Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Configuration Reference, Suggested Workflows, and Optional Review

Text

We've learned about a lot of new concepts and tools over the weekend homework, so it's time to explain a typical workflow with the new tools we've learned, and share a configuration reference.

In this lesson we'll cover the following topics:

- How to set up the Shape Tracker project from the example repo.
- How to use the tools we implemented in Shape Tracker in a new project.
- Direction for creating a project from scratch with a brand new `.gitignore`, `package.json`, `webpack.config.js`, and `.eslintrc`.
- An overview of concepts and terminology for npm, webpack, and eslint.

It's **optional** to read the overview on npm, webpack, and eslint! It's meant to provide a high-level review of important concepts and terminology. However, you very well may already have reached your capacity for ingesting information and a review may not be helpful now.

If that's the case, we recommend revisiting the overview of npm, webpack, and eslint at the end of this course section to (hopefully) reinforce the concepts behind the new tools you are using this week.

Configuration Reference for the Shape-Tracker Repo

We're providing a reference repository for the Shape Tracker project, which also serves as a configuration reference for npm, webpack, and ESLint. Here's the link:

📂Example GitHub Repo for Shape Tracker (https://github.com/epicodus-lessons/section-5-shape-tracker)

Make sure that you are referencing the code from the branch called `1_functioning_environment`. This is the default branch, so running `git clone...` with the URL of the repo homepage will automatically clone down the branch called `1_functioning_environment`. If you need a review on how to navigate between branches, review the lesson on accessing code from different branches (https://www.learnhowtoprogram.com/lessons/accessing-code-from-different-branches).

Once the repo is downloaded to your computer, you simply need to install packages in order to set it up. In the root of the Shape Tracker directory, run this command to install all packages listed in the `package.json`:

```
$ npm install
```

With this command, npm fetches all of the packages listed in the `package.json` and any dependencies those packages rely on and install all of the source code to your project in the `node_modules` directory.

Since the Shape Tracker repo includes a `package-lock.json`, npm will install all packages and their dependencies at their exact versions listed therein.

Typically we need to run `$ npm install` just once to set up your project. However, if you run into installation issues and need to delete `node_modules` to start the installation process over again, you'll have to re-run `$ npm install`.

Later on, if you wanted to add a new dependency, you'd use npm to install the specific package only: `$ npm install [package-name]`. There's a couple things to note:

- We can include the flag `--save-dev` to save the dependency specifically as a development dependency like so: `$ npm install [package-name] --save-dev`.
- Optionally we can list a version number with the package name, like this example with webpack: `$ npm install webpack@4.46.0 --save-dev` If we don't include a version number, the most recent version of the package will be installed.
- We can include the flag `--save-exact` to direct npm to save the exact version listed for the package and not a different minor or patch version number. We would do so like this: `$ npm install webpack@4.46.0 --save-dev --save-exact`.

After we've installed all packages, we're ready to use the npm scripts that we created and defined in `package.json`:

- `$ npm run build` to build our project in development mode, including bundling our JS and CSS, generating HTML, and linting our code. We build a project when we want to verify that webpack is bundling our code correctly and there are no errors, but we don't actually want to start a development server.
- `$ npm run start` to build our project and then open the webpack development server, which will live re-bundle and reload our project when we make changes to the code in the `src` folder. You'll be using this command the most to start and stop your project's server, which we've also configured to build the project.
- `$ npm run lint` to lint all JS files in the `src` folder. We use this command whenever we want to lint our code. We should do this after we've written new code. Since we've configured webpack to use ESLint, webpack will be linting our code every time it builds it (meaning to bundle and process our source code), so you may find that you don't use this command as often.

Everything in this configuration is reusable for future projects. We'll describe what files to reuse in the next section of this lesson. However, we recommend setting up your configuration from scratch for at least a few projects. This will give you more experience with setting up a development environment.

Applying Our New Tools to a New Project

While it's important to practice installing dependencies with npm, and manually setting up a `package.json` file, `webpack.config.js` file, and `.eslintrc` file, it would be painful to have to do this for every project.

To use the same tools we used in the Shape Tracker project in a new project, you'll need to have all of the configuration files for npm, webpack, and ESLint. The only files that will change are those in the `src` folder, as long as the naming convention stays the same.

