Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Creating a package.json with npm

Text

Our first step in creating our project's development environment is to set up Node Package Manager (npm) by creating a `package.json` file. This file lists metadata about a project — its name, version, dependencies (packages used in the project), scripts, and more. In short, we use `package.json` to manage a project's dependencies.

npm (https://www.npmjs.com/) is a package manager for adding open-source JavaScript libraries and packages to our applications via the command line. And yes, that's not a typo — neither npm and webpack are capitalized. A **package manager** is exactly what it sounds like: a tool to help manage (install, upgrade, and configure) all the outside tools and libraries (also known as packages or dependencies) a project requires.

npm originally started as a package manager for Node.js (used for server-side or "back-end" JavaScript), but now it's also commonly used for front-end (client-side JavaScript that uses the browser)

projects as well. Our projects are entirely front-end because all of our code will be loaded in the browser. We will not cover server-side JavaScript (Node) in this course.

There's another very popular package manager called Yarn (https://yarnpkg.com/). You'll likely see many references to both yarn and npm when you are looking through online resources. In some ways, yarn improves on npm, but npm is the older, more established standard.

You should already have Node installed (https://www.learnhowtoprogram.com/intermediate-javascript/setting-up-javascript/installing-node-js) on your computer, which comes with npm. If not, install Node now, as we will need it to use npm. To check if you have node and npm installed, run these commands in the terminal:

```
$ node -v
v16.14.0
$ npm -v
8.3.1
```

Note: you don't have to have the same versions listed in the code snippet.

As you work through this lesson, make sure you run `npm init -y` to create a `package.json` file in your Shape Tracker project.

Setting Up `package.json`

We will need a `package.json` file for *every* project going forward. Over the coming lessons, we will manually add all the packages we need in our `package.json` file. However, once we have our `package.json` file all set up, we can reuse it for other projects, tinkering with it and customizing it as necessary.

Open the command line, then navigate to the root directory of `shape-tracker`. Then type in the following command:

```
$ npm init -y
```

This automatically creates a file called `package.json` in our project's root directory.

We can also run `$ npm init` without the `-y` flag; this creates a command line prompt with questions that you answer to create a basic `package.json` configuration. However, it's just as easy to edit configurations in the `package.json` file directly. All `$ npm init -y` is doing is automatically initializing this file for us — without us doing any custom configuration first.

A quick note: we won't always use `$ npm init` to set up our `package.json` file. In the future, we'll reuse the `package.json` file from project to project, changing small details like the project name. This effectively means we can copy/paste a `package.json` file from another project instead of initializing a new `package.json` file and manually adding packages. This will become clearer as we get more familiar with the process.

Let's open the created `package.json` and take a look:

shape-tracker/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"
}
```

So far, the file is very basic and only includes information like the `"name"` and `"version"` of our project. There are no outside packages yet and we can update any of these values if we wish. However, we won't make any changes to this file just yet.

Note that when we use `npm init -y` the `"name"` of the project is set to the name of the directory we are in. When we reuse a `package.json` file in another project, you'll need to update the `"name"` to reflect your project's name.

Why Do We Need This File?

Before we move on, let's briefly cover the *why* of `package.json`. As we install packages to our project with npm, the name and version number of each package we install will be added to `package.json`. Let's look at an example of this. By the end of the section our Shape Tracker's `package.json` file will look like this:

```
{
  "name": "shape-tracker",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "webpack --mode=development",
    "start": "npm run build && webpack-dev-server --open --
mode=development",
    "lint": "eslint src --ext .js",
    "test": "jest --coverage"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.18.6",
    "@babel/plugin-transform-modules-commonjs": "^7.18.6",
    "clean-webpack-plugin": "^3.0.0",
    "css-loader": "^3.6.0",
    "eslint": "^8.18.0",
    "eslint-webpack-plugin": "^2.7.0",
    "html-webpack-plugin": "^4.5.2",
    "jest": "^24.9.0",
    "style-loader": "^1.3.0",
    "webpack": "4.46.0",
    "webpack-cli": "^3.3.12",
    "webpack-dev-server": "3.11.3"
  },
  "dependencies": {
    "bootstrap": "^5.2.3",
  }
}
```

Soon, we'll cover all of the details in `package.json`. For now, notice the `"devDependencies"` and `"dependencies"` keys — these locations are where all of our packages are listed!

This list of packages provides a set of instructions for anyone that wants to replicate our project (including our future selves). We simply need to clone the project and tell npm to install all of the packages listed in `package.json` using a single command (we'll learn about this in the next lesson), and npm will handle installing all of the packages. This means that there's no need for us to install each package manually! This is a huge deal, especially when a project has dozens or even hundreds of packages.

The `package.json` file also provides some other very helpful functionality — the ability to add our own `"scripts"`. This allows us to run custom commands in the terminal through npm, like serving our project and running tests. This can make our lives much easier as developers. We'll cover this in depth in future lessons.

Now that we have a sense of why this file is important, we're ready to start using npm to install our first dependencies. First, though, we'll learn about the basics of versioning in the next lesson. Versioning is a really important part of setting up a stable JavaScript environment.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/git-best-practices-and-adding-a-gitignore-file)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/semantic-versioning)

Lesson 6 of 49
Last updated March 8, 2023