Tip 48: Leverage Community Knowledge with npm

In this tip, you'll learn how to import external code with npm.

We'll cover the following

- Access external code using npm
- Set up
- Using npm & loadash for installation
- Import loadash
- npm & development tools/dependencies
- npm scripts

In the previous tip, you learned how to use code from different files. In this tip, you'll learn how to use community code created by developers all over the world.

Access external code using npm

Not many years ago, if you wanted to use an open-source library, you were forced to either copy-paste code locally, download a library to your project, or include an external dependency using a `<script>` tag in your markup.

You'd get your code—unless an external source went down—but it was hard to keep dependencies up to date, particularly if you were storing them locally. Not only was it hard to maintain, but you also had to write your custom code assuming the library would be there. This made code really hard to read and test because you never explicitly included anything.

Those days are gone. You can now download code directly to your project, control versions, and import code into individual files using familiar conventions.

You manage all this with a tool called the **node package manager**, or **npm**. There are a few alternatives, such as Facebook's *yarn* project, but they work in mostly the same way, so don't worry too much about the differences.

npm is an important project, and you'll mostly use it for *importing* code, but it can

do a lot more. With npm, you can set your project's *metadata* and *configuration*

information, run *command-line* scripts, and even publish your project for others to use.

Set up

> The setup instructions given below have already been done for you on the platform.

To run the code locally, you'll need to have Node.js installed. After that, you're ready to go. When you install Node.js, you also install npm.

After Node.js and npm are installed, you need to initialize a project. Open up a terminal, go to the root of your project, and type `npm init`. This will start up a configuration tool that will create a `package.json` file for you.

This `package.json` file will contain metadata information for your project, such as *name, description, license*, and so on. It will eventually contain all the external code dependencies. It's important to note that `npm init` only creates the `package.json` file. It doesn't set any other hidden files or directories. You don't need to worry about cluttering your file system.

If you aren't sure what you want, don't worry. You can change all of that information later. Go with the defaults at first if you have any doubts. When you finish, you'll have a file that looks like this:

```
{
    "name": "test",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC"
}
```

Pretty simple. It's just an object containing most of the information you just entered. The only surprise is the scripts field. This is where you'd add *commandline* scripts. You'll see some more of this in a bit.

Don’t let the slimness of the file fool you. This is the most important entry point for a large JavaScript application. It’s also where you store information about external dependencies.

Using npm & loadash for installation

Let’s say you’re building a collection with maps and you wanted to convert an object to a *map*. You could write some code to convert the object to a map, but you really just want a quick solution. With some research, you’d likely come across Lodash—*a suite of tools for converting data*. *How do you get the code into your project?*

In addition to structuring your project, *npm* is also a resource for sharing code. Better still, npm tracks data such as number of downloads, number of open bugs, versions, and so on.

If you open the *npm* page for the Lodash package, you’ll see that it’s been downloaded about 50 million times per month. In essence, 50 million projects are giving Lodash a stamp of approval.

The data collected by *npm* is a tacit endorsement by the greater JavaScript community. You don’t need to avoid code from rarely used projects, but you may want to investigate the code before bringing it into your codebase. Fortunately, there’s always a link to the code base, so you can check it out yourself if you aren’t sure.

 **Evaluating Open Source Code**

Once you’re satisfied that code is worth using, you can install it in your project by running `npm install --save lodash`. It’s not strictly necessary to use the `--save` flag, but it’s a good habit because there are two types of code you’ll install. More on that in a bit.

The `npm install` command does a few things. If there isn’t a `node_modules` directory, the command will create one and copy down the package. Next, it updates your `package.json` to include the version number of the code you’re

importing. Finally, it will create a `package-lock.json` file that includes detailed

information about the version of the installed code along with any other libraries that the code requires.

When you install one package, you may actually install several packages. They'll all be in your `package-lock.json` or your `node_modules/` directory. The original code you installed, `lodash`, will be the only code to appear in your `package.json` file. This is so other developers can see what top-level projects you need without getting stuck in the details of dependencies.

Here's your updated `package.json` file. Note that it now has a dependencies field.

```
{
    "name": "test",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
        "lodash": "^4.17.4"
    }
}
```

Import loadash

Now that you have your code, it's time to use it. Importing the code is simple. You use the same `import` command from the previous tip, but because you're installing a library, you don't need to give a path.

Here's how you'd import `lodash`. You can either import an individual function, such as `fromPairs()`, or you can import the *default* object. The default `lodash` import actually contains `fromPairs()`, but some libraries split things out.

```
import lodash, { fromPairs } from 'lodash';

export function mapToObject(map) {
    return fromPairs([...map]);
}

export function objectToMap(object) {
    const pairs = lodash.toPairs(object);
    return new Map(pairs);
}
```

Looks pretty familiar, huh? The nice part is that no matter where you import the code, you use the same syntax. And when you're reading the code, it's easy to see what functions are from outside the codebase. Anything that doesn't use a relative import must be an external code.

npm & development tools/dependencies

If npm only tracked dependencies, it would be a great project, but it does more. You'll often need code that does work on the codebase but isn't part of the production build.

For example, you'll want a test runner, but you don't need the test runner to be in your production code. npm will handle the development dependencies and give you clean interface for running them.

Say you wanted to add prettier to your project. Prettier is a tool for formatting your code to match style guides. It's a tool for development, not a production dependency.

Because you don't need it in production, you'd install it with the `npm install --savedev prettier` flag. Notice that you're using the `--save-dev` flag. This will also update your `package.json` file, but it will put the dependency under a different key.

```
{
    "name": "test",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
    "lodash": "^4.17.4"
    },
    "devDependencies": {
    "prettier": "^1.8.2"
    }
}
```

Of course, now you probably want to use it. Prettier is installed in your `node_modules` directory, which means you can't access it on the command line

directly. Let's say you wanted to make sure all tabs have a width of four spaces.

The documentation says you can convert code by running this command: `prettier --tab-width=4 --write ./code/*.js`.

The command won't work if you installed the code locally. If you installed the package `globally—npm install -g prettier`—then you'd be able to run the command, but then the package wouldn't live specifically in your project. You'd have to somehow communicate to other developers the project has a global dependency.

You can solve the problem using npm scripts.

npm scripts

With an npm script, you run the exact same command, but the script looks in the `node_modules` directory. To run the command, add it to the script object of your `package.json` file. Change the scripts object to include a key of clean with a value of `prettier --tab-width=4 --write./code/*.js`.

Now, when you're in the same directory as your `package.json` file, you can run `npm run clean` and npm will execute the command using the locally installed prettier package.

```
{
    "name": "test",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "clean": "prettier --tab-width=4 --write ./code/*.js"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
        "lodash": "^4.17.4"
    },
    "devDependencies": {
        "prettier": "^1.8.2"
    }
}
```

The code below shows how prettier will update the code to have a tab-width of four spaces.

Try it out. Clone the code for this course; then navigate to `architecture/npm/script`, and `run npm install` and then `npm run clean`, and

prettier will update the code to have a tab-width of four spaces.

Terminal

Not only is the dependency scoped to your local project, but other developers can see your build process, dependencies, and package information in a single file.

It's hard to overstate how valuable npm is for JavaScript development. If you ever start looking through a new project, you should begin by skimming the package.json file.

Now that you have the tools to combine multiple files and code from open source projects, it's time to think about how to organize your code.

In the next tip, you'll learn how to organize project assets in a single directory with component architecture.