npm Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Explain what "npm" stands for.
2. Explain the purpose of the `package.json` file and `node_modules` directory.
3. Given multiple choices, identify the difference between npm's `package.json` and `package-lock.json` files.
4. Use `npm --version` to check what version is currently installed and use npm to update itself to the latest version.
5. Use `npm init` to create a new package and `npm install` to add a package as a dependency. Then use `require` to import the module and utilize it in a JavaScript file.
6. Given a package version number following the MAJOR.MINOR.PATCH semantic versioning spec that may include tilde (~) and caret (^) ranges, identify the range of versions of the package that will be compatible.
7. Explain the difference between a dependency and a development dependency.
8. Given an existing GitHub repository, clone the repo and use npm to install it's dependencies.
9. Use `npm uninstall` to remove a dependency.
10. Use `npm update` to update an out-of-date dependency.
11. Given a problem description, use the npm registry to find a reputable package (by popularity and quality stats) that provides functionality to solve that problem.
12. Given a package with vulnerabilities due to outdated dependency versions, use `npm audit` to scan and fix any vulnerabilities.
13. Write and run an npm script.

Free Same-Day Delivery: Package Managers & npm

So far, you've written lots of code yourself. Think of all the other developers out there writing code by themselves as well. Wouldn't it be great if we could share that code and all work together - without needing one **giant**office space? Lucky for us, we don't need to share desks: we've got the Internet!

Let's discuss *packages*and *package management*, a simple way of sharing working code across time & space.

We'll cover:

- defining "packages" in relation to programming;
- understanding package management;
- and using npm to manage packages in our JavaScript projects;

Package management

It's rare that you'd prepare a meal by grinding grain to make flour, or that you'd hatch your own chickens just to get some scrambled eggs! Most industries have learned to bundle the work of others into off-the-shelf goods, like a loaf of bread or a dozen eggs, that everyone can benefit from.

Up to now, you've mostly written projects from scratch. This is a little like grinding your own grain: it's a great learning experience, but you'll quickly find that you're writing the same code over and over to accomplish common tasks like authentication, file parsing, or accepting user input. Thankfully, we've got a better way: *packages*.

A *package*is a collection of files & configuration wrapped up in an easy-to-distribute wrapper. By using packages, we can rely on the work of other developers to help move our own projects along at a rapid pace. Even better, we can create our own packages to share our code with the world!

Applications you write may be dependent on packages to work. We refer to these packages as *dependencies*of your code. Depending on the size of your project, you may have hundreds or even thousands of dependencies! In addition, a package may have its own dependencies on other packages. We'll discuss dependency management in an upcoming lesson.

Special delivery!

The oldest and most basic way of sharing code is good old "copy & paste". One developer could write a file they're proud of and share it directly with another person on their team. This is fast and simple, but unsustainable for quite a few reasons:

- Each time the file changes, the author would need to re-share the file.
- The author would need to keep multiple versions of the same file, just in case an old project breaks unexpectedly.
- Other developers might want to improve the file, but how? Now there are lots of different files that no one's keeping track of.

You can probably think of other reasons, too. It's like mailing a gift: you wouldn't throw your unwrapped gift in a mailbox and hope it ends up in the right place! You package your gift up and add important information the postal service can use to manage it appropriately.

Package managers

Software packages work in a similar way. *Package managers*are applications that accept your code, bundled up with some important metadata, and provide services like *versioning*, *change-management*, and even tracking how many projects are using your code. This would be a ton of work for one person to handle by themselves! Package managers have been used for decades to manage server software, but are relatively new to web development.

When we talk about a package manager, we may be referring to a few different things. Most package managers consist of at least two parts: a *command line interface (CLI)*and a *registry*. The CLI is an application you run locally, and lets you download and install/uninstall packages as needed. The registry is a database of package information, tracking which are available at any time.

These parts work together to make your experience smoother. Without a CLI, you'd have to manually download and configure each dependency of your app. Without a registry, you'd have to remember exactly where each package is stored to download it. Yikes!

Package managers may include lots of other functionality, like *bundling*,*build pipelines*, and *dependency management*. At their core, though, the CLI & registry are their primary parts. Without these, they're likely to fall into the broader category of *build tools*, which we'll introduce more about later on.

Package management for JavaScript

Like all languages, JavaScript went through a long period of unmanaged sharing. Since early JavaScript was used exclusively for the browser runtime, embedded`<script>`tags were the preferred way to share code. However, Node.js changed the game! Backend developers working with JavaScript brought common patterns from their own backgrounds, including package management.

