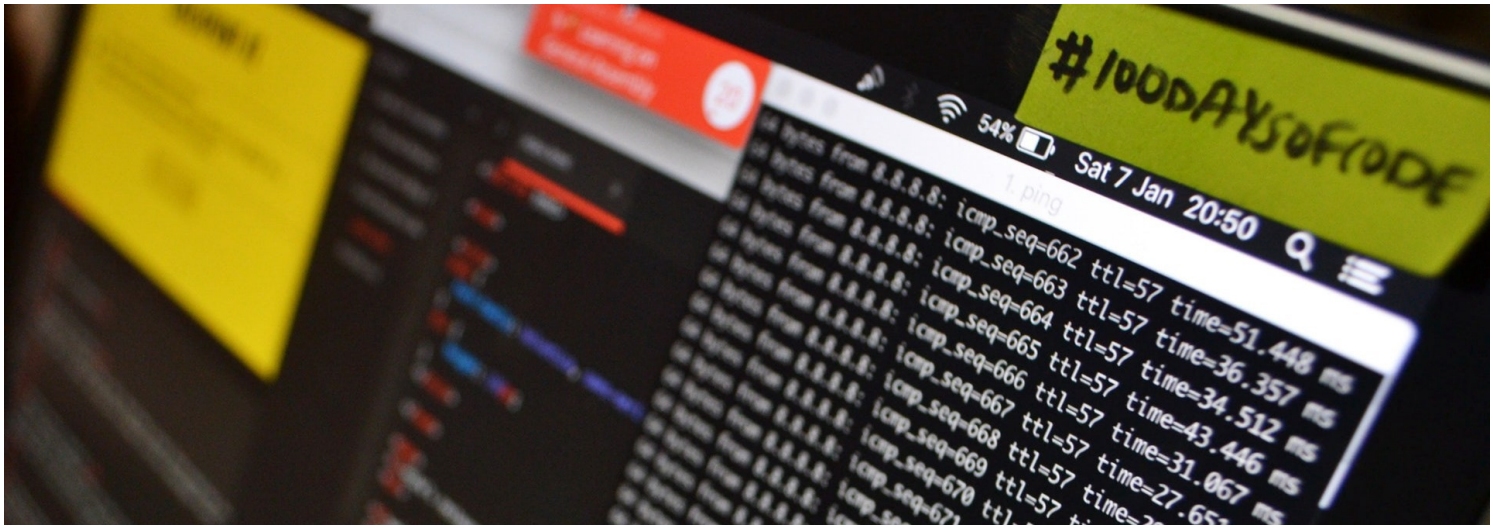
NODE.JS & NPM

Creating CLI Executable global npm module

Uday Hiwarale Follow
Mar 24, 2018 · 8 min read



JavaScript has quickly became the most popular language on the planet and it is growing like anything. After ES6 and ES7, JavaScript official joined club of Object Oriented Programming languages. With `async/await`, it has become very easy to write asynchronous programs. The bigger success of JavaScript came with launch of Node.js which pushed JavaScript on server stack. Hence JavaScript is now found everywhere, from **mobile applications**, **desktop applications**, **IoT devices**, **Servers** to every day **web browsers**.

But above all, what I like the most about JavaScript, Node.js and npm together is the ability to build CLI applications. Somewhere in your life you might have opened the terminal and executed some command like `git clone xxx` or `rm -rf xxx`, these are all CLI commands referring to some program like `git` and `rm`. We can also install a npm

module globally which means we can refer to that module from anywhere in our system using terminal commands.

When you install a module globally, npm will place that module inside a fixed folder (*let's call it* ***npm folder***), for example in Windows it could be `$user/AppData/Roaming/npm` or *any other folder depending on your system*. While installing a global module, npm processes `package.json` of that module and looks for `bin` field. **bin** field is an **object** with **key being the terminal command** and **value being the `.js` file in the module which needs to be executed when user executes that command**. If **bin** field is non-empty, then npm creates necessary files inside **npm folder** so that user can use the commands specified in **package.json**. These file generally do not have any extension and their file name is the same as command name that they will execute on. In Windows, `.cmd` extension file is also generated along with previous file to make sure execution of `.js` file with **Node.js** only. I will explain this bit later.

So let's create our first global module. First thing to remember is, a global npm module is just like local npm module with few extra information in `package.json` file. We are going to develop a CLI application where user can execute `greet` command and a random greeting prints to the terminal in any random language.

