2 Writing Modules

This chapter covers the following topics

- Node's module system
- Initializing a module
- Writing a modules
- Tooling around modules
- Publishing modules
- Setting up a private module repository
- Best practices

Introduction

In idiomatic Node, the module is the fundamental unit of logic. Any typical application or system consists of generic code and application code. As a best practice, generic shareable code should be held in discrete modules, which can be composed together at the application level, with minimal amounts of domain specific logic. In this chapter we'll learn how Node's module system works, how to create modules for various scenarios and how we can reuse and share our code.

Scaffolding a module

Let's begin our exploration by setting up a typical file and directory structure for a Node module. At the same time we'll be learning how automatically generate a `package.json` file (we refer to this throughout the book as "initializing a folder as a package") and to configure `npm` (Node's package managing tool) with some defaults which can then be used as part of package generation process.

In this recipe, we'll create the initial scaffolding for a full Node module.

Getting ready

> ### Installing Node
>
> If we don't already have Node installed, we can go to https://nodejs.org to

> pick up the latest version for our operating system.

If Node is on our system, then so is the `npm` executable.

`npm` is the default package manager for Node, it's useful for creating, managing, installing and publishing modules.

Before we run any commands, let's tweak the `npm` configuration a little:

```
npm config set init.author.name "<name here>"
```

This will speed up module creation and ensure each package we create has a consistent author name, thus avoiding typos and variation of our name.

> ### npm stands for..
>
> Contrary to popular belief, `npm` is not an acronym for Node Package Manager, in fact it stands for "npm is Not An Acronym", which is why it's not called NINAA.

How to do it

Let's say we want to create a module that converts HSL (hue, saturation, luminosity) values into a hex based RGB representation, such as would be used in CSS (for example: `#fb4a45`).

`hsl-to-hex` seems like a good name, so let's make a new folder for our module and `cd` into it.

