

Static Analysis Testing JavaScript Applications



Transcripts for [Kent C. Dodds](#)

(<https://egghead.io/instructors/kentcdodds>) course on [egghead.io](https://egghead.io/courses/static-analysis-testing-javascript-applications) (<https://egghead.io/courses/static-analysis-testing-javascript-applications>).

Description

There are a ton of ways your application can break. One of the most common sources of bugs is related to typos and incorrect types. Passing a string to a function that expects a number, or falling prey to a common typo in a logical statement are silly mistakes that should never be made, but this happens all the time.

We could write a comprehensive suite of automated tests for our entire codebase to make certain mistakes like this never happen, but that would likely be too much work and slow development down to be worth the benefit. Luckily for us, there are tools we can use to satisfy a whole category of testing with a great developer experience.

Lint JavaScript by configuring and running ESLint

Here I have a simple project with a couple of bugs in this example. I'm using `typeof` improperly. I've got a very subtle bug right here. I'm messing up this string interpolation, not using a template literal string.

example.js

```
const name = 'Freddy'
typeof name === 'string'

if (!'serviceWorker' in navigator) {
  // you have an old browser :-(
}

const greeting = 'hello'
console.log(`${greeting} world!`)

[(1, 2, 3)].forEach(x => console.log(x))
```

I'm going to use eslint to make sure that I avoid these problems. I'm going to go ahead and `npm install --save-dev eslint`.

Terminal Input

```
npm install --save-dev eslint
```

With that installed, if I check out my `package.json`, I'll see that that's in my `devDependencies`. I can configure eslint. There are various ways to do this. I'm going to use `.eslintrc`. This is a JSON file where I can configure eslint.

The first thing I need to configure is the version of JavaScript that I want it to be checking. I'm going to say `parserOptions`, and my `ecmaVersion` is 2018, `"ecmaVersion": "2018"`. I'm writing the latest version of JavaScript. Then I can specify some `rules`.

`.eslintrc`

```
{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "rules": {

  }
}
```

One rule that I'm particularly interested in right now is the `typeof`, to make sure that I don't have a typo when I'm checking the `typeof` something. Let's go ahead and add `"valid-typeof": "error"`. That's one of the built-in rules in eslint.

```
{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "rules": {
    "valid-typeof": "error"
  }
}
```

Now with that set, I can run `npx eslint` on my source directory, `src`, and I'll get an error for that particular rule. This error is actually going to fail my build if I were to put this eslint script into my build.

Terminal Input

```
npx eslint src
```

If I didn't want this particular rule to fail my build, then instead of error, I could say `warn`.

`.eslintrc`

```
{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "rules": {
    "valid-typeof": "warn"
  }
}
```

Then if I run eslint again, I'm going to get a warning for that same rule failure. This will not fail my build, but it will let me know that there is a problem there. I can also disable this rule entirely by adding `off`. Now if I run eslint, it's not going to give me any output.

```
{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "rules": {
    "valid-typeof": "off"
  }
}
```

Another option that I have here is I can extend, add an `extends` property here. There are a lot of different configurations that I can install. There's a built-in configuration called `eslint:recommended`. With that, I can run `npx eslint src` and I get a whole bunch of errors here in the Terminal.

```

{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "extends": [
    "eslint:recommended"
  ],
  "rules": {
    "valid-typeof": "error"
  }
}

```

Then I can go into my `example.js`. I can fix each one of these to avoid the problems these eslint rules were written for. To save us some time, I'm going to go ahead and just paste in the fixed code. With that code fixed, now I can run `npx eslint src`. I'm getting a couple other errors that I'm going to configure eslint to avoid.

The screenshot shows a VS Code editor with a file named `example.js` in the `src` directory. The file contains the following code:

```

const name = 'Freddy'
typeof name === 'string'

if (!('serviceWorker' in navigator)) {
  // you have an old browser :-()
}

const greeting = 'hello'
console.log(`${greeting} world!`)

;(1, 2, 3).forEach(x => console.log(x))

```

The `EXPLORER` sidebar on the left shows the project structure with `example.js` selected. The `TERMINAL` on the right shows the output of running `npx eslint src`:

```

/Users/kdodds/code/static-testing-tools/src/example.js
  4:26  error  'navigator' is not defined  no-undef
  9:1   error  Unexpected console statement no-console
  9:1   error  'console' is not defined   no-undef
 11:27  error  Unexpected console statement no-console
 11:27  error  'console' is not defined   no-undef

✖ 5 problems (5 errors, 0 warnings)

```

Next, we're going to go to our `.eslintrc` file and we're going to set the environment, `"env"`, for our code as `"browser": true`. Now if I run this again, I'm going to just see unexpected console statement. If I want to keep those there, then I can say `"no-console": "off"`. I'm going to remove `valid-typeof` because we're going to get a good configuration from the eslint recommended configuration.

`.eslintrc`

```
{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "extends": [
    "eslint:recommended"
  ],
  "rules": {
    "no-console": "off"
  },
  "env": {
    "browser": true
  }
}
```

With that, I can now run `npx eslint src` and everything is working.

Terminal Input

```
npx eslint src
```

Let's go ahead and add this to our script so we don't have to run `npx` every time. We'll go to `package.json` to our `scripts`. We'll add a `"lint": "eslint src"`.

`package.json`

```
"scripts": {  
  "lint": "eslint src"  
},
```

Now we can run `npm run lint`.