Ok, so first we need to create a sample `package.json` file to work with. Let's create a folder `greeting-project` and execute `npm init -y` command inside it. This will create a dummy `package.json` file to work with. Our folder structure should look like below.

```
greeting-project
|_ bin\
|_ lib\
|_ package.json
```

`bin` folder will contain all executable `.js` files, in our case there should be `index.js` (*name of the file can be anything*) file inside it because when user will execute `greet` command, that's the file we want to execute. `lib` folder contains other files which `index.js` might use. We can also install any dependency modules **which our CLI application may depend on**. In our application, we will use `lodash` and `colors` modules to help our module with some things which you will see next. These modules

must be installed locally, hence we will execute command `npm install --save lodash colors` and npm will install these modules for us.

Finally, our `package.json` will look like below.

```
{
    "name": "greeting-project",
    "version": "1.0.0",
    "dependencies": {
        "colors": "^1.2.1",
        "lodash": "^4.17.5"
    }
}
```

There can be many other fields as well but I removed unnecessary fields to make it look clean. Now, we have to add other fields manually to make this module global. But before that, let's write our program and test it locally. We are going to create few files like below.

```
greeting-project
|_ bin
  |_ index.js
|_ lib
  |_ greet.js
```

`greet.js` file contains all logic of our CLI application module. Content of this file looks like below.

```
const _ = require("lodash");

const GREETINGS = {
    en: "Good Morning",
    de: "Guten Morgen",
    fr: "Bonjour",
    ru: "Dobre Utra",
    kr: "Annyeonghaseyo"
};

// greet by the language code
```

```
exports.greet = function (code) {
    if (code) {
        // check if value associated with the language code exists
        if (!GREETINGS[code]) {
            return "Error! We don't support this language.";
        }
        else {
            return GREETINGS[code];
        }
    }
    else {
        // return greeting in english if code is empty
        return GREETINGS['en'];
    }
}

// greet a random greeting
exports.greetRandom = function () {
    // _.values returns values of objects in array
    // _.sample returns any random item in array
    return _.sample(_.values(GREETINGS));
}
```

greeting-project-greet.js hosted with ❤️ by GitHub view raw

(greet.js)

From above code, you can see that we have imported `lodash` module on top. Then we created a JavaScript object with **key being the language name of greeting and value being the greeting itself in that language**. After that, we exported two functions, `greet` and `greetRandom`. `greet` function returns the greeting in the language received by the function and `greetRandom` returns any random greeting.

`index.js` file is supposed to execute when user executes `greet` command in terminal. Initially, what we are going to do is, import `greet.js` and test the `greetRandom` function. It's content look like below.

```
#!/usr/bin/env node

const colors = require('colors');
const greet = require("../lib/greet");

```

```
// print random greeting
console.log(
    // wraps text with rainbow color formatting
    colors.rainbow(
        // returns the random greeting text
        greet.greetRandom()
    )
);
```

greeting-project-index.js hosted with ❤️ by GitHub view raw

(index.js)

If you notices the first line of code which is `#!/usr/bin/env node` which looks obviously suspicious is a **shebang** which tells operating system what interpreter or application to pass that file to for execution. In our case, it's `node` which is obvious from above shebang. But Windows unfortunately do not support shebang, instead it passes a file to the default interpreter or application associated with the extension of that file. Hence npm creates `.cmd` file inside **global npm folder** so that Windows will use `node` interpreter to execute `.js` files even default application associated with `.js` extension might be something else.

In `index.js` , we have imported `colors` module which will help us print colorful text to the terminal. We also imported `greet.js` file from `lib` folder which contains application logic. For test purpose, we will print random greeting to the terminal. This can be done by using `greetRandom` function from `greet.js` file. Rest of the code should be obvious to you.

Now let's test it locally before installing it globally to use it from CLI. To test this program, we will run `index.js` using node like `node ./bin/index.js` which will print this in rainbow color to the terminal.

```
Dobre Utra
```

Above response can be different in your case as we are printing a random greeting. Our application seems to be working. Now what we want is when we execute `greet` command in terminal instead of `node ./bin/index.js` command, same response should

be shown. That means we need to map `greet` command with `./bin/index.js` file. This is done by modifying `bin` field in `package.json` as we talked about earlier. The important thing to remember is **when we will execute `greet` command, that command will translate to `node ./bin/index.js` from the global node folder.**

There is one more boolean value field `preferGlobal` in `package.json` which if set to `true` prints warning to the console when user is installing this module locally. This doesn't prevent module to be installed locally, but this will certainly shed some light on confusion in case user notices. If a module can be used both locally and globally, then `preferGlobal` is set to false or rather does not added to the `package.json` as it's default value is `false`. In our case, we case we want user to use it both locally and globally, we will not add it in `package.json` to begin with. Hence, our final `package.json` will look like below.

```
{
    "name": "greeting-project",
    "version": "1.0.0",
    "main": "./lib/greet.js",
    "bin": {
        "greet": "./bin/index.js"
    },
    "dependencies": {
        "colors": "^1.2.1",
        "lodash": "^4.17.5"
    }
}
```