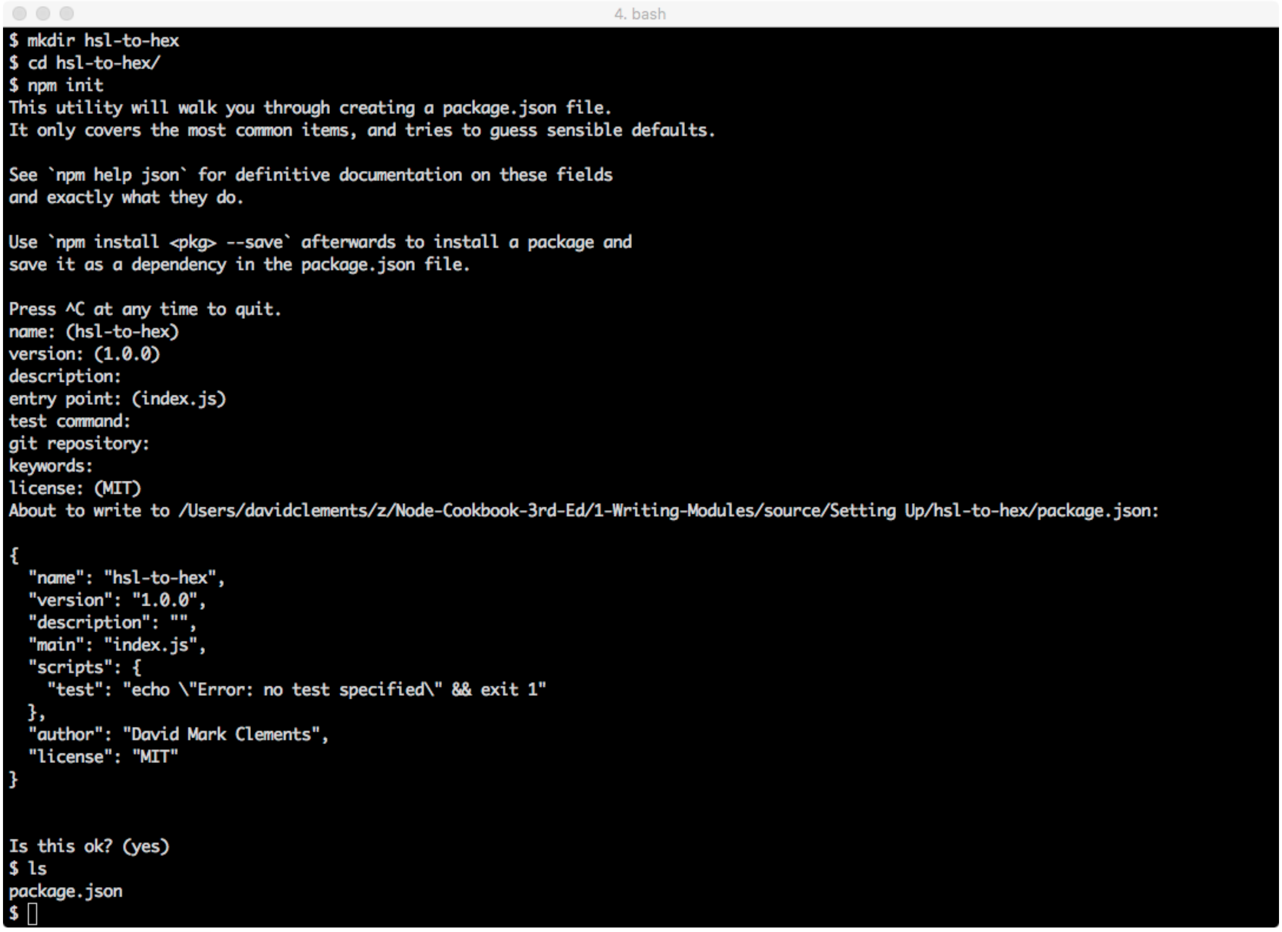
```
mkdir hsl-to-hex
cd hsl-to-hex
```

Every Node module must have a `package.json` file, which holds metadata about the module.

Instead of manually creating a `package.json` file, we can simply execute the following command in our newly created module folder:

```
npm init
```

This will ask a series of questions. We can hit enter for every question without supplying an answer. Notice how the default module `name` corresponds to the current working directory, and the default `author` is the `init.author.name` value we set earlier on.



```
$ mkdir hsl-to-hex
$ cd hsl-to-hex/
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (hsl-to-hex)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
license: (MIT)
About to write to /Users/davidclements/z/Node-Cookbook-3rd-Ed/1-Writing-Modules/source/Setting Up/hsl-to-hex/package.json:

{
  "name": "hsl-to-hex",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "David Mark Clements",
  "license": "MIT"
}


Is this ok? (yes)
$ ls
package.json
$ 
```

An `npm init` should look like this

Upon completion we should have a `package.json` file that looks something like the following:

```json
{
  "name": "hsl-to-hex",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "David Mark Clements",
  "license": "MIT"
}
```

How it Works

When Node is installed on our system, `npm` comes bundled with it.

The `npm` executable is written in JavaScript, and runs on Node.

The `npm config` command can be used to permanently alter settings. In our case we changed the `init.author.name` setting so that `npm init` would reference it for the default during a modules initialization.

We can list all current configuration settings with `npm config ls`.

> ### Config Docs
>
> See https://docs.npmjs.com/misc/config for all possible `npm` configuration settings

When we run `npm init` the answers to prompts are stored in an object, serialized as JSON and then saved to a newly created `package.json` file in the current directory.