Terminal Input

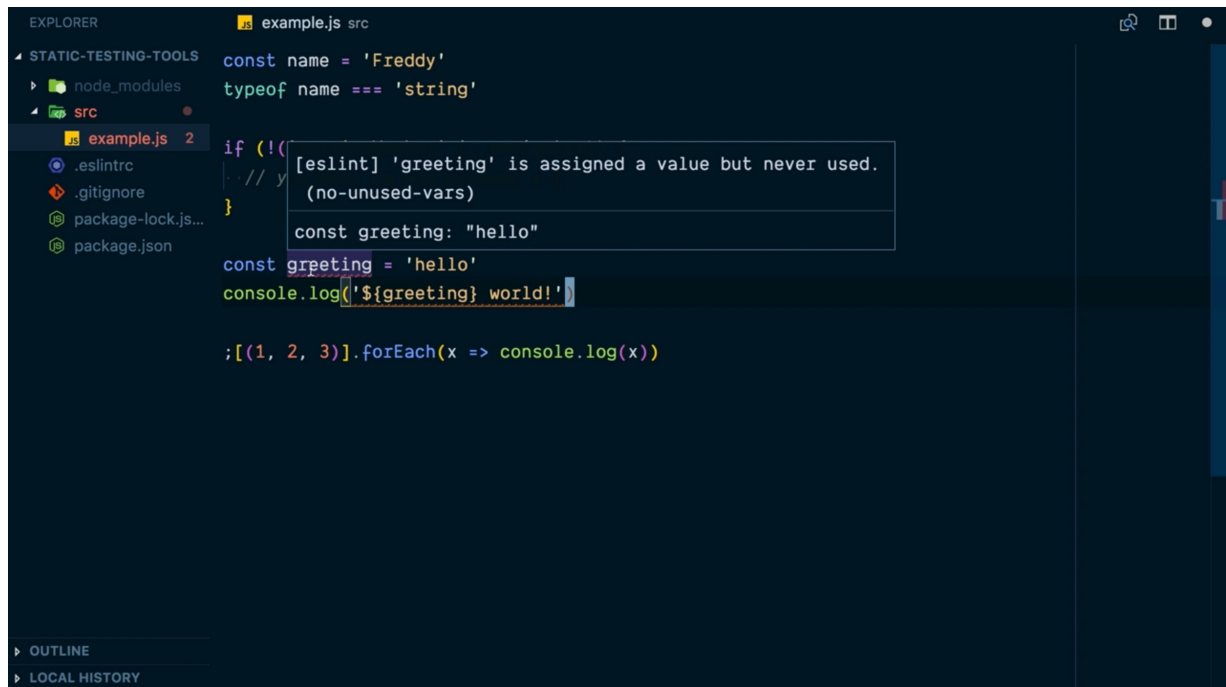
```
npm run lint
```

In review, all we had to do here was install `eslint` as a dev dependency. We also added a `script` in our `package.json` for linting the `src` directory. Then we configured `eslint` in our `.eslintrc`, first specifying the version of JavaScript that we're going to be writing.

We also specified some custom rules, the environment that our JavaScript is going to be running in so it would know what global variables are available, and a configuration that we want to extend.

I'm going to do one more thing, and that is with most modern editors there is an `eslint` plugin that you can use. I'm going to go search for `eslint` in the extensions for VS Code here. I already have it installed. I'll just enable it.

Then I'll reload my editor. Now if I go to that `example.js` file and I make some sort of error, like change this to a regular string, I'm going to see an error here without having to run the `eslint` script in the terminal. This helps me have a much faster feedback loop as I'm editing the code.



Format Code by installing and running Prettier

The first thing I'm going to do here is `npm install` as a dev dependency prettier, `npm install --save-dev prettier`.

Terminal Input

```
npm install --save-dev prettier
```

With that installed, I can now run `npx prettier src/example.js`. It will automatically format this file for me.

Terminal Input

```
npx prettier src/example.js
```

If I look at `example.js` and I do some really weird formatting here -- we'll just put things all over the place -- `prettier` can take that and turn it into something that looks reasonable. If I want to have `prettier` save this value, then I can add a `--write` flag. It will write the changes to disk.

Terminal Input

```
npx prettier --write src/example.js
```

With those changes, I'll go ahead and add another script here for `"format"`. We'll say `"prettier --write"`. I could say `src/example.js`, but I actually want prettier to format all the files in my project.

package.json

```
"scripts": {  
  "lint": "eslint src",  
  "format": "prettier --write src/example.js"  
},
```

Prettier is actually capable of formatting more than just JavaScript, but JSON and CSS and GraphQL even. Let's go ahead and provide prettier with a glob, `"**"` to match any file in the project that it can format.

```
"scripts": {  
  "lint": "eslint src",  
  "format": "prettier --write \"\\\"  
},
```

Say, ****** any file that ends in **.** any of these extensions,

js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|mdx|graphql|vue.

```
"scripts": {  
  "lint": "eslint src",  
  "format": "prettier --write \"**/*.+  
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|md  
x|graphql|vue)\\\"  
},
```

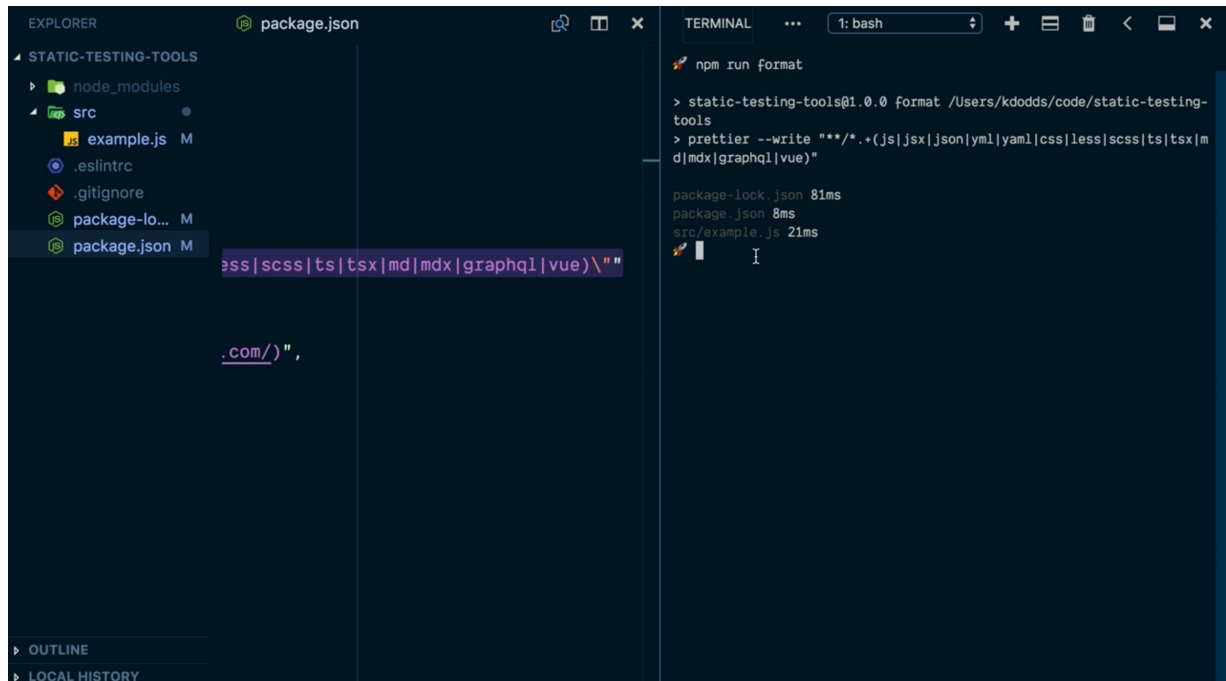
I'm pretty sure that's everything that prettier can support at this moment, but keep a look out, because prettier keeps on adding support for more. Depending on your project, you may or may not care about all of these. So go ahead and just list the ones you care about.

With that now, I'll save **package.json**. Open up my terminal and I'll run **npm run format**.

Terminal Input

```
npm run format
```

Prettier will go through my whole project for all files that match the glob that I provided and attempt to format them.