Node.js was released in 2009. In early 2010, *npm* was released and included in Node.js. *npm*, the "Node Package Manager", was designed to give Node.js engineers a similar experience to backend development in other languages. It wasinspired by `yinst`, a package manager used at "Yahoo!", where npm's creator had worked previously.

npm took off quickly as the *de facto*standard for Node.js packages. However, the JavaScript development world was still fragmented! Some frontend developers working in the browser runtime created their own package managers for frontend-oriented packages (Bowerwas one popular manager still in use today). Ultimately, the confusion of dealing with multiple package managers for the same programming language grew too great, and frontend developers started adding their packages to npm. Today, npm is the most widely-used package manager for all JavaScript packages, regardless of whether they're backend dependencies, frontend dependencies, or command-line tools.

An aside on "npm": If you're attentive to grammar, seeing the name of this package manager written in all lowercase letters may be infuriating! It's a hot topiconline, too. We'll stick with formatting used by npm itself, but you may see it capitalized elsewhere. Just remember: we're all referring to the same tool!

Getting started with npm

Here's a great thing about npm: since it's part of Node.js, you don't have to install it separately. Once you've added Node.js to your system, you've got npm for free. Nice!

We'll walk through setting up & using the `npm`CLI tool soon, but if you'd like to experiment on your own, here are a few basic terminal commands to get you started:

- `npm`will show npm's help info, including some common commands and how to access more detailed guides.

- `npm init`will set your current project directory up for npm. This requires answering a few questions to generate a `package.json`file, a critical part of npm's dependency management functionality.

- `npm install`will download and install a package into your project. You can use the `-g`(or `--global`) flag to install a package for use everywhere on your system. To have some fun, run `npm install -g cowsay`. Once the download's completed, try running `cowsay Hello, world!`.

Just the basics

Here's a (very short) overview of how npm works its magic. Let's imagine we're installing a package called `pack-overflow`:

- You request the package with `npm install pack-overflow`.
- The `npm`CLI tool updates your `package.json`file to include `pack-overflow`as a dependency and requests the package from the npm registry.
- `npm`downloads the package and installs it to the `node_modules`folder in your current directory (one will be created if it's not already there). It chooses the most recent version by default.
- `npm`creates a `package-lock.json`file that includes where the installed package is located and exactly which version was used.
- You're all set! You can now `require('pack-overflow');`in your project.

We'll walk through this process in more detail later in this lesson.

What we've learned

Package management is a massive topic that we're just scratching the surface of. You'll get lots of practice using packages and npm as we get into more complex projects. For now, after reading this lesson, you should be comfortable with:

- discussing the role of packages & package managers;
- and explaining what npm stands for.

Next up, we'll learn more about what npm does can do for us.

Return To Sender: Understanding Dependency Management With npm

Now that you've seen npm in action, let's dig into the details. How can we read npm's file changes ourselves?

We'll cover:

- dependency management;
- semantic versioning;
- and how npm implements these features;

Dependency management

To understand dependency management, let's revisit our kitchen. Making a sandwich depends on us having bread. The type of sandwich depends on a certain type of bread: a hamburger might call for a sesame seed bun, while a falafel wrap uses pita. In each of these cases, we'd consider the bread a*dependency*of our sandwich.

Of course, the *dependency chain*goes further than our breadbox. The baker who made our bread has dependencies as well. Baking a gluten-free loaf? They might need almond flour. Specialty breads might require a unique oven or technique. Even though we don't see this process in our own kitchen, we're dependent on it too! If the baker can't make the correct bread, we can't create the sandwich of our dreams.

Software has a similar problem with dependencies. If my application depends on an authentication library that itself depends on an insecure password encryption package, then my application is now inherently insecure. Oh no!

Keeping all the possible dependencies of an application straight ourselves would be nearly impossible. Package managers to the rescue!

Get the right package every time

Many package managers, including npm, have the ability to *resolve*correct dependency versions. This means the manager can compare all the packages used by an application and determine which versions are most compatible. This ability makes dependencies much safer: there's less worry that an update will break your app if your package manager is warning you of changes.

npm accomplishes this dependency resolution process using both the`package.json`and `package-lock.json`files. Let's take a look at how this works.

Ask...

The `package.json`file contains lots of JSON-formatted metadata for your project, including its `dependencies`. Each dependency is formatted like so:

```
"package-name": "semantic.version.number"
```

The package name tells npm which package to search for, and the *semantic version number*lets the CLI know more about exactly which version of that package to grab. npm compares the version number with all your own dependencies to resolve the correct version.

You should consider your `package.json`'s dependencies to be a list of requests. Adding a dependency here lets you say "I'd like at least version 1.0 of the 'vue' package, please". It sets the stage for dependency resolution.

...and you shall receive!

The actual record of packages being used by an application is in`package-lock.json`. This file, commonly known as a *lockfile*in package manager parlance, contains every detail needed to identify the exact version of an npm package that's being used by an application. The lockfile is the key; without it, you can't say with any certainty whether a particular version was installed or not.

