npx eslint main.js
```

ESLint does not allow any console statements and detects tab indentation. It wants spaces and also sees two lines exceeding 80 characters.

We now run ESLint with the fix option to see what it can do.

```
npx eslint main.js --fix
```

indentations and a maximum line length of 80 characters. This is the same configuration we have for ESLint.

npx prettier main.js --write

We now look at our main.js file:

```
function printUser(firstName, lastName, number, street,
  console.log(
    `${firstName} ${lastName} lives at ${number}, ${stre
  );
}

printUser(
  'John',
  'Doe',
  48,
  '998 Primrose Lane',
  53718,
  'Madison',
  'United States of America'
);
```

Prettier managed to reformat our code without us specifying any configuration to fix our max-len rule while **ESLint** could not. But Prettier did not warn us about the console.log statement which enters the `code quality` rules. So in order to have the best possible linting experience detecting both `code quality` and `code formatting`, one should definitely use both tools.

format your already written code. This allows EditorConfig to be used in many more languages and projects than Prettier.

Using EditorConfig will also avoid Prettier of uselessly formatting your file on save as your editor will already have done the formatting based on EditorConfig's rules. The tricky part though is to make sure Prettier and EditorConfig have the same rules without repeating your configuration in two separate files.

Conclusion

The combination of all three tools definitely improves your development experience. The question is how do we configure the tools to make them work well together.

But before we dive any deeper, I would like to warn you. I have already seen a project in which the team abandoned ESLint in favor of Prettier. If you do not want to pick all three tools for your project and need to choose only one, the choice is yours. Keep in mind though that **Prettier does not provide any linting capability**. It just formats your code without checking its syntax. It is therefore **highly recommended to use ESLint** before even thinking about **Prettier**.

This being said, if you want to use both, **formatting conflicts** can appear. ESLint could decide to format your code differently from Prettier. Moreover, using EditorConfig and Prettier together will make you write a redundant configuration. So how should you configure all these tools to make them play well together? The answer is in the **next article on the subject on how to Set up ESlint, Prettier & EditorConfig without conflicts**.

François Hendriks

Developer @Theodo. Fan of VueJS and TypeScript!

Generate Images With Dynamic Content On A Symfony Project With High Performance And Code Maintainability →

Gonzalo Arrivi • a year ago

I don't really see the appeal of having an ".editorConfig" file that you have to check is in sync with the configuration of "Prettier". The main problem you use as thesis for this article should be easily solved by the solutions given here https://prettier.io/docs/en... .
Also, if I'm not mistaken, currently every major editor would have some kind of support to use prettier formatting, so...