The screenshot shows the VS Code interface. The Explorer panel on the left shows a project named 'static-testing-tools' with a 'src' directory containing 'example.js'. The package.json file is open in the editor, showing a 'format' script in the 'scripts' field. The terminal window on the right shows the command 'npm run format' being executed, which runs 'static-testing-tools@1.0.0 format /Users/kdodds/code/static-testing-tools'. The output shows that 'package-lock.json' was formatted in 81ms, 'package.json' in 8ms, and 'src/example.js' in 21ms.

```
package.json
{
  "name": "static-testing-tools",
  "version": "1.0.0",
  "description": "A collection of static testing tools",
  "main": "index.js",
  "scripts": {
    "format": "prettier --write \"**/*.+(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|mdx|graphql|vue)\""
  },
  "keywords": [],
  "author": "kdodds",
  "license": "MIT"
}
```

```
terminal
1: bash
npm run format
> static-testing-tools@1.0.0 format /Users/kdodds/code/static-testing-tools
> prettier --write "**/*.+(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|mdx|graphql|vue)"

package-lock.json 81ms
package.json 8ms
src/example.js 21ms
```

Those that are gray needed no changes. If I make a change to the bug in `example.js` again -- and I'll just add a bunch of spaces and whatever else and save that.

Run `npm run format` again -- then we'll see that source example was changed.

Terminal Input

```
run npm format
```

The screenshot shows a VS Code editor with a file named `example.js` in the `src` directory. The file contains the following code:

```
const name = "Freddy";
typeof name === "string";

if (!("serviceWorker" in navigator)) {
  // you have an old browser :- (
}

const greeting = "hello";
console.log(`${greeting} world!`);
[[1, 2, 3]].forEach(x => console.log(x));
```

The `src` directory in the Explorer sidebar contains `example.js`, `.eslintrc`, `.gitignore`, `package-lock.json`, and `package.json`.

The terminal shows the execution of the `format` script:

```
npm run format
> static-testing-tools@1.0.0 format /Users/kdodds/code/static-testing-tools
> prettier --write "**/*.{js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|mdx|graphql|vue}"
package-lock.json 81ms
package.json 8ms
src/example.js 21ms
npm run format
> static-testing-tools@1.0.0 format /Users/kdodds/code/static-testing-tools
> prettier --write "**/*.{js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|mdx|graphql|vue}"
package-lock.json 80ms
package.json 9ms
src/example.js 22ms
```

In review, what we did here is we installed `prettier` as a dev dependency. Then we created a format `script` to use prettier with `--write` so that it would write it to the file. Then we provided a glob that matched all the files that prettier is capable of formatting for us.

In addition to this, many text editors do have support for prettier built in. I'm going to go ahead and find prettier.

Here, I already have it installed. I'll just enable it and reload.

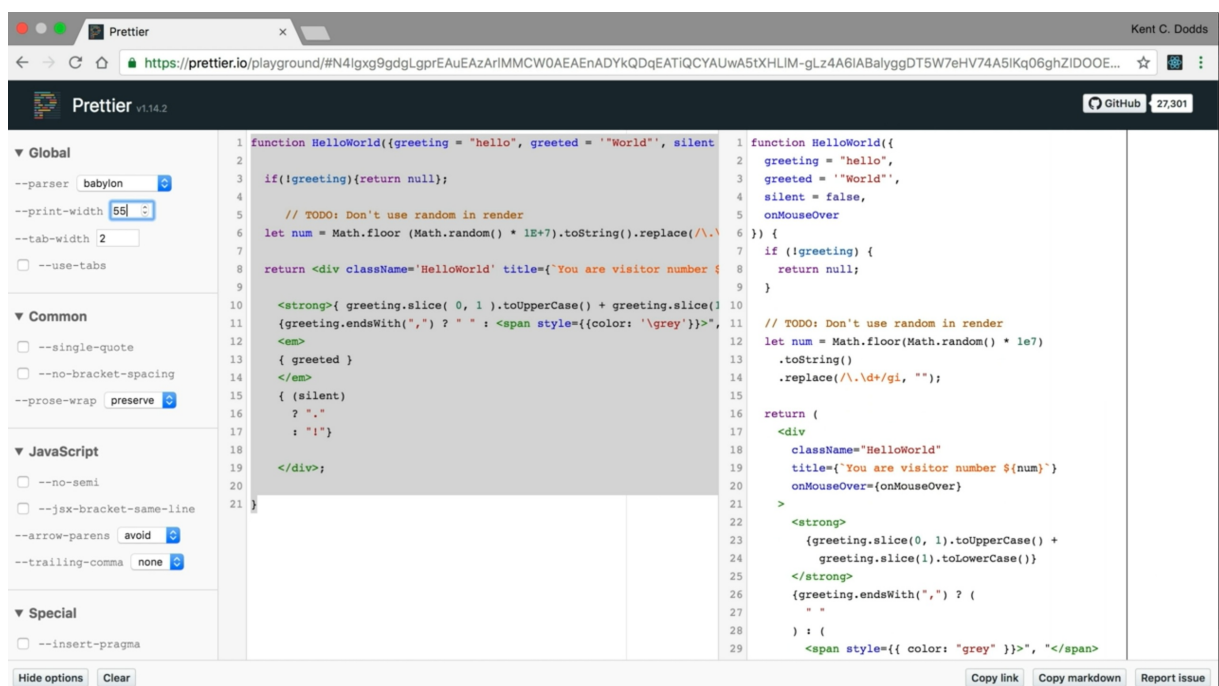
Now if I open up my settings here and go to my `settings.json`, then I can enable `formatOnSave`. Then I can go back to this example file, and I can make all kinds of weird changes here and hit the save key, and it will automatically format for me.

Configure Prettier

When I ran Prettier on my code, it actually changed a couple things. Now, I have double quotes instead of single quotes, and I added semicolons. None of these things are actually changing the

way that the program runs, but it is divergent from my personal style. Prettier allows you to configure some things with the way that it formats your code.

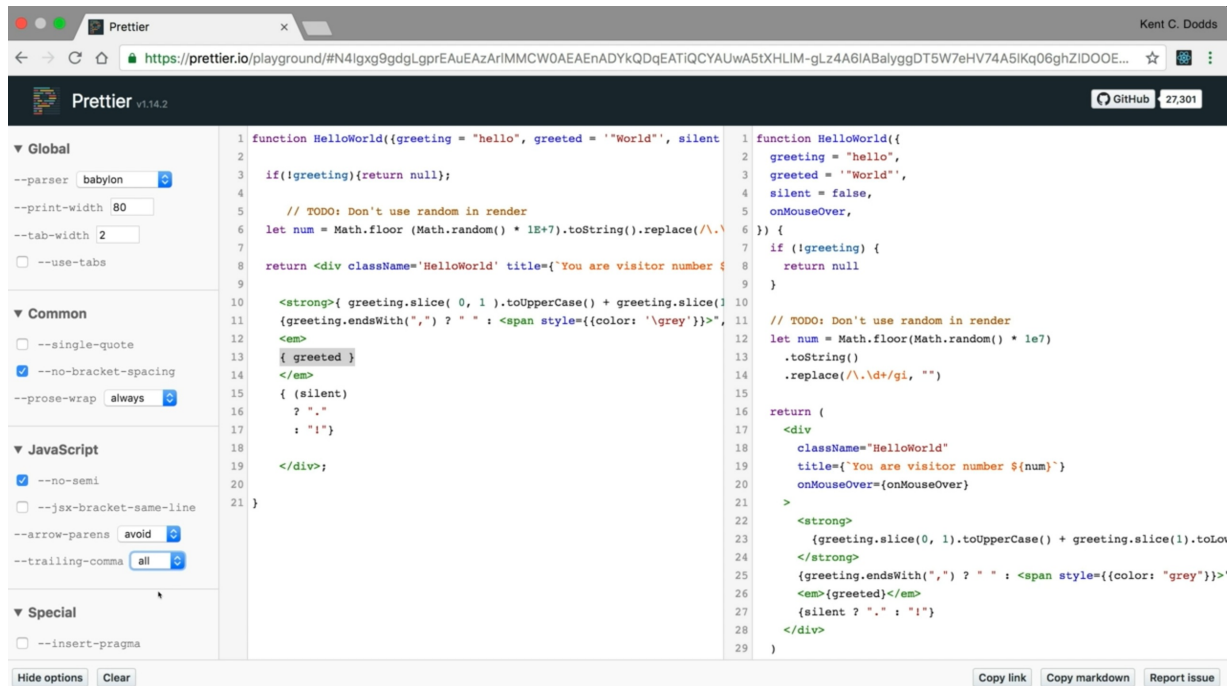
Let's take a look at the [Prettier Playground](https://prettier.io/playground) (<https://prettier.io/playground>) here to see what options are available. You can choose a `--parser`, but that'll be chosen based off of the file name, so we don't need to deal with that. There's also a `--print-width`, so we can make it out to the width that we specify.