The lockfile for npm will be updated whenever an update is made to`package.json`and `npm install`is run. You can do this manually (for example, when you'd like to try a particular version of a package), or you can run `npm update <package-name>`. While you'll frequently make manual changes to your`package.json`, you should never make manual changes to your`package-lock.json`! Let npm be responsible for generating the lockfile.

Little boxes, all the same.

When the `npm`CLI utility installs a package, it adds it to the `node_modules`subdirectory in your project. Each package will be placed in a directory named after itself, and contain the raw code for the package along with any associated`package.json`s and documentation.

The `node_modules`folder is special for a couple reasons. For one, it's a great way to keep dependencies separated for each project. Some package managers keep dependencies in a central location on your computer. While npm can do this when run with the `--global`flag, it's not ideal, as it makes it harder to keep different versions of the same dependency. By keeping `node_modules`for each project separate, you can have as many different versions of each package as you like! Each project has the specific version it needs right on-hand.

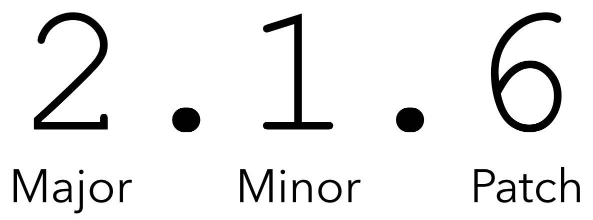
This introduces a challenge, though. If you have **every**version of a package, imagine how much space that might take up! Your `node_modules`folders, especially on larger apps, may grow to a massive size. There are build tools you will encounter that minimize storage space being used by dependencies, but in general it's good practice to keep `node_modules`out of git repositories or other version control. After all, future users can use your `package.json`along with `npm install`to recreate their own `node_modules`directory!

Saying a lot with three little numbers

Let's talk about that `semantic.versioning.number`above. *Semantic versioning*(often abbreviated to *semver*) is a way of tracking version numbers that lets other developers know what to expect from each release of your package.

Semantic version numbers are made up of three parts, each numbered sequentially and with no limit on how large they can be. The leftmost digit in semver is most significant, meaning that `1.0.0`is "larger" than `0.8.99`, though both are valid.

Here's a high-level overview:



- *Major*changes should be considered *breaking*. They will be incompatible with other major versions of the same package and may require significant changes in any app that depends on them. Creating a sequel to a hit video game would be a major change.

- *Minor*changes generally represent new features. These shouldn't totally break anything, but might require a little tweak to keep dependent apps up-to-date. Adding a new level to a video game would be a minor change.
- *Patch*-level changes are for fixing bugs or small issues. These shouldn't break any other functionality or force dependent apps to make any changes themselves. Fixing a typo in a video game's instructions would be a patch-level change.

Notice that each of these versions is most-concerned with compatibility. This is semver's greatest strength! With it, we can compare two versions of a package and know immediately whether they are compatible or not, even if we don't know much about how the code changed between those two versions.

Creating version ranges

Of course, part of the reason we're using a package manager is that we may not know exactly which version we need. Don't worry, though - semver & npm have you covered! When adding a new dependency to your `package.json`file, you can designate a *range*by adding some special characters to your version number:

- `*`indicates "whatever the latest version is".
- `>1.0.0`indicates "any version above major version 1".
- `^1.0.0`indicates "any version in the 1.x.x range".
- `~1.0.0`indicates "any patch version in the 1.0.x range".
- `1.0.0`indicates "exactly version 1.0.0".

You may also omit consecutive trailing zeroes, so `^1.0.0`is the same as `^1.0`or just `^1`. `~1.0.2`("any patch version greater than 1.0.2") would need to written out in its entirely, though. You should consider the numbers in your semver to be a minimum value, so `~2.1.3`would include `2.1.4`, but not`2.1.2`.

npm's website includes a fantastic semver calculatoryou can use to practice as you're learning this new syntax. Check it out!

Semantic versioning & npm

Semantic versioning is npm's secret weapon for dependency management. Using the rules of semver, npm is able to determine whether a package will be compatible with your application or not based on minimum-acceptable versions you set.

You might determine your minimums by trial and error, or you might just start building against the latest version and work hard to keep your code up to date as dependencies change. No matter how you do it, npm will make sure the packages you install fit within the version range you've set in your `package.json`.

Beware! While npm helps manage your dependencies, it won't automatically keep them up to date! Out-of-date dependencies may introduce serious security risks and require a substantial amount of work to fix. At the very least, you should ensure that any apps you maintain stay up-to-date with the latest patch versions of their dependencies. We'll look at some cool tools npm provides to assist with this during lecture.

