Using lint-staged, husky, and pre-commit hooks to fail fast and early

Dominic Fraser Follow
Jul 1, 2019 · 7 min read ★

lint-staged + husky npm packages

This post will look at setting up `lint-staged` and `husky` for running `pre-commit` checks. A lot of context is given in this post, but the actual code changes are very small!

What are pre-commit checks?

Pre-commit checks run after staging your changes and running `git commit` and before a commit is completed. If the checks fail then the commit is not made and an error shown, while if all checks pass the commit is made as normal.

Why run pre-commit checks?

Pre-commit checks are commonly used to run linting scripts and tests, allowing each commit to be as clean as possible. As the lint-staged docs state, they prevent '💩 slipping into your code base!'.

I've worked in repositories both with and without pre-commit checks, and it was not until recently that I appreciated the full value they have. They had often seemed annoying, interrupting a 'wip' commit by bailing out on a style rule and preventing a

commit being made without either fixing the error or using the `--no-verify` commit flag.

A combination of two things have changed my opinion on this.

Firstly, working in a repository that has approaching a 20 minute Continuous Integration pipeline. This pipeline is triggered on every push and pull request, and contains install, build, lint, and several test steps. Having a simple lint failure (unused import for example) after 15 minutes is surprisingly different in its mental effect than after 5 minutes.

This was given extra clarity after I was recommended Paul Armstrong's React Europe 2019 'Move fast with confidence' talk. I can highly recommend the entire talk, but the linked timestamp covers points he makes around the negative effect of context switching. He makes the point that in the time taken for CI to run and fail a developer will often move on to thinking of other work, and suffer the cost of switching context away from it if CI fails. This also effects the PR reviewer if they have begun a review, only to then have CI fail while they are reviewing.

Effort should therefore be put in to fail on simple checks as quickly as possible, and pre-commit hooks can be used to detect failures before they are even committed. The extra few seconds per commit far outweigh the lost time of CI failing after 15 minutes, determining and fixing why, and then keeping half an eye on the next 20 minute run to make sure it does not fail again.

The point is also made that we should trust our tools. If it is possible for them to fix linting errors for us then we should lean into this (after all the PR still will go through human review) and save as much time as possible.

This post will look at the setup I have since begun to add to all repositories my team owns.

Setup

We will look at two different repository structures. One that has a single package, with a `package.json` at the top level, and another that has multiple packages, with a root `package.json` and then one in each package under `/packages/package-name/package.json`. In the multi-package repo we have a single linting config set up in the root package, with its rules applied across all sub packages.

The assumption is made that any linters and tests are already set up. Here we will refer to `eslint` (with the prettier plugin recommended to be added) and `stylelint`, but any

commands can be replaced with linters of your preference. We will use `npm`, but again `yarn` can be used if preferred.

For both repository structures we will want to install `lint-staged` and `husky` in our root package.

```
npm install lint-staged husky --save-dev
```

At the time of writing the versions are `lint-staged@8.2.1` and `husky@2.4.1`.

Single package

An example repo structure could be:

```
|-- package.json
|-- README.md
|-- prettierrc.yml
|-- eslintrc
|-- stylelintrc.json
|-- .gitignore
|-- src
   |-- scripts.js
   |-- styles.scss
```

The assumption is made that linting scripts such as the below are already in place in your `package.json`:

```
"scripts": {
    "lint:scss": "stylelint 'src/**/*.scss' --syntax scss",
    "lint:scss:fix": "stylelint 'src/**/*.scss' --syntax scss --fix",
    "lint:js": "eslint . --ext .js,.jsx",
    "lint:js:fix": "npm run lint:js -- --fix",
}
```

Here we have a two scripts that can lint files, and a paired script that will lint them and make any changes it can to automatically fix and errors found. This is not always possible, a missing import is too complex for the linter to know how to fix for example, whereas a missing semi-colon can simply be added.

The configuration required here is straightforward. This is added to the root `package.json`.

```
"lint-staged": {
  "src/**/*.{js,jsx}": [
    "eslint . --fix", "git add"
  ],
  "src/**/*.scss": [
    "stylelint --syntax scss --fix", "git add"
  ],
},
"husky": {
  "hooks": {
    "pre-commit": "lint-staged"
  }
},
```

The `husky` object is used to specify which hook to use, and that `lint-staged` is to be ran on it.

The `lint-staged` object is used to search for staged files that match the `micromatch` pattern in its key. An array of commands is then run against those files.

In this example we only look at files under the `/src` directory, though we could look at all files if we wished.

Notice that in the NPM script the match pattern is passed as a CLI flag, whereas in the `lint-staged` configuration it is as the object key.