I'll go ahead and leave the `--print-width` at 80. There's also a `--tab-width`, so four spaces versus two spaces. Even one space, or three, or four, whatever you want to do here. I like to leave it at two. You can also specify to use tab characters instead of spaces. I'll leave that off.

Then you can choose `--single-quote` instead of double quotes. There's also `--no-bracket-spacing`. That's something I prefer. These brackets around here go away right here. `--prose-wrap` applies to stuff like Markdown. I prefer that, so I'll say *always*.

I prefer no semicolons, so I'll remove those with `--no-semi`. JSX bracket next line, we'll put the closing bracket on the same line with `--jsx-bracket-same-line`. I don't like that, so we'll leave it. `--arrow-parens`, I like to avoid those. We can have them always be printed, but I'll avoid those. I like trailing all commas. That comma can be trailing there with `--trailing-comma`. I prefer that.



Let's go ahead, and I'm going to create a configuration file that has all of these options enabled. Where we put that is right at the root. We'll add a `.prettierrc`, and then I'll just paste this code in here with all of those configuration options that I specified.

`.prettierrc`

```
{
  "arrowParens": "avoid",
  "bracketSpacing": false,
  "jsxBracketSameLine": false,
  "printWidth": 80,
  "proseWrap": "always",
  "semi": false,
  "singleQuote": true,
  "tabWidth": 2,
  "trailingComma": "all",
  "useTabs": false
}
```

Now, if I run `npm run format`, it will run a formatting with the configuration options that I wanted.

Terminal Input

```
npm run format
```

Another bit of configuration that I want to do here is, in normal projects, you're going to have directories where it has generated code, whether that's for your coverage reports or your build files. We don't want to format those.

I'm going to add another file called `.prettierignore` with a dot in front. I want to ignore `node_modules`, which is the default, but I'll put this in here as well. I'll add `coverage`, `dist`, `build`, `.build`, whatever makes sense for your situation. We'll just say `etc`. With that, now, I won't worry about Prettier attempting to format my generated files in my project.

`.prettierignore`

```
node_modules
coverage
dist
build
.build

# etc...
```

In review, all that we needed to do to configure Prettier is we added a `.prettierrc` file in our project, with all the configuration options that we wanted. Then we added also a `.prettierignore` file, so that we can make sure that Prettier doesn't attempt to format files that are generated.

With that configuration, we ran our `npm run format` script again, which runs `prettier --write` across all the files that Prettier supports. That reformatted our example to look the way that I wanted it to.

Disable Unnecessary ESLint Stylistic Rules with `eslint-config-prettier`

As you go about configuring ESLint and extending other configurations, you might end up extending a configuration that conflicts with Prettier. In that case, you'll want to specifically configure a certain rule.

`"semi": ["error", "never"]` No semicolons in this project, for example. Even in this case, this can be pretty annoying.

`.eslintrc`

```
"rules": {
  "semi": ["error", "never"],
  "no-console": "off"
},
```

If I go to my project now and add a semicolon after 'hello', I'm going to see a little red underline saying there's an extra semicolon. I'll want to remove that.

The red goes away, but that's annoying. We have our Prettier integration with my editor. If I hit **cmd+s** to save, then that semicolon will go away automatically for me, thanks to Prettier.

That in itself is annoying. I don't want to even see the red underline. What I really want is to have this rule just be completely disabled. I'll say, **off** Now I have that semicolon. It's not a big deal for me because I can hit **cmd+s** and it'll go away anyway.

```
"rules": {
  "semi": "off",
  "no-console": "off"
},
```

This applies to any of the rules that you have. If you want to make sure that you always have the parentheses around or you want to make sure those parentheses are gone for these arrow functions, you don't want to see a red underline for something that Prettier is going to remove for you anyway.

There's a configuration that we can extend that will automatically disable all the rules that Prettier renders irrelevant. We're going to **npm install** as a dev dependency, **--save-dev eslint-**

config-prettier.

Terminal Input

```
npm install --save-dev eslint-config-prettier
```

While that's installing, I'll go to my `.eslintrc` here. At the very end, I'll add `eslint-config-prettier`. The configurations that come at the end will win in a conflict for rules for all the configurations that come before it.

`.eslintrc`

```
{
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "extends": [
    "eslint:recommended", "eslint-config-
prettier"
  ],
  "rules": {
    "semi": "off",
    "no-console": "off"
  },
  "env": {
    "browser": true
  }
}
```

Then the rules that I specify myself here will win in a conflict with any of the rules specified by these configurations. I can get rid of this `"semi": "off"`, because `eslint-config-prettier` is going to disable that for me.

```
"rules": {  
  "no-console": "off"  
},
```

I'll save this configuration. I look in `example.js`. I can add this semi-colon after 'hello'. I can add parentheses around my arrow functions. No configurations are going to get mad at me for things that Prettier is going to fix for me anyway.

`example.js`

```
const greeting = 'hello';  
console.log(`${greeting} world!`)  
;[(1, 2, 3)].forEach((x) => console.log(x))
```

In review, all that we needed to do for this was add as a dev dependency `eslint-config-prettier` and then extend `eslint-config-prettier` in our ESLint configuration.

Validate all files are formatted when linting

In a project with multiple people on the team, you're going to have some people who have the editor integration going on with their project and everything looks great, and you'll have some people who don't. Let's see what we can do about making sure that people run the format script.

