Lesson | Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Installing Dependencies with npm: webpack and webpack-cli

Text

We're now ready to start installing the packages we'll work with. We can install any npm package with the `$ npm install [PACKAGE-NAME]` command where `[PACKAGE-NAME]` is the package we want to install.

As we learn how to use npm to install packages, we'll also learn about:

- Version pinning and version ranges.
- `package-lock.json`.
- Uninstalling packages.
- The difference between `dependencies` and `devDependencies`.
- Using the `npm install` to direct npm to install packages listed in a `package.json` file.

As you work through this lesson, make sure you actually install the two packages to your Shape Tracker project: webpack and webpack-cli.

Version Pinning

Most of the packages we'll install specify a version of the package. This is called **version pinning**. When a package is updated, it won't necessarily play nicely with all the other packages in our environment. This is exactly what we were talking about in the last lesson! For that reason, it's more important to create a stable environment where all packages work together instead of a potentially chaotic environment where we are always installing the latest version of a package without fully testing whether it works with other packages we're using. The latter is a recipe for breaking our code — and a very frustrating debugging situation.

Version pinning is commonly used throughout the industry. As stated in the last lesson, companies generally do not automatically upgrade to latest releases, especially if the version they are already using is reliable and they don't need any new features that more recent versions offer.

While you are welcome to explore JavaScript packages further on your own, you are expected to use the pinned versions for independent projects. After all, if you were working at a tech company, you'd be expected to use the company's environment (which would likely be consistent across teams) instead of the other way around!

Installing npm Packages

Now we're ready to install our first package: webpack. We'll cover webpack itself further in the next lesson. For now, we are just learning how to install packages with npm. In the root of the Shape Tracker project, go ahead and run this command:

```
$ npm install webpack@4.46.0 --save-dev --save-exact
```

We specify a version with `@`. The version itself is `4.46.0`. That's major version 4, minor version 46, and patch 0. We also have two flags `--save-dev` and `--save-exact` — which we will explain in more detail below.

Additional `npm install` Commands

We could also install webpack without a specified version number with the following command.

```
$ npm install webpack --save-dev --save-exact
```

However, we want to pin the version of webpack that we're using so we don't run into versioning issues between our dependencies, where one dependency can't work with another dependency.

Finally, we can also run the following command, where we don't specify any package:

```
$ npm install
```

What npm will do is look inside of `package.json` and install any dependencies listed. We'll use `npm install` anytime we clone down our project and need to install all of our project's dependencies. For the next few lessons, we'll focus on installing specific packages.

How npm Installs the Package to our Project

When we run `npm install` (with or without a package), three things will happen for us automatically:

npm will add the package to a directory named `node_modules`. If the directory doesn't exist yet, npm will create it for us. Note that you should never create the `node_modules` directory yourself, and we recommend against editing it. Editing `node_modules` directly is a less common practice that some experienced developers occasionally do.

After running the command to install webpack, go ahead look inside the `node_modules` directory. You'll see a lot of packages have been installed, not just webpack! These are all the packages that webpack itself depends on — and they've been added for us automatically by npm.

Remember that we add `node_modules` to our `.gitignore`. The `node_modules` directory can get very big and it would really bloat our repository to include this code. And because `node_modules` is automatically created there's just no reason to add it to our remote repo. All that we need to install these packages locally in our project is listed in our `package.json` file.

npm will add the name and version number of the package to our `package.json` file. As we mentioned previously, if there is no package specified when we run `npm install`, npm will install all of the packages listed in `package.json`.

After we install webpack, if we take a look at `package.json`, we'll see the following has been added automatically for us:

```
"devDependencies": {
    "webpack": "4.46.0"
}
```

Finally, npm will add our new dependencies to a file called `package-lock.json`. This file should never be edited either — that's why part of the file name is `lock`. If we open it up, we will see that it

already has thousands of lines in it. This is the tree of our project's current dependencies all at their exact versions. Again, it's not just webpack, but also all the dependencies that webpack depends on.

The purpose of `package-lock.json` is to list the exact versions of all dependencies in our project. While we'll manage the packages we need in our projects via `package.json`, just like we did with webpack, npm handles listing all of webpack's dependencies at their exact versions in `package-lock.json`.

So, every time we add a new package with npm, the `package-lock.json` will get updated with that package's dependencies and the exact versions they were installed with.