Next we run the `:fix` version of each command, meaning that the linter will try to fix any errors it finds. If it does it will then make these changes, but they will not be staged. `git add` then stages the change made, meaning that the completion of `lint-staged` will directly commit any fixes made in the current commit.

If the linter cannot make the fix then `lint-staged` will bail out, preventing the commit being made, and printing the linter's output to the console to allow manual fixes to be made.

Multi-package

If we had a multi-package repository it may look like this:

```
|-- package.json
|-- README.md
|-- prettierrc.yml
|-- eslintrc
|-- stylelintrc.json
|-- .gitignore
|-- packages
   |-- package-one
     |-- src
```

```
        |-- package.json
        |-- scripts.js
        |-- styles.scss
    |-- package-two
      |-- src
        |-- package.json
        |-- scripts.js
        |-- styles.scss
```

A reminder here that this example has `eslint` installed in the root. If you wish `eslint` to be installed in each package then the setup in this example non-hoisted lerna repo can be used.

We will look at only running `eslint` here to keep the code more readable, but the same `stylelint` commands can be used as in the single package example previously.

There are two ways this may be set up, using `lerna`, or more manually. We won't go into how `lerna` works in depth here, just know it is a tool for working in multi-package repositories where the packages may depend on each other. It allows them to be easily linked and published. One feature is that running the script

```
"lerna:lint:js": "lerna run lint:js:fix"
```

from the root package would result in any `lint:js:fix` script in any sub-package being ran.

An alternative to achieving the same is more manual, using the example above what `lerna` does in one set would need two scripts:

```
"lint:one:js": "npm run lint:js:fix --prefix packages/package-one",
"lint:two:js": "npm run lint:js:fix --prefix packages/package-two"
```

For both options each sub-package would need the follow in their individual `package.json`'s for either `lerna` or `--prefix` to target.

```
"scripts": {
    "lint:js:fix": "npm run eslint . --ext .js,.jsx --fix",
}
```

Standard setup

`per package`

```
"scripts": {
    "lint:js:fix": "npm run eslint . --ext .js,.jsx --fix",
}
```

`root`

```
"scripts": {
  "lint:one:js": "npm run lint:js:fix --prefix packages/package-one",
  "lint:two:js": "npm run lint:js:fix --prefix packages/package-two"
}
"lint-staged": {
  "packages/package-one/**/*.{js,jsx}": [
    "npm run --silent lint:one:js",
    "git add",
  ],
  "packages/package-two/**/*.{js,jsx}": [
    "npm run --silent lint:two:js",
    "git add",
  ],
},
"husky": {
  "hooks": {
    "pre-commit": "lint-staged"
  }
},
```

Lerna setup

`per package`

```
"scripts": {
    "lint:js:fix": "npm run eslint . --ext .js,.jsx --fix",
}
```

`root`

```
"scripts": {
  "lerna:lint:js": "lerna run lint:js:fix"
}
"lint-staged": {
  "packages/**/*.{js,jsx}": [
    "npm run --silent lerna:lint:js",
    "git add",
  ]
},
"husky": {
```

```
  "hooks": {
    "pre-commit": "lint-staged"
  }
},
```

Negating files

If there are certain files that you do **not** wish to have linters ran over then a negation pattern can be used to achieve this. This is worth mentioning as the syntax seems to differ slightly from the `micromatch` standard syntax, as noted in this negation issue.

Say we had a `flow-typed` file at the root level we did not want included, as well as some other script files. This pattern could be used:

```
"!(packages|flow-typed)/**/*.{js}": [
  "npm run --silent lint:js -- --fix", "git add"
]
```

Here we would split out a separate step to look at any `js` files **not** in `/packages` or `/flow-typed`.

Running tests

We have only looked at running linters in these examples, but it is also possible to run tests in the same hook, targeting only files that have changed tests using `jest --bail –-findRelatedTests`. It is for you to decide if this is something you want to include.

This can be seen in the screen cap given in the talk linked earlier, Paul Armstrong's 'Move fast with confidence'.

Final thoughts

By adding a `husky` and a `lint-staged` object to your `package.json` you can quickly integrate `pre-commit` checks to your workflow, customise them to fit your individual preferences, and save time for all developers that work in that repository.

The hooks will allow common errors to be found, fixed, and added, without any additional interaction added — all before CI is even run.

Thanks for reading! 😀

Other posts I've written include:

- Testing React applications with jest, jest-axe, and react-testing-library

- Four simple automated accessibility testing tools
- Customising CodeceptJS E2E tests

Lint Staged | Husky | Continuous Integration | Testing | JavaScript