I'm going to open my default user settings. I'll disable this format on save. Now I can make some changes. Hit the save key. It doesn't format automatically.

example.js

```
const name = 'Freddy'
typeof name === 'string'

if (!('serviceWorker' in navigator)) {
    // you have an old browser :-(
}

const greeting = 'hello'
console.log(`${greeting} world!`)
;[(1, 2, 3)].forEach(x => console.log(x))
```

This is what it would look like, potentially, if somebody who doesn't use Prettier in their editor went ahead and tried to save some code. Because ESLint isn't checking this kind of formatting for us, we want to have some sort of automated validation to make sure that things have been formatted properly.

One thing that we can do is if I run Prettier with `npx` and run it on source example, `npx prettier src/example.js`, then I can add the flag `--list-different`. It's going to list for me all the files that would be changed by Prettier.

Terminal Input

```
npx prettier src/example.js --list-different
```

I'll go ahead and `--write` this. Then I'll run Prettier with `npx prettier src/example` again. I don't see any files listed.

Terminal Input

```
npx prettier src/example.js --write
npx prettier src/example.js --list-different
```

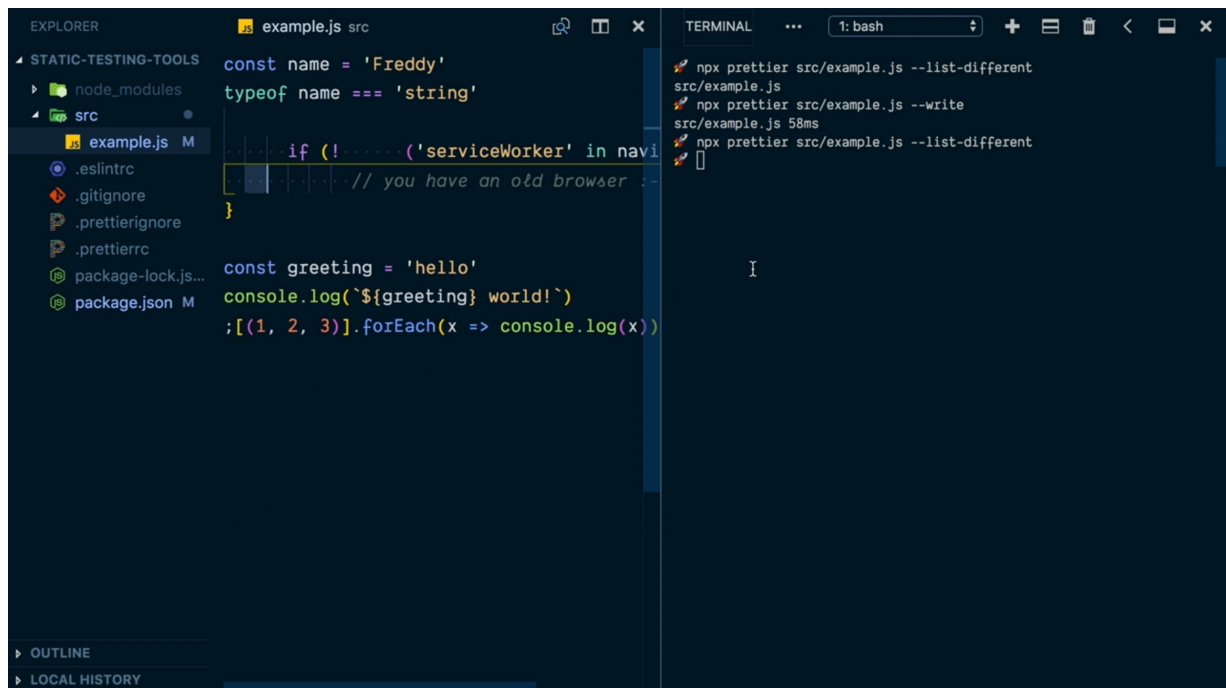
This script will exit with a non-zero exit code, meaning the script will fail if any of these files are listed. We can make it part of a validation script. I'm going to make a new script in `package.json`, called `validate`. We'll `npm run lint`. Then we'll run our Prettier script here. We'll copy this. Instead of `--write`, we'll run `--list-different`.

package.json

```
"scripts": {
  "lint": "eslint src",
  "format": "prettier --write \"**/*.+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)\",
  "validate": "npm run lint && prettier --
list-different \"**/*.+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)\",
},
```

With that now, let's go ahead. I'll make a big mess of this. We'll throw a couple things all over the place.

example.js

A screenshot of the Visual Studio Code editor. The Explorer sidebar on the left shows a project structure with 'src' containing 'example.js'. The main editor window displays the content of 'example.js', which includes a TypeScript-style type check and a JavaScript snippet. The terminal on the right shows the execution of 'npx prettier' commands, which successfully format the file and report the time taken (58ms).

```
const name = 'Freddy'
typeof name === 'string'

if (!('serviceWorker' in navigator)) {
  // you have an old browser
}

const greeting = 'hello'
console.log(`${greeting} world!`)
;(1, 2, 3).forEach(x => console.log(x))
```

```
npx prettier src/example.js --list-different
src/example.js
npx prettier src/example.js --write
src/example.js 58ms
npx prettier src/example.js --list-different
```

Now if I run `npm run validate`, I'll see my ESLint runs.

Terminal Input

```
npm run validate
```

Everything passes ESLint, but Prettier is saying, "Hey, there's a file here that should have been formatted but wasn't." Then I can run `npm run format`. Then I'll run the `validate` script again. Everything passes fine.

Terminal Input

```
npm run format
npm run validate
```

This is a little bit uncomfortable for me. We've got a huge string of text in `package.json` that we're repeating in both of these scripts. If we wanted to add a new file, we'd need to make sure to add it to both of them.

What I'm going to do is just a quick little clean-up. We'll add a `prettier` script. The `prettier` script will be responsible for this part. The flags will be handled by the individual scripts.

`package.json`

```
"scripts": {
  "lint": "eslint src",
  "format": "prettier --write \"**/*.+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)\"",
  "prettier": "prettier \"**/*.+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)\"",
  "validate": "npm run lint && prettier --
list-different \"**/*.+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)\""
},
```

Here, for our format, we'll run `npm run prettier`. Then we'll forward along the write flag, `-- --write`. For our `validate` script, we're going to run `npm run prettier`. We'll forward along the `--list-different` flag. We'll get rid of that. Now things look a little bit cleaner.


```
"scripts": {  
  "lint": "eslint src",  
  "format": "npm run prettier -- --write",  
  "prettier": "prettier \"**/*.+  
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr  
aphql|mdx)\"",  
  "validate": "npm run lint && npm run  
prettier -- --list-different"  
},
```

In review, to validate that all the files in the project have been formatted by Prettier, we can use a `--list-different` flag when we run Prettier. If there are any files that would be different if Prettier were to format them, then it will fail the script. Otherwise, the script will pass.

Avoid Common Errors with Flow Type Definitions

To install Flow, we're going to `npm install` as a dev dependency, `--save-dev flow-bin`. `flow-bin` will be added to our dev dependencies here, and it includes a binary in the `.bin` directory called Flow. We can use that in our scripts in `package.json`.

Terminal Input

```
npm install --save-dev flow-bin
```

I'll add a `flow` script, and that will simply be `flow`.