Look carefully at `bin` field. As we discussed that **key of this field is command**, in our case it is `greet` and **value is file to execute with that command which** is `./bin/index.js` in our case. There is one more field `main` in above `package.json` which tells node that when somebody is trying to import this module locally like `const greeter = require('greeting-project')`, then provide him/her `./lib/greet.js` file to implement business logic of our application. *If `main` is missing, then node by default will try to pull `index.js` file from module's root directory which is clearly missing in our case*. Adding `main` makes our module both locally and globally usable. We are not going to test local installation, though.

> *If you just have one command in your program and that command is same as npm package name of your project then you can use* ***path to bin file directly as value of*** `bin` ***field*** *like* `{"bin": "./bin/index.js"}` *which will be equivalent to* `{"bin":{"greeting-project": "./bin/index.js"}}` *. But avoid this, so that even package name changes, command won't change.*

Now let's install our module globally. Generally we have installed global modules like `npm install -g package_name` where **package_name** is published module available on npm's registry. Since we haven't published our module and we don't have any intention of publishing it until we done all our testing, we kinda have to trick `npm` to install from local source code. This can be done using `npm install -g local_dir_path` where **local_dir_path** is directory path of source code of the module we want npm to install globally. Since we are inside the folder of our module's source code, we can use `npm install -g ./` which will instruct npm to create symbolic link from **global npm folder** to the current folder. This will also create `greet` and `greet.cmd` files inside **global npm folder** as well.

Now are free to execute `greet` command from terminal. When I execute `greet` command in my terminal, I get random greetings. I hope it is working for you too.

Any changes made inside `index.js` **will reflect immediately because npm created only symbolic link**, hence greet command refers to the `index.js` file inside our local source code.

Now let's understand about command line arguments. When we execute any `.js` file using Node.js, node provides `process` variable which contains information about the executing process. `process.argv` returns the arguments used while executing a `.js` file with node. Let's add following line to at the end of our `index.js` and execute `greet --lang ru` command.

```
console.log(process.argv);
```

Execution of `greet --lang ru` command will print below response to the terminal.

```
Bonjour
[ 'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\Uday\\AppData\\Roaming\\npm\\node_modules\\greeting-
project\\bin\\index.js',
  '--lang',
  'ru' ]
```

Focus on part inside square brackets. These are the arguments of the process. First element is the path of node interpreter and second element is the path of file being executed. **Later elements are space separated text values added after** `greet` or `node ./bin/index.js` **command**. We can use this information to execute either `greet` or `greetRandom` function depending on user choice.

Let's modify `index.js` file to incorporate that logic.

(index.js)

Now, when we execute `greet --lang ru` command, only **Dobre Utra** is getting printed. But when we use `greet --lang` or `greet` then a random greeting is printed, because `lang` variable in `index.js` will be empty in those cases.

Looks like our app is working well. Now it's time to publish it on npm registry so that other people can use it. This is just like publishing local module with command `npm publish`. When other people will install our module with command `npm install -g greeting-project`, npm copies source code from it's registry to **global npm folder**, **creates necessary files for CLI execution** and **installs dependencies of our module**. Once npm done installing our module, users can execute `greet` command on their system with ease.

. . .

What happens when a user installs module locally?

Well, nothing bad happens. But then user has to define a command that was supposed to be used from terminal, inside `package.json`. `package.json` has `scripts` field which is

JSON `key-value` object where `key` is short-name of command that is defined as `value`. For example,

```
// packag.json

{
  "scripts": {
      "greet-ru": "greet --lang ru"
  }
}
```

Now, user can run this command from terminal using **npm** like

```
npm run greet-ru
```

Also, we have exposed business logic of our app through `package.json` in `main` field like below.

```
"main": "./lib/greet.js"
```

By importing `greeting-project` package, user can use this business logic however he/she wants in application.

. . .

I hope this tutorial was fun for you guys. You can find repo of above example on GitHub at https://github.com/thatisuday/npm-greet-global-module-example.

. . .

I have written more advanced tutorial on making CLI apps more powerful and interactive using **command.js** and **inquirer.js** which is available on Medium.

Making CLI app with ease using commander.js and Inquirer.js

In my previous post about Making CLI Application in Node.js (I am going to call it previous blog)which you can find it...

itnext.io

. . .

(Support Me on Patreon / GitHub / Twitter/ StackOverflow / Instagram)

JavaScript Cli Nodejs NPM Cli Apps