Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Improving Development by Linting Code

Text

In this lesson, we'll add a linter to our project. This will be the last tool we'll use to improve our developer experience, though there are many other tools out there! Linters check our code for errors, but even better, linters tell us when we're writing code that's not very good!

We'll use ESLint (https://eslint.org/) to **lint** JavaScript and find any errors or style/convention issues in our code. By the end of this lesson, we'll have made a few changes to our project:

- Use npm to install eslint and eslint-webpack-plugin to our project.
- Configure webpack to use eslint-webpack-plugin to lint our JavaScript files as it bundles our code.
- Create a new file called `.eslintrc` that configures how ESLint lints our code.

- Add a new script to npm in order to invoke ESLint to lint our code, separate from webpack's bundling process.

Installing ESLint

Let's install `eslint` to install ESLint, a popular JavaScript linter, along with `eslint-webpack-plugin`, which allows us to use the ESLint with webpack. In the terminal, navigate to the root of the Shape Tracker project and enter the following commands.

```
$ npm install eslint@8.18.0 --save-dev
$ npm install eslint-webpack-plugin@2.7.0 --save-dev
```

Let's update our webpack configuration so our code is automatically linted whenever we build:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugi
n');
const ESLintPlugin = require('eslint-webpack-plugin');   //
new line!

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  devtool: 'eval-source-map',
  devServer: {
    contentBase: './dist'
  },
  plugins: [
    new ESLintPlugin(), // new line!
    new CleanWebpackPlugin(),
    ...
  ]
  ...
}
```

For `ESLintPlugin`, we require the package at the top of `webpack.config.js` to make it available to call on within our config file, and then we add a new line to the `plugins` to create a new instance of the plugin with `new ESLintPlugin()`.

By default eslint-webpack-plugin, or `ESLintPlugin`, is configured to look for JavaScript files only and ignore the `node_modules` directory. We don't want to lint `node_modules` because they are external JavaScript libraries that have (hopefully) been tested by other developers. This is common practice.

We can optionally pass an object as an argument into `new ESLintPlugin()`. Within the object we can specify certain options we want configured for ESLint. **We won't do this in our projects**, but here is what this might look like:

```
// Do not add this to your project.
// It is only an example.
new ESLintPlugin({
  option1: value1,
  option2: value2
})
```

To explore this more, check out the list of available options we can use to configure ESLint in the eslint-webpack-plugin project README (https://github.com/webpack-contrib/eslint-webpack-plugin/tree/2.x#options) for version 2.x.

Note that ESLint won't work yet — and we won't be able to build our project until we add an `.eslintrc` configuration file in just a moment. If you try to build now, you'll get an error. That's because our webpack configuration is trying to run ESLint but can't do so without a configuration file.

Configuring ESLint

To use ESLint, we'll need to add a configuration file called `.eslintrc` in the root directory of our project:

.eslintrc

```
{
  "parserOptions": {
    "ecmaVersion": 2018,
    "sourceType": "module"
  },
  "extends": "eslint:recommended",
  "env": {
    "es6": true,
    "browser": true,
    "node": true
  },
  "rules": {
    "semi": 1,
    "indent": ["warn", 2],
    "no-console": "warn"
  }
}
```

Our configuration file is written using JSON (JavaScript Object Notation), but it can also be written in JavaScript or YAML (Yet Another Markup Language). ESLint is very configurable and we just have a few basics here.

- Within the `"parserOptions"` , we specify the parser should look for "modules" (`"sourceType": "module"`) that are written using up through ES2018 (`"ecmaVersion": 2018`). So far we are just using a few of these newer JavaScript features (such as `import` and `export` statements), but we'll use more soon. Note that ES2018 provides even more functionality than ES6 (which was released in 2015). Why use a more advanced version here? Using an older version caused bugs in our configuration of ESLint and this was the hotfix. A hotfix is when a bug causes problems and needs to be fixed immediately. Also, the configuration will enable ESLint to recognize some newer JS features we'll be using in the next section as well.

- With `"extends": "eslint:recommended"`, we use ESLint's recommended set of rules for linting. We could customize these or use other sets as well. Once again, ESLint's recommended rules are very strict — so we can always customize them further as needed. To see exactly which rules are included in the recommended set, go to ESLint Rules (https://eslint.org/docs/rules/). All rules that include a checkmark are part of the recommended set. As you'll see, there are a lot of them!
- Within `"env"`, we let ESLint know a few things about our global environment. Specifically we are using ES6 and we are working in the browser.
- Within `"rules"`, we add a few basic rules:
  - First, we are using semicolons and setting the error level to `1`, which means the linter will give us a warning about missing semicolons. (An error level of `2` means the linter will throw an error instead — stopping our code in its tracks.)
  - We also add a rule for indentation. We pass `"warn"` instead of `1`. The second argument in the array is the number of spaces our code should be indented, so this is a little confusing. In this case, `2` means indented spaces, but it often means the error level. We'll cover the error level more shortly.
  - The last rule is to warn us about any `console.log()` statements in our code. We've asked ESLint to warn us if there are any in our source code. This will help remind us to remove `console.log()` statements before we finalize our code and push it to our remote repo.

Now that we have an `.eslintrc` configuration in place, we can now build our project again — and it will automatically be linted for us! Try introducing an error (for instance, add a typo to a variable name) and see for yourself!

Adding an npm Script to Lint without webpack

It would be nice to have the option to lint our code without bundling it with webpack. We can add a script to our `package.json` file to handle this:

package.json