`package.json`

```
"scripts": {
  "lint": "eslint src",
  "flow": "flow",
  "format": "npm run prettier -- --write",
  "prettier": "prettier \"**/*.+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)\"",
  "validate": "npm run lint && npm run
prettier -- --list-different"
},
```

Now, if we run `npm run flow`, it's going to tell us that we need to run Flow with `init` to configure Flow. Let's go ahead and do that.

Terminal Input

```
npm run flow
```

We'll `npm run flow init`, and that's going to create a new file for us called `.flowconfig`, where we configure Flow to ignore certain files, include specific files, and we have several other options available to us.

Terminal Input

```
npm run flow init
```

We're going to go ahead and leave those out. Now, we're going to add a new file that I'm going to call `flow-example.js`. I'm going to paste a bunch of code into here to demonstrate how to use Flow. Here, I have an add function that takes `a` and `b`.

`flow-example.js`

```
function add(a: number, b: number): number {
  return a + b
}
type User = {
  name: {
    first: string,
    middle: string,
    last: string,
  },
}
function getFullName(user: User): string {
  const {
    name: {first, middle, last},
  } = user
  return [first, middle,
last].filter(Boolean).join('')
}
add(1,2)

getFullName({name: {first: 'Joe', middle: 'Bud',
last: 'Matthews'}})
```

Each of those is a `number`, and it returns a `number`. We can define those types right in our JavaScript, making it a lot easier for us to avoid common mistakes. We can also define custom types. This is a `User` object that has a `name` property. That `name` property has properties of its own.

This `getFullName` function is expecting a `User` object that is a `user` of that type, and it returns a `string`. Then we can use those functions, where we can add one and two, and get the full name of this `User` object.

Now, if I open up my terminal here, and I run `npm run flow`, it's going to start up flow for me, and type check all of the files that I have in my project that have a Flow pragma at the top. We're all set. We don't have any errors.

Terminal Input

```
npm run flow
```

Let's see what would happen if I did have an error. I'll pass a string in here.

flow-example.js

```
add('1', 2)
```

Run flow again, and immediately, it's going to tell me it cannot call the string `1`, because it's a string, and that's not compatible with a number.

It says here's where that string is, and this is the function that it's being called into. That's the number that this one is supposed to be. You can either fix the type definition by changing this to accept a string, or you can fix the function call so that this is a number.

There's a whole category of errors that I don't need to worry about running against, because the types are going to be checked statically when we run our `validate` script. Speaking of the `validate` script, let's go to our `package.json`.

We've got our Flow right here. Let's add flow to our validate script. We'll say `&& npm run flow`.

`package.json`

```
"validate": "npm run lint && npm run prettier --  
--list-different && npm run flow"
```

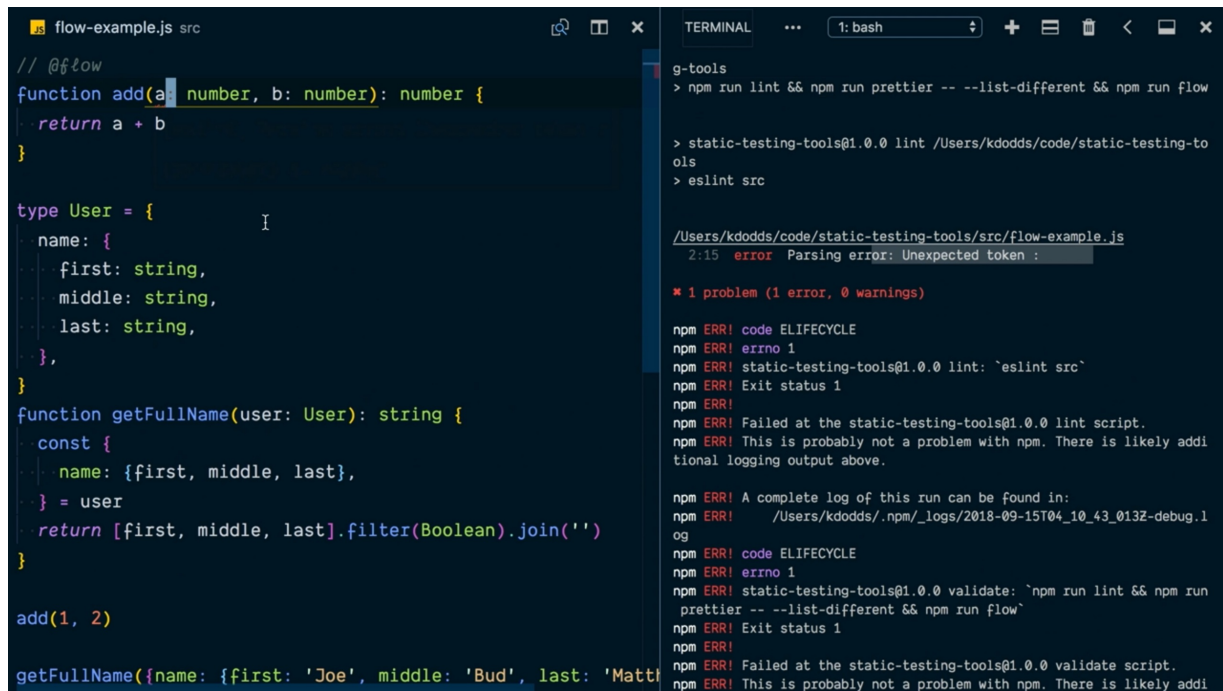
Now, if we run `npm run validate`, we're actually going to have a problem. We're getting a parsing error from ESLint.

Terminal Input

```
npm run validate
```

The parsing error is it's saying there's an unexpected token. That token is in `flow-example.js`, that colon right there.

`flow-example.js`



The screenshot shows a code editor with a file named `flow-example.js` containing Flow-typed JavaScript code. The code includes a function `add` and a `User` type definition. The terminal on the right shows the command `npm run lint && npm run prettier -- --list-different && npm run flow` being executed. The output shows an ESLint error: `2:15 error Parsing error: Unexpected token :` at the line `function add(a: number, b: number): number {`. The error message indicates that ESLint cannot parse the Flow syntax.

ESLint actually doesn't understand Flow syntax. To help it out, we're going to install a parser. I'm going to `npm install --save-dev babel-eslint`.

Terminal Input

```
npm install --save-dev babel-eslint
```

With `babel-eslint` installed, I can go to my ESLint configuration and add `parser: babel-eslint`.

`.eslintrc`

```
{
  "parser": "babel-eslint",
  "parserOptions": {
    "ecmaVersion": "2018"
  },
  "extends": [
    "eslint:recommended"
  ],
  "rules": {
    "no-console": "off",
    "semi": ["error", "never"],
  },
  "env": {
    "browser": true
  }
}
```

Now, if I run `npm run validate`, ESLint is going to pass, Prettier will pass, and Flow will pass. We're all set.

Terminal Input

```
npm run validate
```

In review, to install Flow, you `npm install` as a dev dependency `flow-bin`, and get that added to your dev dependencies. That will add the Flow binary, so you can use it in your scripts here. Then we also added that to our `validate` script.