What we've learned

Dependency management can be a lot of work! We're lucky to have package managers to help sort things out for us.

After reading this lesson, you should feel confident:

- defining "dependency management" and explaining why we need it;
- explaining npm's `package.json`and `package-lock.json`files;
- and evaluating semantic versions to find acceptable ranges.

Using npm to Perform Common Tasks - Part One

Now that you know what npm is and why it's useful, let's dig further into the details of how to use npm to perform common tasks.

We'll cover:

- verifying what version of npm is installed and how to use npm to update itself to the latest version;
- using npm to initialize a new package or project;
- using the npm registry to find a package;
- using npm to install a package;
- using an npm package in code;
- and understanding the difference between a dependency and a development dependency.

Using npm to manage npm

To confirm if you have the npm CLI installed, you can run the command `npm --version` or `npm -v`. If you have the npm CLI installed, you'll see its version number displayed in the console. If you don't have the npm CLI installed, you receive an error.

The npm CLI is available as an npm package, which allows you to use the npm CLI to update itself. If you're using macOS or Linux, you can update the npm CLI to the latest version by running the following command:

```
npm install -g npm@latest
```

If you installed Node.js using the default installer, you might need to prefix the above command with `sudo` like this:

```
sudo npm install -g npm@latest
```

The `sudo` command allows you to run a command with the security privileges of another user, typically your computer's administrator or superuser account. When using the `sudo` command, you'll be prompted for your account's password.

If you're using Windows, upgrading npm is a little trickier. See this page in the official npm docs for detailed information. Note: If you are using WSL (Windows Services for Linux) on Windows, then you will really be using Linux, and therefore the Windows specific instructions won't apply. Just use the normal methods.

Using npm to manage a project's dependencies

Now let's walk through the steps to initialize a project to use npm. Then you'll use npm to install and use a dependency in code.

Initializing a project to use npm

Any Node.js project that contains a `package.json` file is technically an npm package, though most of these projects will never be published to the npm registry for consumption by the general development community. Given that, it's common to refer to these unpublished npm packages as just "projects".

There's no better way to learn npm than to work through some examples, so open a terminal window and follow along!

If you haven't already, create a folder for your project, then use the `cd` command to browse to that folder. From within your project folder, run the following command:

```
npm init
```

npm will prompt you to supply the following field values, one at a time:

`package name`(or simply `name`) - If you're going to publish your package, setting your package name to something useful is very important. For typical development projects, it's okay to just accept the default value, which will be the name of the current folder.

`version`- If you're using semantic versioning for your package or project, then specify an appropriate starting point (i.e. `0.0.1`), otherwise just accept the default of `1.0.0`.

`description`- A description is really only necessary if you're going to publish your package, as it's displayed to users when they're searching the npm registry.

`entry point`(or `main`) - The file to use as the entry point to your application (typically `index.js`or `app.js`).

`test command`- If you're going to write tests for your package, you can provide the command to run those tests. For now, just press enter without providing a value to accept the default value.

`git repository`- If you want other developers to be able to find the Git repository for your package, you can provide the URL to the repo here. For now, it's okay to skip it by pressing enter.

`keywords`- Keywords are used to help people find your package in the npm registry. For now, just leave this field blank.

`author`- If you're the author of the package and you want your name (and contact information) associated with the package, you can provide that information here. For now, let's just leave this field blank.

`license`- This is the license for your package. It's only important to provide if you're going to publish your package. This defaults to the ISC License, which for our purposes, will work just fine (since we're not going to publish our package).