If we wanted to create a template repository that contained the basic configuration (which we'll learn how to do later in this section), we'd include the files and folders that are listed in the example below for a repo called `template-repo`. The file structure shows only the completed configuration files you must include in your project, and NO auto-generated files, or files that should change from project to project. As you review the example, pay attention to the comments.

```
template-repo/
├── src/
│   ├── // Our source code goes in src/
│   ├── // The JS, CSS, and HTML files in here change from
project to project
│   ├── index.js // We always need to include the entrypoin
t JS file called index.js
│   └── index.html // We always need to include a template
HTML file called index.html
├── package.json  // In this file we need to update the "na
me" key to the name of the project
├── webpack.config.js
├── .gitignore
├── .eslintrc
└── README.md
```

As long as our project contains the above folder and file structure, including the completed configuration files, we are ready to go to use npm and webpack:

- To set up the project, we'll need to run `$ npm install` to install all packages. We'll do this once unless we run into errors and have to delete the `node_modules` folder and reinstall dependencies.
- To instruct webpack to build the bundle, generate the HTML, and lint our JS, we'll run `$ npm run build`.
- To build our project (as described in the last bullet point) and serve it with the webpack dev server, we'll run `$ npm run`

`start`.

After completing this setup, we're ready to write code in the `src` folder.

A few notes about the example `template-repo/` file and folder structure:

- In the `src` file, you'll still need a template HTML file, and a JS file called `index.js` to serve as the entry point for webpack to make a bundle. You'll also still need to import other JS and CSS files into `index.js`.
- What's NOT included in the file tree above are all of the auto-generated files and folders:
  - The `dist/` folder with the generated HTML and bundled code
  - The `node_modules/` folder with the source code for the packages installed by npm
  - The `package-lock.json`, which is created by npm to list the exact versions of all dependencies installed to our project with npm.
- In `package.json`, you'll need to update the `"name"` of your project.

Creating a Project from Scratch

If you are creating a project from scratch, including a `package.json`, `webpack.config.js`, and `.eslintrc`, remember to create your `.gitignore` and commit it to your Git history as the very first step! Your `.gitignore` should include these files:

.gitignore

```
node_modules/
.DS_Store // only include this if you are on a Mac
dist/
```

It will be easiest to follow the weekend homework in order to recreate a project setup that uses npm, webpack, and eslint. Here's the basic steps to follow:

- Create a `.gitignore` with all files or folders that should be ignored.
- Create a `package.json`, whether you copy one or use `$ npm init -y` to generate a new one via the command line.

From here on, it doesn't matter whether you install and configure webpack, ESLint, or Bootstrap first.

As far as webpack goes, it really doesn't matter what order you follow to install the plugins and loaders, but they individually do have an effect on the location, naming, and contents of your source code, so you'll likely find it easiest to follow along with the weekend homework.

Also note that while you could install and configure ESLint before webpack, you can't configure the ESLintPlugin for webpack until webpack itself is installed and configured.

Optional: npm Overview

npm stands for node package manager, which comes installed with Node. With npm we can install, modify, and uninstall packages that we want to use in a project. At its most simple, a **package** is simply an external JavaScript library — a set of code that we can download and use in our project. In technical terms, a package (https://docs.npmjs.com/about-packages-and-modules#about-packages) is a file or folder that's described by a `package.json` file.

The `package.json` file contains all of the information about our project's metadata: its name, version, entry point, dependencies, scripts, and more. We can use a `package.json` like a manifest that describes everything about the project and everything that we need to run the project.

When a package is installed with npm, the source code of the package, and any dependencies that package relies on, gets added to the `node_modules` folder. This reminds us of two terms:

- A **dependency** (https://nodejs.dev/learn/npm-dependencies-and-devdependencies) is any code that other code relies on for its functionality. We can install a package as a dependency in the projects we create, and those packages can have dependencies or their own.
- A **module** (https://docs.npmjs.com/about-packages-and-modules#about-modules) is any JS file or folder in the `node_modules` directory that can be loaded by npm into our projects. In simpler terms, a module is a unit of functionality.

Workflow and Commands

We use npm scripts to run packages installed in our project.

