



npm Code-Along

[New Attempt](#)

Due No Due Date **Points** 1 **Submitting** a website url

FORK

[_ \(https://github.com/learn-co-curriculum/react-hooks-npm-lab/fork\)](https://github.com/learn-co-curriculum/react-hooks-npm-lab/fork)  [_ \(https://github.com/learn-co-curriculum/react-hooks-npm-lab\)](https://github.com/learn-co-curriculum/react-hooks-npm-lab)  [_ \(https://github.com/learn-co-curriculum/react-hooks-npm-lab/issues/new\)](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 can create a `package.json` file, we'll need a project and a project folder to contain all the files. For this code-along, we'll be building out a clock application that changes color

 **Help**

In this lesson, a sub-folder has been created for us to use, `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` 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`.

Note: The next step will create a `package.json` file in whatever directory you are in, which in turn will be where the `node_modules` folder is. If you do not change directory into `color-clock`, you'll end up creating a file in the main directory of this lesson, and `color-clock/index.js` will be looking for `node_modules` in the wrong directory.

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. Let's add a working script in 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 print out 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 previous labs, most will have something like this:

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

This 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 `learn` or `learn test` in a lab, `npm test` 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 re-find a solution other programmers have found.

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
```

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.

We'll also need one more package to run our application in the browser. Run this command:

```
$ npm install serve
```

This `serve` (<https://www.npmjs.com/package/serve>) package will run a lightweight server. We can set up another npm script to run the server using the `serve` package:

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



```
"start": "serve"  
}
```

Run `npm start` to run this script, and open up localhost:5000 [_\(http://localhost:5000\)_](http://localhost:5000) in the browser.

If `package.json` file has the correct package, and the node module has been installed, you should 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.