Then you can start using the Flow comment at the top of your files, which will enable Flow for this particular file. Then you can start adding type definitions to your JavaScript to help avoid some

really common errors in programming with JavaScript.

Validate Code in a pre-commit git Hook with husky

It's great that we have this `validate` script that people can run before they commit their code to make sure that they don't commit anything that is breaking linting rules, or they forgot to format with Prettier, or is breaking flow.

It would be great if we could automatically run this `validate` script when people commit code so they don't forget to do so. We're going to install a tool called Husky that will install git commit scripts.

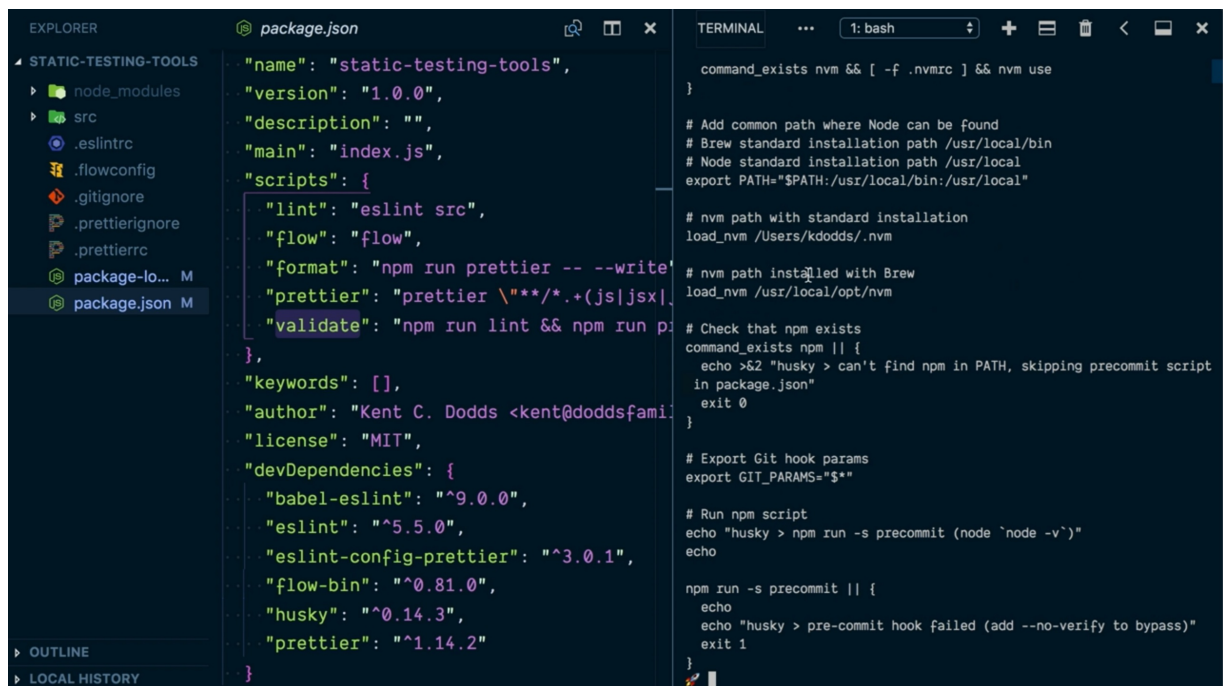
I'm going to `npm install --save-dev husky`. We're going to want to watch the output here to see what Husky does. Here, it says, we're starting Husky here. We're going to set up git hooks. Then we're finished.

Terminal Input

```
npm install --save-dev husky
```

What does that mean? Let's go ahead and take a look at our `git` directory. There's a `hooks` directory inside of there. There are a bunch of git hooks in here. We have some samples that we can look at. For example, the pre-commit file. We'll say, `pre-commit`

```
cat .git/hooks/  
cat .git/hooks/pre-commit
```

A screenshot of a code editor interface. On the left, the 'EXPLORER' sidebar shows a file tree with 'package.json' selected. The main editor area displays the contents of 'package.json', which includes fields like 'name', 'version', 'description', 'main', 'scripts', 'keywords', 'author', 'license', 'devDependencies', and 'husky'. The 'scripts' section contains 'lint', 'format', 'prettier', and 'validate'. The 'husky' field is set to '^0.14.3'. On the right, a 'TERMINAL' window shows a bash prompt and a series of commands and outputs related to setting up Node.js and npm, including paths and environment variables.

```
"name": "static-testing-tools",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "lint": "eslint src",
  "flow": "flow",
  "format": "npm run prettier -- --write",
  "prettier": "prettier \"**/*.+(js|jsx|json|yml|yaml|css|less|scss|ts|tsx|md|graphql|mdx)\"",
  "validate": "npm run lint && npm run prettier -- --list-different && npm run flow"
},
"keywords": [],
"author": "Kent C. Dodds <kent@doddsfamily.dev>",
"license": "MIT",
"devDependencies": {
  "babel-eslint": "^9.0.0",
  "eslint": "^5.5.0",
  "eslint-config-prettier": "^3.0.1",
  "flow-bin": "^0.81.0",
  "husky": "^0.14.3",
  "prettier": "^1.14.2"
},
"husky": {
  "hooks": {
    "pre-commit": "npm run validate"
  }
}
```

Here, this is actually a Husky-generated file. Husky puts this file in here when it's installed. It will call our `pre-commit` script that we configure in our `package.json`. Let's see that in action.

I'm going to add a script here called `precommit`. What do we want to do before people commit? We want to run this `validate` script, so `precommit` will be `npm run validate`.

`package.json`

```
"scripts": {
  "lint": "eslint src",
  "format": "npm run prettier -- --write",
  "prettier": "prettier \"**/*.+(js|jsx|json|yml|yaml|css|less|scss|ts|tsx|md|graphql|mdx)\"",
  "validate": "npm run lint && npm run prettier -- --list-different && npm run flow",
  "precommit": "npm run validate"
},
```

With that, we can `git commit --am 'stuff'`. It runs Husky's `precommit` script. That ran our linting, our Prettier, and flow. If we were to make a change that had a problem...for example, we have some ESLint issue in `example.js`.

Terminal Input

```
git commit -am 'stuff'
```

We try to commit bad stuff, `git commit -am 'bad stuff'`. Husky's going to run. Because ESLint had an error, this commit hook will not allow the commit to go through. If I look at the `git status`, I'll see that I still have that file modified.

Terminal Input

```
git commit -am 'bad stuff'  
git status
```

I can get around this if I add that `--no-verify` flag. I could `git commit -am 'bad stuff' --no-verify`. Then it doesn't even run Husky at all. That's a convenient workaround if you need it.

Terminal Input

```
git commit -am 'bad stuff' --no-verify
```

To do this, we installed Husky. Then we added a `precommit` script to run `npm run validate`. Any time we tried to commit, Husky will run `npm run validate` to validate that we are not breaking linting, that we didn't forget to run Prettier, and that all of our files passed flow.

