npm Code-Along

 (https://github.com/learn-co-curriculum/react-hooks-npm-lab)  (https://github.com/learn-co-curriculum/react-hooks-npm-lab/issues/new)

Learning Goals

- Use a `package.json` file to manage project dependencies
- Install a project dependency using npm
- Import code from a package into a JavaScript file

Introduction

When using npm, it is often the case that we aren't familiar with *all* of the code in the dependency tree. Building modern JavaScript applications relies on our ability to use the tools built for us by others. As it turns out, most of those tools are *also* built using *other people's* tools. One package may be used in another, which is used in another, and another, and so on...

Using npm, we download specific packages of code. If those packages have dependencies, the dependencies are also downloaded in a recursive manner. For the purposes of our own application, however, **we only need to know about the node packages *we* specifically need to get our app working**. We don't need to worry about what packages *those* packages need. Why? Because every node package includes a `package.json` file that lists out all dependencies. This file lets Node know what to download when we run `npm install`. Node will download all the packages, check the `package.json` files present in each of those packages, download any additional packages, and repeat.

We will see in future labs that as the number of packages increases, more and more happens when we run `npm install`. All *we* need to worry about, though, is the top level — what is listed in *our* application's `package.json` file.

In this code-along, we are going to practice the process of setting up a `package.json` file. We will also install an npm package or two and use their functionality in new code we write.

Getting Started

Before we create our `package.json` file, take a moment to look at the `package.json` file that is already at the top level of the directory for this code-along. If you look at the "name" attribute at the top, you will see that it is the same as the name of the directory, `react-hooks-npm-lab`. This is the `package.json` file that belongs to this code-along; you **should not** make any changes to this file! Instead, we will build out a simple application **within** this lesson's directory and create a `package.json` file for that application.

Note that this lesson's files include a sub-folder, `color-clock`, that contains some basic starter files for a project. If you look at `color-clock/index.html`, you'll see a script tag:

```
<script src="index.js" type="module"></script>
```

Taking a look inside `index.js`, we can see that this script relies on a unique function call, `format(new Date(), "MMMM do yyyy, h:mm:ss a")`. We're also **importing** that function from a `node_modules` folder that contains a date formatting library called `date-fns`. Our goal is to get this code working. **We do not need to change** `index.js`. Instead, we will need to set up a `package.json` file and install the **`date-fns`** (https://www.npmjs.com/package/date-fns) package.

Navigate to the Project Directory

The first thing to do is change directory into this folder in your terminal by typing the command `cd color-clock`.

The next step is to create a `package.json` file in the `color-clock` directory, which in turn will be where the `node_modules` folder is.

Important: to avoid overwriting the `package.json` file for this code-along be sure to change directory into `color-clock` before creating the `package.json` file!

Create a package.json File

The `package.json` can be written quickly from scratch, but we actually have a handy command for creating these files: `npm init`.

Run `npm init` and you will be prompted to confirm the information that will be stored in `package.json`, starting with the name of the project.

Most prompts will provide a default value. Some are blank and can be left this way for now. Follow the prompts by pressing enter in the terminal on each prompt until you reach the end, when you will be prompted to type 'yes' to confirm. A fully constructed `package.json` file will then appear in the `color-clock` directory.

Add a Script

In the process of creating the `package.json` file, you were prompted to write a test script. We left it blank at that time, but we can add it to the `package.json` file ourselves. Let's do that now to see how this works.

Open the newly created `package.json` file and look for a section titled `"scripts"`. Let's replace the default `"test"` script with a shell command:

```
"scripts": {
  "test": "echo 'Hello World!'"
}
```

We can now call this script and have it run by using the command `npm test` in the terminal (if that doesn't work, try `npm run test`). You should see a printout of `Hello World!`.