```
...
  "scripts": {
    "build": "webpack --mode=development",
    "start": "npm run build && webpack-dev-server --open --
mode=development",
    "lint": "eslint src --ext .js"
  },
...
```

Now we can run the script `$ npm run lint` and ESLint will lint all JavaScript files in our `src` folder. It's a simple configuration since we're storing all of our JS in one place:

- `eslint` invokes ESLint
- `src` specifies the folder to lint
- `--ext .js` stands for files with a `.js` extension

Customizing ESLint

ESLint can be a real headache to configure. The default configuration is very strict and it will throw an error at many different things, even when our code is working otherwise. In most cases, these errors are helpful because they'll lead to better code. At other times, though, ESLint can bog us down. In these cases, we can update our ESLint configuration file to be a little less strict.

There are three basic levels for how ESLint can handle an error:

- `0` or `"off"` means that ESLint will ignore a rule entirely. In general, we don't want to do this. It's nice to at least get a warning (even if we plan to ignore it).
- `1` or `"warn"` means that ESLint will provide a warning but will not throw an error. If we don't want a rule to throw an error, we'll generally change it to be a warning instead.
- `2` or `"error"` means that ESLint will throw an error and stop our code in its tracks. That means we can't build until we fix the error (or change the rule to a warning). In general, we should focus on fixing the error *instead* of changing the rule to a warning, but there will be times when it makes sense to update the rule.

Because we are using the recommended set of rules, many are already automatically set to level `2` — the highest level. We can override the recommended settings by adding custom configurations to the `"rules"` object of `.eslintrc`.

Let's go over an example of updating a rule to be a warning instead of an error. The recommended set of rules does not allow `debugger;` statements. This makes sense — in general, we should add breakpoints via the *Sources* tab of Chrome DevTools instead.

However, let's say we don't know about this rule yet (after all, there are so many ESLint rules) and we decide we want to add a `debugger;` statement directly to our code:

src/index.js

```
import Triangle from "./triangle.js";
import './css/styles.css';

function handleFormSubmission() {
  event.preventDefault();
  debugger;
  const length1 = parseInt(document.querySelector('#length
1').value);
  const length2 = parseInt(document.querySelector('#length
2').value);
  const length3 = parseInt(document.querySelector('#length
3').value);
  const triangle = new Triangle(length1, length2, length3);
  const response = triangle.checkType();
  const pTag = document.createElement("p");
  pTag.append(response);
  document.querySelector('#response').append(pTag);
}

window.addEventListener("load", function() {
  document.querySelector("#triangle-checker-form").addEvent
Listener("submit", handleFormSubmission);
});
```

Now if we run `$ npm run build`, we'll get the following error:

```
ERROR in
/Users/staff/Desktop/shape-tracker/src/index.js
  6:3  error  Unexpected 'debugger' statement  no-debugger

✕ 1 problem (1 error, 0 warnings)
```

This error causes our build to fail entirely. As we can see, this is pretty annoying if we want to be able to use `debugger;` statements. There are actually several ways we can address this issue.

Let's say we just want to have a `debugger;` statement briefly — and we don't actually want to update the rule. We can do the following in our code:

```
/* eslint-disable */
debugger;
/* eslint-enable */
```

This disables ESLint before the specified line and then enables ESLint again after the specified line (or lines). We can even make it more specific:

```
/* eslint-disable no-debugger */
debugger;
/* eslint-enable no-debugger */
```

Here, we specify that *just* the `no-debugger` rule should be disabled for the line. We should always be more specific if possible, especially if there's a significant amount of code in the section we are temporarily disabling.

In general, we don't want to leave these comments in our code for long — but they can be helpful if we are trying to debug something and we really need to build our code and take a look in the browser to see what's going on. For instance, ESLint may be doing an excellent job pointing at an error in our code — but we might need to look at the code in the browser to understand what's going on.

By the way, these comments should *never* be included in a submitted independent project. They aren't portfolio-ready code!

The other option is to change an error to a warning in our `.eslintrc` file. If we do this, we don't have to add comments directly to our code — and these rules will apply *everywhere* for ESLint.

Here's how we can update the rule in our `.eslintrc` file:

.eslintrc

```
"rules": {
  "semi": 1,
  "indent": ["warn", 2],
  "no-debugger": "warn" // new line
}
```

We simply add a line to the `"rules"` object with a key of `"no-debugger"` (the name of the rule) and a value of `"warn"`. This will override the default setting.

Now if we build our code, we'll get a warning instead:

```
6:3  warning  Unexpected 'debugger' statement  no-debugger
```

Our code will build and we'll be able to use our `debugger;` statement in our built project.

Using ESLint Effectively

Here's the general process we should follow if ESLint throws an error:

- Find and fix the error if possible, then build our code as usual.
- If we can't find and fix the error without looking at our built project in the browser, use comments to disable the section of code that is causing problems. Once the error is fixed, remove those comments.
- As a last resort, if we really find that ESLint is being too strict, we can update the rule to give us a warning instead of throwing an error. The name of the rule is always included in

the error that ESLint throws — so we can just put that rule directly in our `.eslintrc` file and change its setting to `"warn"`.

We've found that students will have issues with a wide variety of rules — and we can't troubleshoot or list them all. Use this section to guide you through any rules that cause you issues. You are welcome to customize your `.eslintrc` file to suit you and your coding needs — after all, the purpose of ESLint is to help you write better code, not cause you headaches. You will not be penalized on your independent project for any custom configurations you make in `.eslintrc`.

We recommend looking over at the ESLint Configuration (https://eslint.org/docs/user-guide/configuring) page. There's a lot of information here — so you don't need to absorb it all. Ultimately, it's fine to just use the recommended rule set — and to update rules as needed (but hopefully sparingly).

Now that we've added a linter, ESLint will help us write better code and avoid bugs along the way.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/improving-development-with-a-live-development-server)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/adding-a-production-dependency-bootstrap)

Lesson 18 of 49
Last updated March 8, 2023

disable dark mode