At this point in the process, npm will display a preview of the `package.json`file and confirm if you want to continue:

```
{
  "name": "introduction-to-npm",
  "version": "1.0.0",
  "description": "A simple project to explore using npm",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Go ahead and answer with "y" or "yes". You'll now have a `package.json`file in the root of your project. If you want, you can open the `package.json`file in a code editor and make additional edits to it.

Pro tip:*If you're like the majority of developers, you'll get tired of stepping through the above prompts time and time again when initializing a project to use npm. To save valuable time, you can pass the `--y`flag to the `npm init`command to generate a `package.json`file with all of the default values like this: `npm init --y`.*

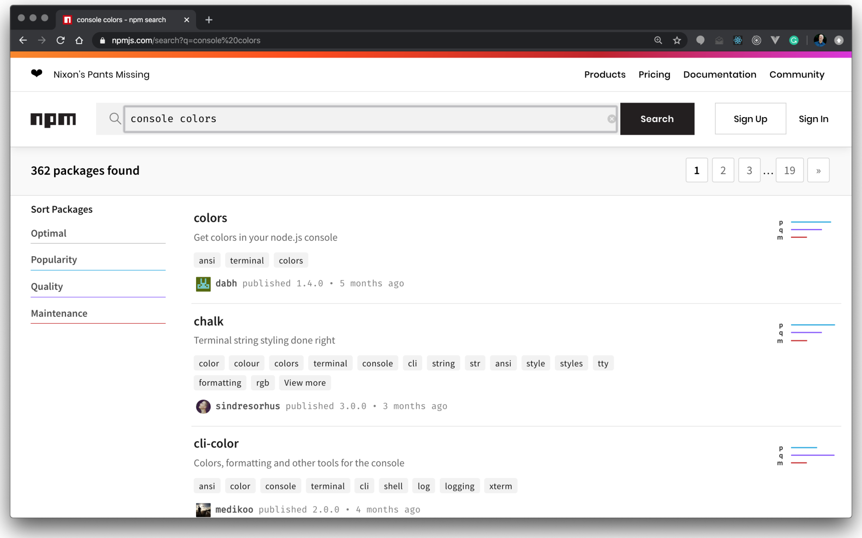
Finding packages in the npm registry

With more than a million packages in the npm registry, there's literally a whole world's worth of code for you to explore and to incorporate into your projects and applications.

When selecting a package, it's helpful to ask yourself the following questions:

- **Does the package do what I need?**Most packages in the npm registry will include some documentation on how to use the package. Usually you can review that documentation to determine if the package will suit your needs. Sometimes, you might need to review any additional documentation that's available in the package's code repository (i.e. GitHub, GitLab, or wherever the package's source code lives). Alternatively, you can install the package into a throwaway project to safely test it in a sandbox environment.
- **How popular is the package?**Popularity isn't always everything, but it can be an effective way to determine if a package is useful and reliable. It also increases the likelihood that other developers on your team might have experience with using a particular package.
- **Is the package being maintained?**If you're going to take a dependency on a package, you need to have confidence that the package is actively being maintained by its developer(s). To do that, review the package's associated code repository (i.e. GitHub, GitLab, etc.) Have there been recent commits and recent releases? Review the repository's issues to see if consumers of the package are getting their questions answered and bugs are being fixed.

Let's use the npm registry to search for a package that'll allow you to add color to the messages logged to the console. Open the npm websiteinto a browser tab. At the top of the page, use the "search packages" field to search for the keywords "console colors".



362 packages found

1 2 3 … 19 »

Sort Packages
Optimal
Popularity
Quality
Maintenance

colors
Get colors in your node.js console
ansi terminal colors
dabh published 1.4.0 • 5 months ago

chalk
Terminal string styling done right
color colour colors terminal console cli string str ansi style styles tty formatting rgb View more
sindresorhus published 3.0.0 • 3 months ago

cli-color
Colors, formatting and other tools for the console
ansi color console terminal cli shell log logging xterm
medikoo published 2.0.0 • 4 months ago

Package search results, by default, will be ordered by what npm refers to as "Optimal" sorting, which combines popularity, quality, and maintenance into a single score "in a meaningful way" (see the npm documentationfor more information). You can manually change the sorting by clicking on the "Popularity", "Quality", or "Maintenance" links on the left side of the page.

Go ahead and take a moment to review the pages for the `colors`and `chalk`packages, which at the time of this writing, the top two results.

Both the `colors`and `chalk`packages are quite popular and both look like they're being maintained. When you have more than one valid option, it really comes down to a matter of preference on how the package approaches the problem that it solves. In this particular situation, let's select the `colors`package as it uses an interesting approach to how colors are applied to console messages.

It's also possible to search the npm registry from the terminal using the `search` command. For more information about that command, see this page in the npm documentation.

Installing a dependency

Now that you've initialized your project using `npm init` (which generated a `package.json` file), you can use `npm install` to install an npm package locally into your project.

To install the `colors` npm package, run the following command:

```
npm install colors
```

The installation process generates the following output:

```
npm WARN introduction-to-npm@1.0.0 No repository field.

+ colors@1.4.0
added 1 package from 2 contributors and audited 1 package in 0.346s
found 0 vulnerabilities
```

When installing a package, npm will validate your project's `package.json` file and report back to you about any issues that it found. In the above output, you can see that the `repository` field is currently blank. That's okay though; as mentioned earlier, if you're not going to publish your package, it's okay to leave the `repository` field blank.

In the output, you can see the version of the `colors` package, `1.4.0`, that npm downloaded and installed into the `node_modules` folder. npm also informs you that it didn't find any vulnerabilities in the `colors` package.

Later in this lesson, you'll learn more about security vulnerabilities in npm packages and how you can discover and resolve them.

Here's what the `dependencies` field in the `package.json` file looks like after installing the `colors` package:

```
{
  "dependencies": {
    "colors": "^1.4.0"
  }
}
```

Git and the `node_modules` folder

When using `npm install` to install an npm package locally into your project, npm downloads and installs the specified package to the `node_modules` folder. If the installed package has its own dependencies (npm packages often depend upon other npm packages), npm will automatically download those dependencies into the `node_modules` folder. This process recursively continues until all of the required dependencies are accounted for. Because of this, the `node_modules` folder tends to be very large, containing many folders and files.

If you're using Git for source control, you'll need to add a `.gitignore` file to the root of your project and add the entry `node_modules/` so that the `node_modules` folder won't be tracked by Git. Later in this lesson you'll see that you only need to commit the `package.json` and `package-lock.json` files to your repository as that's all that npm needs to download and install your project's dependencies.

Pro Tip: *While configuring Git to not track the `node_modules` folder is important to do, it's not necessarily the only thing you want to configure Git* *not to track. For a more comprehensive `.gitignore` file for Node.js projects, you can use GitHub's `.gitignore` file for Node.js projects.*

Using a dependency in code

After installing an npm package, you can import it into a Node.js module using the `require` function.

Go ahead and add a file named `index.js` to your project. Then use the `require` function to import the `colors` module:

```
const colors = require('colors');
```

After importing the module, you can use it to add color to your `console.log()` method calls like this:

```
console.log('hello'.green); // outputs green text
console.log('i like cake and pies'.underline.red) // outputs red underlined text
console.log('inverse the color'.inverse); // inverses the color
console.log('OMG Rainbows!'.rainbow); // rainbow
console.log('Run the trap'.trap); // Drops the bass
```

Now you can use Node.js to run and test your code:

```
node index.js
```

You should see five lines of text displayed in the console formatted in various ways using the `colors` package.

The above code snippet was taken directly from the documentation for the `colors` npm package, which can be found here on the npm registry.

Dependency types

npm tracks two types of dependencies in the `package.json` file:

- Dependencies (`dependencies`) - These are the packages that your project needs in order to successfully run when in production (i.e. your application has been deployed or published to a server that can be accessed by your users).
- Development dependencies (`devDependencies`) - These are the packages that are needed locally when doing development work on the project. Development dependencies often include one or more tools that are used to build and test your application.

There are actually three additional types dependencies that npm can track including peer dependencies (`peerDependencies`), bundled dependencies (`bundledDependencies`), and optional dependencies (`optionalDependencies`). These dependency types are less often used, so we won't be covering them here. For more information about these dependency types, see the npm documentation.

Installing a development dependency

To install a development dependency, you simply add the `--save-dev` flag:

```
npm install mocha --save-dev
```

The `--save-dev` flag causes npm will add the package to the `devDependencies` field in the `package.json` file:

```
{
  "dependencies": {
    "colors": "^1.4.0"
  },
  "devDependencies": {
    "mocha": "^7.0.1"
  }
}
```

Separating the development dependencies from the application's "regular" dependencies keeps the package installation process as lean as possible, by allowing npm to install just the packages that are actually needed for the package or application to successfully run.

What we've learned

After reading this lesson, you should be comfortable with:

- verifying what version of npm is installed and how to use npm to update itself to the latest version;
- using npm to initialize a new package or project;
- using the npm registry to find a package;
- using npm to install a package;
- using an npm package in code;
- and understanding the difference between a dependency and a development dependency.

Using npm to Perform Common Tasks - Part Two

Ready to dig further into the details of how to use npm to perform common tasks? Let's get started!

We'll cover:

- installing an existing project's dependencies;
- using npm to uninstall a package;
- using npm to update a package;
- finding and fixing npm package security vulnerabilities;
- and writing and running npm scripts.

Installing an existing project's dependencies

When getting started with an existing project that already contains `package.json` and `package-lock.json` files, you'll need to use npm to install its dependencies. If you don't install a project's dependencies, you'll almost always receive errors when attempting to run the application.

To install an existing project's dependencies, simply run the `npm install` command without providing any package names. This causes npm to install the dependencies listed in the `package-lock.json` file.

During the package installation process, npm will display the overall status as it downloads and installs each package. When the installation process is completed, npm will display information in the terminal that summarizes what was done:

```
added 108 packages from 555 contributors and audited 206 packages in 1.834s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

In the output, you can see that 108 packages were added to the project. npm also gives you a friendly heads up that 14 of those packages are looking for funding! You can also see that npm didn't find any vulnerabilities in any of the packages.

Older versions of npm didn't utilize `package-lock.json` files, so sometimes you'll work with existing projects that'll only have a `package.json` file. If there's just a `package.json` file, then running the `npm install` command will install the dependencies listed in that file.

Uninstalling a dependency

Sometimes you'll install a dependency only to later on realize you actually don't need it. When this happens, you can use npm to remove the dependency from your project.

