Javascript Outside of the Browser

npm

NodeJS is capable on its own for a myriad of tasks, but you'll often want to reuse a module or script that someone else has written in order to simplify your own project. In addition, you may want to publish your own script for others to use. These tasks are done with npm.

npm stands for Node Package Manager. It is a command-line tool that can be used for preparing, installing, and publishing packages, or bundles of code, in NodeJS. It also facilitates running test and build scripts.

npm is an immensely powerful tool. For our purposes, we're going to focus on creating a package.json file, and installing local dependencies. This will allow us to install Webpack and Babel in future sessions, which will be required in order to make use of ES6 features.

If you installed NodeJS from the official website, npm comes bundled with it. Before we continue, verify that you have npm installed by running this in the console:

```
npm --version
```

The version of npm we're using in this tutorial is 6.9.0. If your version is close to that, then it should be fine.

We'll start out by manually writing a package.json file in your project's directory. The package.json file specifies information about a NodeJS project, including the name, version, main entry point, and dependencies. npm uses this file to install dependencies, run scripts, and publish the NodeJS package.

As the name suggests, package.json must be a valid JSON file. Here are some of the properties that npm uses to determine what to do. Most of these properties are only required if you intend on publishing a package, but you should get in the habit of defining at least these properties anyway.

The `name` property defines the name of the package. It has a max length of 214 characters, and must not contain any uppercase letters, spaces, or any characters that are not "URL-safe". A common convention is to use kebab-case or caterpillar-case, like this:

```
{
  "name": "my-package"
}
```

The name is required in order to publish packages.

The `version` property is a string defines the version of the package in semver (specifically, node semver) format. In a nutshell, that means the version is determined by 3 numbers seprated by dots. The first number is the major version number, or versions that change the component in such a way that it is no longer backwards compatible, the second number is the minor version number, which increments when functionality is added but the change is still compatible with the last major update. The last number is the patch version number, which increments whenever bug fixes are made. There are other details to semver but that should cover the basics.

This is the first release, so "1.0.0" should suffice.

```
{
  "name": "my-package",
  "version": "1.0.0"
}
```

The `description` property is a string that briefly describing the purpose of the package to humans.

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "My sample Node package"
}
```

The `main` property defines the path to main entry point of the package, relative to the project's root. If you deploy a package to npm, this script will be run when it is "required" via the `require()` function. It has uses even if you don't intend on publishing your package though: the script defined by the `main` property will be executed if you run `node .` in the console, at least on a POSIX system.

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "My sample Node package",
  "main": "index.js"
}
```

The `scripts` property is an object that defines scripts that npm can run. This is where you would define scripts for running tests, building for the web, or starting a web server. The key in the `scripts` object is the name of the script to run, and the value is the command to execute. For example, if I wanted to make an npm command to echo a message to the console, I could write:

```
{
  "scripts": {
    "hello": "echo \"Why hello friend!\""
  }
}
```

And then run:

```
npm run hello
```

Notice that the value is written in Bash script, or if your on a Windows machine, Windows Batch script. You need to be very careful when writing scripts here, because if you use any commands that are not compatible with either Bash or Batch, your package will only work on certain systems. For that reason, you'll typically only want to write a script that executes a NodeJS script, like this:

```
{
  "scripts": {
    "hello": "node hello.js"
  }
}
```

`hello.js`:

```
console.log('Why hello friend!')
```

The `dependencies` property defines an object that lists all the external dependencies your package has. Specifically, it defines any packages that should be downloaded when your package is installed, or if your package is included in another package. The key in the `dependencies` object is the name of the external package that exists in the npm public repository. The value is the version to use, which can either be the specific version or the version within a particular range. We'll get into this a little later.

There are other crucial properties in package.json that you'll want to define if you intend to publish your package, including `author`, `license`, `repository`, and more. Details about each property can be found on npmjs.com. For the purpose of this session though, this is enough to get started.

Now that you know about some of the crucial properties in package.json, I'll introduce a helper tool that comes bundled with npm for generating the package.json file automatically: `npm init`.

Running `npm init` in the console runs a small helper program that asks a few questions in the console and generates minimal package.json file. Just fill in the prompts and press enter/return to go to the next option. Most of the options are optional, and pressing enter/return without filling something in will apply the default value. If you don't know what to enter, just press enter.

Now that we have a package.json file, we can install external dependencies. Try running the following command in the console to install the cowsay module:

```
npm install --save cowsay
```

If the installation was successful, you should see some new files in your package's folder. For one, a `node_modules` folder is created. It contains the cowsay module and all of its dependencies. This folder is not included if you publish your package, and it should not be included in source control.

A `package-lock.json` file is also created. This file defines a dependency tree so that whenever you reinstall components (via the `npm install` command), the same dependencies are downloaded. You will probably never need to edit this file, and it is not published with your package, but do include this file if you are maintaining your project in source control. Otherwise, other developers could install your package's dependencies and have different results.

Now that the cowsay module is installed locally, we can use it in our main script by including it via `require`. You'll include it just as you would with a global module like `fs` or `path`:

```
const cowsay = require('cowsay')
```

According to the documentation, cowsay generates ASCII art of a cow saying something, which can then be output into the console. I'll make it so it generates a cow saying "Hello!" with its eyes closed:

```
const cowsay = require('cowsay')

console.log(cowsay.say({
  text: 'Hello!',
  e: '^^'
}))
```

So, we've gone over what npm is, and how it can be used to install dependencies, run scripts, and publish packages. We also went over some of the crucial details in `package.json` files. The purpose and importance of the node_modules and package-lock.json files was briefly covered. And we showed how to include an installed dependency in a script.

Next we'll cover Webpack, and how it can bundle multiple scripts into one file.