Auto-format all files and validate relevant files in a precommit script with lint-staged

It's great that we're running this `precommit` script to validate everything before people commit their code automatically, but I think it would be pretty cool if we could also automatically format all the code that's being changed for people, so they don't have to remember to run Prettier.

As our project grows, running linting across the entire project is unnecessary. We should really only be running across the files that changed. This is where `lint-staged` comes in handy. I'm going to `npm install` as a dev dependency, `--save-dev lint-staged`.

Terminal Input

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

With that installed here, and in my dev dependencies, on my `precommit` script, I can now add `lint-staged`. Then we'll run `npm run flow`. We still need to run flow across all the files, regardless of which files changed, but we can at least take care of the linting and Prettier on a per-file basis.

package.json

```
"precommit": "lint-staged && npm run flow"
```

Now, let's go ahead and configure `lint-staged`. I'm going to add a `.lintstagedrc`. Here, this is a JSON file, where I'll specify `linters`. For all files that match `.js`, we want to run ESLint. The way that this is going to work is, when we commit code, `lint-staged` is going to look at the list of the files that we're about to commit, and it watch those files against this glob.

`.lintstagedrc`

```
{
  "linters": {
    "*.js": [
      "eslint"
    ]
  }
}
```

Any file that ends in `.js`, it's going to pass to this command. That could be `one.js`, `two.js`, and `three.js`. This is what `lint-staged` will actually run.

```
{
  "linters": {
    "*.js": [
      "eslint one.js two.js three.js"
    ]
  }
}
```

For us in our configuration, we simply provide the command that we want to have run with all of those files.

```
{
  "linters": {
    "*.js": [
      "eslint"
    ]
  }
}
```

Then let's add one for Prettier. I'm going to copy this, because I don't want to type all this out again. We'll just copy the glob that we're using for Prettier, and we'll put it in here. The commands we want to have run for files that match this glob are first `prettier --write` to format that file.

```

{
  "linters": {
    "*.js": [
      "eslint"
    ],
    "**/*.*+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)": [
      "prettier --write"
    ]
  }
}

```

Then when we're in the process of committing a file, if we make a change to it in that process, we need to re-add that file so that it will be staged for the commit. We're going to say `git add`. The command that `lint-staged` is actually going to run will look something like `one.js`, `package.json`, and `foo.css`.

```

{
  "linters": {
    "*.js": [
      "eslint"
    ],
    "**/*.*+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)": [
      "prettier --write one.js package.json
foo.css",
      "git add"
    ]
  }
}

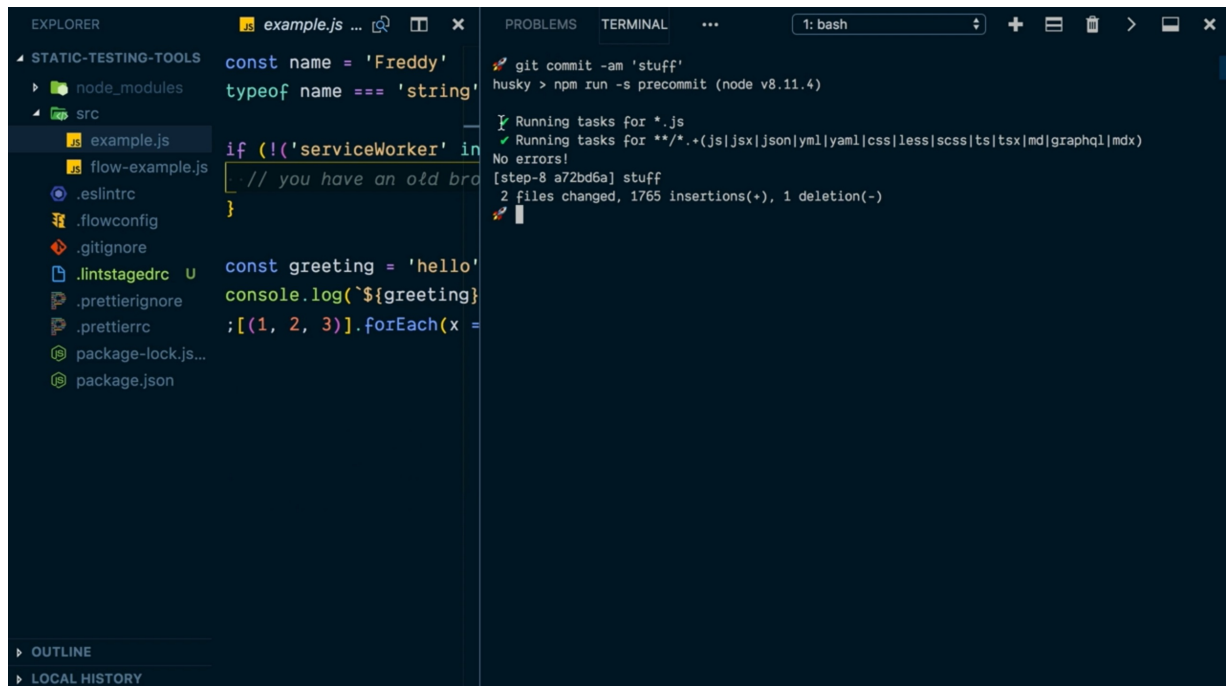
```

Once that's finished, it's going to run `git add` on all of those files as well.

```
{
  "linters": {
    "*.js": [
      "eslint"
    ],
    "**/*.*+
(js|jsx|json|yaml|yml|css|less|scss|ts|tsx|md|gr
aphql|mdx)": [
      "prettier --write",
      "git add one.js package.json foo.css"
    ]
  }
}
```

Let's go ahead and see this in action. I'm going to open up this `example.js`, and we'll mess up its formatting in terrible ways. I'll save that, and then I'll open the Terminal up again, and we'll just expand that, so we can see everything.

`example.js`



```
const name = 'Freddy'
typeof name === 'string'

if (!('serviceWorker' in window)) {
  // you have an old browser
  console.log('hello')
}

const greeting = 'hello'
console.log(`${greeting}`)

;[(1, 2, 3)].forEach(x => {
  console.log(x)
})
```

```
git commit -am 'stuff'
husky > npm run -s precommit (node v8.11.4)
Running tasks for *.js
Running tasks for **/*.+(js|jsx|json|yaml|css|less|scss|ts|tsx|md|graphql|mdx)
No errors!
[step-8 a72bd6a] stuff
2 files changed, 1765 insertions(+), 1 deletion(-)
```

We'll run `git commit -am 'stuff'`. You can see that it ran all the tests for JS. Then it ran the tests for all of those files, which incidentally formatted this file for us, and re-added it. Then it ran flow, and there were no errors.

Terminal Input

```
git commit -am 'stuff'
```

In review, to make this work, we added our `lint-staged` to our dev dependencies in our `package.json`. Then we update our `precommit` to run `lint-staged` and `npm run flow`, because Flow needs to be run across all files, regardless of which ones were changed.

Then we added this configuration in `.lintstagedrc` to run ESLint with all JavaScript files, and get Prettier and `git add` with all files that Prettier can format.