Imagine that you install the `lodash` package by running the following command:

```
npm install lodash
```

Here's the output displayed by npm after the package is installed:

```
+ lodash@4.17.15
added 1 package from 2 contributors and audited 1 package in 0.609s
found 0 vulnerabilities
```

At this point, your `node_modules`folder will contain a folder named `lodash`that contains the code for the `lodash`package. Your `package.json`file will also list `lodash`as a dependency:

```
{
  "dependencies": {
    "lodash": "^4.17.15"
  }
}
```

`lodash`is an amazing library, but sometimes application requirements change and you no longer need the libraries that you thought you needed. After reviewing your application, you realize that `lodash`isn't being used, so you decide to remove the package from your project.

To remove or uninstall the `lodash`package, you can run the following command:

```
npm uninstall lodash
```

Here's the output displayed by npm after the package is uninstalled:

```
removed 1 package in 0.439s
found 0 vulnerabilities
```

When npm uninstalls a package, it'll remove the package and all of its dependencies from the `node_modules`folder. npm will also update the`package.json`file (and the `package-lock.json`file) so the package is no longer listed as a dependency:

```
{
  "dependencies": {}
}
```

Updating a dependency

Being able to leverage code that you didn't write yourself can be a huge timesaver. And while you don't have to directly maintain code contained within a package that you've taken as a dependency, you might find yourself needing to maintain the dependency itself.

For example, a package might contain a bug or the developer of the package might add a new feature that you now want to use in your application. When this happens, you can use npm to update the package.

Imagine that you added the `lodash`package as a dependency to a project awhile ago; back when the latest version of `lodash`was `3.0.0`. You can simulate this situation by installing a specific version of `lodash`:

```
npm install lodash@3.0.0
```

Which results in the following output:

```
+ lodash@3.0.0
added 1 package from 5 contributors and audited 1 package in 0.606s
found 3 vulnerabilities (1 low, 2 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

For now, ignore the security vulnerabilities that npm found. You'll learn in a bit how to use `npm audit`to resolve those issues.

Here's what the dependency in the `package.json`file looks like at this point:

```
{
  "dependencies": {
    "lodash": "^3.0.0"
  }
}
```

Remember, that the semver `^3.0.0`means that you'll accept any minor and patch versions for `lodash`major version `3`.

Now imagine that you want to update `lodash`, so you can use some cool features that were added in version `3.1.0`. You can update the `lodash`package by running the following command:

```
npm update lodash
```

Which results in the following output:

```
+ lodash@3.10.1
updated 1 package and audited 1 package in 0.404s
found 3 vulnerabilities (1 low, 2 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

And here's what your dependencies look like in the `package.json`file:

```
{
  "dependencies": {
    "lodash": "^3.10.1"
  }
}
```

Updating all project dependencies

Instead of updating your project's dependencies one by one, you can update all of your dependencies with a single command:

```
npm update
```

This updates all your project's dependencies while respecting each dependency's semver (as stated in the `package.json`file).

Re-installing a dependency with updated semver information

If you want or need to update a dependency to a version that's greater than what's allowed by that dependency's semver, you can use the `npm install`command to re-install the dependency with updated semver information.

If you currently have `lodash`installed as a dependency with the semver`^3.10.1`, you can re-install `lodash`to update the dependency to `4.0.0`:

```
npm install lodash@4.0.0
```

Here's the resulting output:

```
+ lodash@4.0.0
updated 1 package and audited 1 package in 0.605s
found 3 vulnerabilities (1 low, 2 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

And the updated `package.json`file:

```
{
  "dependencies": {
    "lodash": "^4.0.0"
  }
}
```

Pro tip:*You can easily update a package to the latest version by specifying `latest`for the version. For example, to update the `lodash`package* *to the latest version, run the command `npm install lodash@latest`. At the time of this writing, this results in the `lodash`package being added as a dependency with a semver of `^4.17.15`.*

Finding and fixing package security vulnerabilities

Bugs are an unavoidable part of writing code and sometimes those bugs result in security vulnerabilities. npm packages aren't immune to either of these realities.

Luckily, npm recognizes this and gives us a way to find and fix package security vulnerabilities. npm maintains a list of reported package security vulnerabilities that's primarily supported by developers in the JavaScript community.

Through the npm registry, you can report any security vulnerabilities that you've discovered. See this page in the npm documentationfor more information.

When you install a package, npm checks their list of reported security vulnerabilities to see if the package(s) being installed have any known issues. If any issues are found, npm will display a warning.

We can see this in action by installing an older version of the `lodash`package:

```
npm install lodash@3.0.0
```

Which results in the following output:

```
+ lodash@3.0.0
added 1 package from 5 contributors and audited 1 package in 0.314s
found 3 vulnerabilities (1 low, 2 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