To run packages installed in our project, we can write an npm script. For example, we've set up a `"lint"` script that calls on ESLint to lint our JS files in the `src` folder.

package.json

```
...
  "scripts": {
    ...
    "lint": "eslint src --ext .js"
  },
...
```

When we enter `$ npm run lint` in the root of our directory, npm runs `eslint src --ext .js` in the terminal, which invokes the ESLint package from within the `node_modules` folder to only lint JS files that are in the `src` folder, including subdirectories within `src` .

The npm script for ESLint is just one example — we can set npm scripts to invoke other libraries or tasks.

We also use npm commands to install project dependencies.

As long as you have a completed `package.json` file, to set up your project and install all packages, simply run the following command in the root of your project:

```
$ npm install
```

If you are building a project from the ground up, or managing individual packages, you'll use commands to install and uninstall packages:

```
$ npm install [PACKAGE-NAME]
```

```
$ npm uninstall [PACKAGE-NAME]
```

When installing packages, we often include a version number. npm packages use **semantic versioning**. Review the lesson on semantic versioning (https://www.learnhowtoprogram.com/lessons/semantic-versioning) if you need a refresher.

You can also manually remove or add a package to the list of dependencies in `package.json`, and then ensure that your `node_modules` and `package-lock.json` are updated by running this command:

```
$ npm prune
```

Optional: webpack Overview

webpack is a module bundler. A **module** is a JavaScript file, CSS file, image, or other asset. As a **bundler**, webpack takes multiple files and concatenates them into a single bundled file. In the process, webpack optimizes the bundle for speed and efficiency.

When webpack bundles modules, it creates a dependency graph, which is a representation of all of the files in our source code and how they are connected. webpack starts by loading the entry point `index.js`, then webpack loads any files that are imported into `index.js` (listed at the top of the file), and then webpack loads any files that are imported into those files, and so on and so forth.

For webpack's dependency graph, a **dependency** is a file or package that is used in the source code of our project, in the `src` folder. The dependencies in the Shape Tracker project include `index.js`, `triangle.js`, `css/styles.css`, and the two Bootstrap files and anything that Bootstrap relies on. (If you are wondering, "what about the HTML?", remember that HTML is not bundled with webpack.)

This is in contrast to an npm dependencies, which include a larger list of packages that we've downloaded to our project, like webpack.

webpack's core functionality is to concatenate and minify JavaScript files. However with the help of plugins, loaders, and additional built-in webpack configurations, webpack can handle much more:

- Bundling CSS with our JS. For this we use the loaders `css-loader` and `style-loader`.
- Generating HTML files. For this we use `HtmlWebpackPlugin`.

- Linting our code, and clearing out old files. For this we use `ESLintPlugin` and `CleanWebpackPlugin`.
- Improving our developer experience with source maps and a developer server. For source maps we use our browser's DevTools and a built-in webpack configuration for `devtool`. For the developer server we use `webpack-dev-server`, which is not a plugin, but an npm package that we can configure webpack to use.
- And so much more that we did not learn about.

We install loaders and plugins with npm, and we configure them in `webpack.config.js`.

To enable webpack to bundle our code, we specify an entry point in our source code (`index.js`), and we connect any files into that entry point that we want bundled. To connect files together we export a file and import it into another file. If you need a review on importing and exporting files, revisit the lesson ES6 Imports and Exports (https://www.learnhowtoprogram.com/lessons/es6-imports-and-exports).

In order to invoke the functionality of webpack and webpack's dev server, we've set up npm scripts in `package.json` that run commands in the terminal to bundle our code and server it all in development mode.

Students who previously used Live Serve to serve their projects should now use webpack development server via the `"start"` npm script: `$ npm run start`.

Optional: ESLint Overview

ESLint is called a **linter** that looks for errors in our code, as well as poorly written code that doesn't meet conventions. ESLint is specifically for JavaScript, and it can vastly improve our development experience by pointing out issues that we otherwise might easily miss, like a missing semicolon.

With webpack's ESLintPlugin, we've configured webpack to run eslint on our JavaScript files, every time we build our project. This is super helpful, because it fully automates and integrates the tool!

However, we've also set up an npm script called `"lint"` in order to invoke ESLint whenever we need to, with `$ npm run lint`.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/adding-a-production-dependency-bootstrap)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/journal-5)
Lesson 20 of 49

Last updated March 8, 2023

disable dark mode

Epicodus (http://www.epicodus.com)