In all the JavaScript-based labs you've encountered so far, this sort of script is how we run tests. If you look at the `"test"` script on JavaScript labs in the previous phase, most will have something like this:

```
"test": "mocha -R mocha-multi --reporter-options spec=-,json=.results.json"
```

The mocha command is actually a command that you can run in the terminal. This is a call to the testing package, `mocha`, along with a second package, `mocha-multi` that helps with reporting. When you run `npm test` in a lab, the command specified in the "test" script is what gets called.

Scripts are often useful for things like testing or to start a necessary process, like a local server.

Install a Package

With `package.json` set up, we can now add a package we want to include in our project.

Now, we're building a colorful clock — the project is simple enough that we *could* build it entirely out of custom code. Here's the thing though: one of the reasons packages exist and are so useful is because programmers often run into the same problems over and over. Node packages are written so we don't have to recreate a solution to a problem other programmers have already solved.

In the case of a colorful clock, we have to deal with formatting time. This is such a common problem, that a package has been created to help us: `date-fns`. `date-fns` is a handy package that comes with a number of functions that make displaying dates and times simpler than trying to figure out JavaScript's built-in functions.

Let's install `date-fns` and incorporate it into our clock. To install a package and save it to your `package.json` file, run `npm install` followed by the package name. In our case, that would be:

```
$ npm install date-fns@2.30.0
```

NOTE: We are specifying 2.30.0 as newer versions of date-fns will not work with this lab. We can always specify the version of our packages using `@`!

This command will add the package to the list of dependencies in `package.json`. When `npm install` is run, all dependencies are installed. If you were to publish this repository on GitHub, other users would now be able to clone down the repo and install whatever is listed in `package.json` to get the program working.

The second package we'll need to run our application in the browser is **serve** (https://www.npmjs.com/package/serve), which will run a lightweight server. To install it, we run:

```
$ npm install serve
```

Next, in the `"scripts"` section in `package.json`, let's add an npm script to run the server using the `serve` package:

```
"scripts": {
  "test": "echo 'Hello World!'",
  "start": "serve"
}
```

If you run `npm start` to run the script and open **localhost:3000** (http://localhost:3000) in the browser, you will see that the clock is not appearing. Go ahead and open the console and you'll see that we're getting an error:

```
Uncaught TypeError: Failed to resolve module specifier "@babel/runtime/helpers/esm/typeof". Relative references must s
```

The specifics of this error are beyond the scope of this lesson, but basically what it means is that not all of the files in our project are currently set up to be interpretable by the browser. Before we can get our clock running correctly, we need to install one more tool, `esbuild`. `esbuild` is a JavaScript bundler, which is a tool that handles all of a project's dependencies, and combines the code into a single file that is browser-ready. There are a number of different JavaScript bundlers available; we're using `esbuild` because it is relatively easy to configure and works fine for our simple application.

Stop the server with `ctrl-c`, then install `esbuild`:

```
$ npm install esbuild
```

Then we'll add one more script to run the build:

```
"scripts": {
  "test": "echo 'Hello World!'",
  "start": "serve",
  "build": "esbuild index.js --bundle --outfile=dist/out.js"
}
```

When we run a build using `esbuild`, it makes sure that all the dependencies are included and up to date, and combines the code from multiple files into a single file that is ready to be loaded in the browser. Note that the name of this file is specified in the build command above: `dist/out.js`.

Go ahead and run `npm run build`. You should now see the `dist` folder in your file tree and the `out.js` file inside it. The final step is to update the script in the `index.html` file to use this new file. Find this line:

```
<script src="index.js" type="module"></script>
```

Change the `src` property to `dist/out.js`. Now we're finally ready to start the server. Run `npm start` then open up **localhost:3000 (http://localhost:3000)** in the browser. You should now see a colorful clock appear!

Conclusion

When building our own applications, we will often rely on existing packages to handle specific pieces of a project. Although we only installed a couple of packages for this code-along, there were additional layers of dependencies for them so many additional dependencies were installed as well. It isn't necessary to understand *how* each of these works. The main thing to grasp is how to implement and use the specific dependencies you need.