In the above output, you can see that three vulnerabilities were found. To see additional details about the vulnerabilities, you can run the following command:

```
npm audit
```

Which displays the following information:

```
                       === npm audit security report ===

# Run  npm install lodash@4.17.15  to resolve 3 vulnerabilities
SEMVER WARNING: Recommended action is a potentially breaking change
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Low           │ Prototype Pollution                                          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/577                             │
└───────────────┴──────────────────────────────────────────────────────────────┘


┌───────────────┬──────────────────────────────────────────────────────────────┐
│ High          │ Prototype Pollution                                          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
```

```
│ Path          │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/782                             │
└───────────────┴──────────────────────────────────────────────────────────────┘


┌───────────────┬──────────────────────────────────────────────────────────────┐
│ High          │ Prototype Pollution                                          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ lodash                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/1065                            │
└───────────────┴──────────────────────────────────────────────────────────────┘


found 3 vulnerabilities (1 low, 2 high) in 1 scanned package
  3 vulnerabilities require semver-major dependency updates.
```

***Pro Tip:** When working with existing projects, you can use the `npm audit` command to audit the project's dependencies for any reported security vulnerabilities.*

In the audit report, you can use the `severity` field to determine how important it is to address the issue. npm defines four security levels:

- Critical - Address immediately
- High - Address as quickly as possible
- Moderate - Address as time allows
- Low - Address at your discretion

The `lodash@3.0.0` package contains one "low" vulnerability and two "high" vulnerabilities. You can use the following command to attempt to fix any security vulnerabilities:

```
npm audit fix
```

Which displays the following output:

```
fixed 0 of 3 vulnerabilities in 1 scanned package
  1 package update for 3 vulnerabilities involved breaking changes
  (use `npm audit fix --force` to install breaking changes; or refer to `npm audi
```

Unfortunately, as you can see in the above output, the `npm audit fix` command will only work if a fix is available in a minor or patch version of the package. When a fix requires updating to a new major version of a package, that's considered by npm to be a "breaking change".

If you need to move to a newer major version of a package to resolve a security vulnerability, you can pass the `--force` flag:

```
npm audit fix --force
```

And here's the resulting output:

```
npm WARN using --force I sure hope you know what you are doing.

+ lodash@4.17.15
updated 1 package in 0.463s
fixed 3 of 3 vulnerabilities in 1 scanned package
  1 package update for 3 vulnerabilities involved breaking changes
  (installed due to `--force` option)
```

Upgrading to a newer major version might break the code in your application! If you need to do this to resolve a security vulnerability, you'll need to test your application to ensure it still works correctly after updating the dependency.

If a fix isn't available for a security vulnerability, you'll need to do additional research on the reported issue to determine the best course of action for your project.

GitHub also keeps track of known security vulnerabilities in npm packages and will alert you when one of your repositories contains a dependency with a known security vulnerability. See this page in GitHub's documentationfor more information.

Writing and running npm scripts

In addition to helping you manage your project dependencies, npm also gives you a convenient way to define and run scripts that execute one or more commands that you'd normally run from the terminal.

npm scripts are defined using the `scripts`field in the `package.json`file:

```
{
  "scripts": {
    "start": "node index.js"
  }
}
```

Once you've defined the `start`script, you can run it from a terminal like this:

```
npm start
```

As the name suggests, the `start`script is used to define the command (or commands) to start your application. The `start`script is just one of the available predefined script names.

For a list of the available script types, see this page in the npm documentation.

It's also common to use a script to run tools that are installed using npm. For example, if you were writing unit tests for your project, you might install the`mocha`npm package and write a `test`script to execute `mocha`like this:

```
{
  "scripts": {
    "start": "node index.js",
    "test": "mocha --watch"
  }
}
```

And to run the `test`script, you run the following command:

```
npm test
```

Defining custom scripts

You can also define custom scripts. Imagine that you installed the `nodemon`package, a file watcher that will restart your application when changes are made to project files.

Defining a custom script for starting your application using `nodemon`allows you leave the `start`just as it is. That way you've got the flexibility to start your application with or without file watching.

To define a custom script, simply define a script with a name that's not in the list of predefined npm scripts:

```
{
  "scripts": {
    "start": "node index.js",
    "test": "mocha --watch",
    "watch": "nodemon index.js"
  }
}
```

To run the `watch`script, you'd run it like this from the terminal:

```
npm run watch
```

`nodemon`will start your application and begin watching your project files for changes. If you make a change to the `index.js`file, `nodemon`will stop and restart the application so that you can see the result of your code change.

What we've learned

After reading this lesson, you should be comfortable with:

- installing an existing project's dependencies;
- using npm to uninstall a package;
- using npm to update a package;
- finding and fixing npm package security vulnerabilities;
- and writing and running npm scripts.