One big thing to note is that **we do not add `package-lock.json` to the `.gitignore`**, even though it is an auto-generated file. This is because the results of installing packages can vary over time. This means that even though we're installing webpack at version `4.46.0`, some of webpack's dependencies may not be installed at specific version numbers, which of course can lead to bugs. `package-lock.json` solves this issue by listing exact (and working) versions of all dependencies, and npm knows to use this file to install dependencies when it is present in a repo. That's why we always save `package-lock.json` in our remote repos.

Install Flags `--save-exact` and `--save-dev`

Now let's take a look at the flags we added to our `$ npm install` command.

- `--save-exact`: This flag ensures that we save the *exact* version of webpack to our list of dependencies in `package.json`. This is very important and we'll discuss this further in a moment.
- `--save-dev`: This flag specifies that we want the package to be a development dependency. It will be added to `"devDependencies"` in `package.json`. If we don't add this flag, it will automatically be added as a dependency that's available

for production and development. While everything will still work correctly if we add webpack as a production dependency, we don't need webpack in production — and having unnecessary dependencies would just bloat our production code.

Installing `webpack-cli` and Version Ranges

We also need to install a package to have access to the **CLI (command line interface)** for webpack. This package will allow us to use webpack from the command line. In the root of the Shape Tracker project, go ahead and enter this command now:

```
$ npm install webpack-cli@3.3.12 --save-dev
```

This time, we have not included the `--save-exact` flag, which we'll discuss below.

Let's take a look at `package.json` again to see how these commands have changed the contents of the file:

package.json

```
{
  "name": "shape-tracker",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "4.46.0",
    "webpack-cli": "^3.3.12",
  }
}
```

Note the difference in the package numbers between webpack and webpack-cli. The latter has a caret symbol `^` before the version number. Well, this symbol and the tilde symbol `~` can really mess us up, because they specify an acceptable **version range** within the version we specify in the installation command. Here's how this works.

- The caret `^` symbol tells npm to install the package at the version we specify, or a later minor version. This means that npm could install webpack-cli at version 3.3.12 or 3.4.12.
- The tilde `~` symbol tells npm to install the package at the version we specify, or a later patch version. This means that npm could install webpack-cli at version 3.3.12 or 3.3.18.

For the webpack-cli package, if npm installs the latest minor version of a package and it isn't compatible with one of our other packages, it will cause errors. We see it happen all the time to students.

When we use the `--save-exact` flag, the `^` and `~` symbols will not precede the version number — so we won't have this issue. This begs the question — why would we not always use `--save-exact` to

pin dependencies to their exact version? Well, when we specify a version range for a package, we can get updates for it, which can include fixes to security vulnerabilities, ones that we may not learn about for a while. So, always pinning exact version numbers is not always the best approach.

If you do find yourself having version incompatibilities in your project, this is the first thing you should check — are there any `^` or `~` that result in slightly different packages being installed? (You may have forgotten to use the `save-exact` flag originally — or you may be working off a `package.json` file that didn't use them.) This is the most common issue students run into when packages that should be playing nicely together are not working correctly.

If this happens, remove the caret `^` and tilde `~` from version numbers and then follow the steps in the next section of this lesson.

Removing Dependencies

Sometimes, you may want to remove a dependency from your project — or you may want to install a different version of the dependency. We'll cover a couple ways to do this now, but don't actually uninstall any dependencies.

To remove a dependency, we can use npm's `npm uninstall` command. To use it, we can specify the name of the package we want to remove, like so:

```
$ npm uninstall webpack-cli
```

We can also adjust and remove packages manually.

If you want to completely remove a dependency manually, first remove the reference to the dependency you want to remove from `package.json`. For instance, if we wanted to remove webpack, we'd

actually delete the line from the `package.json` file.

If you just want to just change the version number, or remove the `^` or `~`, you can also do so manually by editing `package.json`.

Anytime you change a dependency in `package.json`, you need to make sure that npm applies the changes with the command `$ npm prune`. This command ensures only dependencies listed in the manifest are actually installed and at the correct versions (while others are removed).

However, we've found that `$ npm prune` doesn't always get the job done. Fortunately, there's a more foolproof way to reinstall dependencies: trash the `node_modules` folder altogether. Then run `$ npm install` again. npm will automatically repopulate the folder and all its dependencies.

Whenever you are having a problem with your development environment and you are trying to reinstall dependencies, make sure you follow these troubleshooting steps!

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/semantic-versioning)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/introduction-to-webpack)