There's More

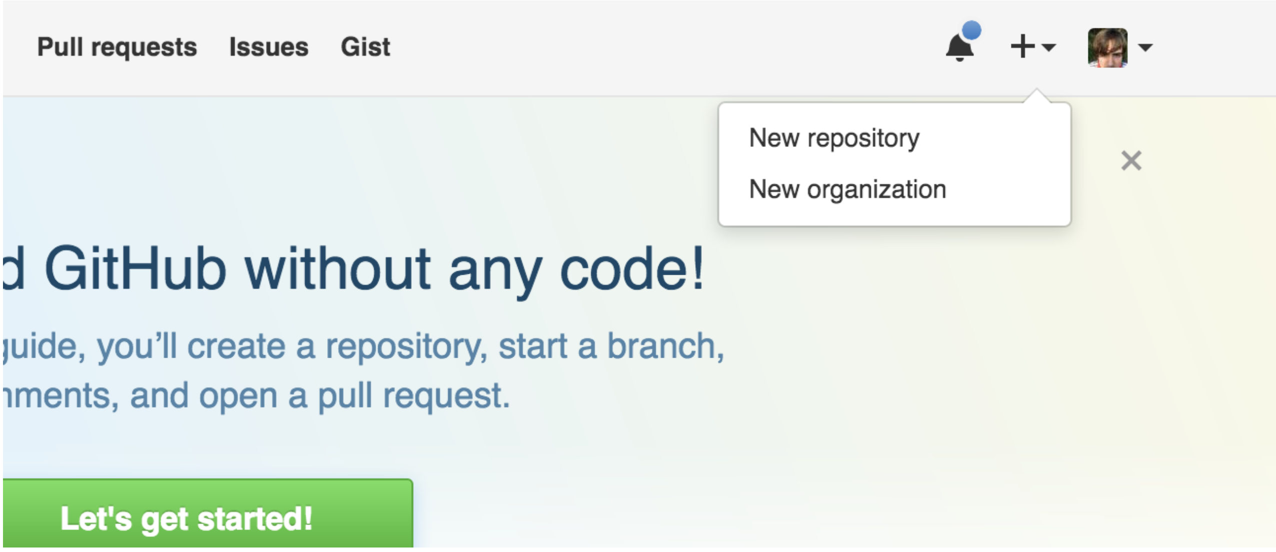
Let's find out some more ways to automatically manage the content of the `package.json` file via the `npm` command.

Reinitializing

Sometimes additional metadata can be available after we've created a module. A typical scenario can arise when we initialize our module as a git repository and add a remote endpoint after creating the module.

> #### Git and GitHub
>
> If we've not used the `git` tool and GitHub before, we can refer to http://help.github.com to get started.
>
> If we don't have a GitHub account we can head to http://github.com to get a free account.

To demonstrate, let's create a GitHub repository for our module. Head to GitHub and click the plus symbol in the top right, then select "new repository".



Select "New repository"

Specify the name as "hsl-to-hex" and click "Create Repository".

Back in the terminal, inside our module folder, we can now run:

```
echo -e "node_modules\n*.log" > .gitignore
git init
git add .
git commit -m '1st'
git remote add origin http://github.com/<username>/hsl-to-hex
git push -u origin master
```

Now here comes the magic part, let's initialize again (simply press enter for every question):

```
npm init
```

```
$ git init
Initialized empty Git repository in /Users/davidclements/z/Node-Cookbook-3rd-Ed/1-Writing-Modules/source/Setting Up/hsl-to-hex/.git/
$ git add package.json
$ git commit -m '1st'
[master (root-commit) eed6409] 1st
 1 file changed, 11 insertions(+)
 create mode 100644 package.json
$ git remote add origin http://github.com/davidmarkclements/hsl-to-hex
$ git push -u origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 375 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://github.com/davidmarkclements/hsl-to-hex
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (hsl-to-hex)
version: (1.0.0)
description:
git repository: (http://github.com/davidmarkclements/hsl-to-hex)
keywords:
license: (MIT)
About to write to /Users/davidclements/z/Node-Cookbook-3rd-Ed/1-Writing-Modules/source/Setting Up/hsl-to-hex/package.json:

{
  "name": "hsl-to-hex",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "David Mark Clements",
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "git+ssh://git@github.com/davidmarkclements/hsl-to-hex.git"
  },
  "bugs": {
    "url": "https://github.com/davidmarkclements/hsl-to-hex/issues"
  },
  "homepage": "https://github.com/davidmarkclements/hsl-to-hex#readme",
  "description": ""
}


Is this ok? (yes)
$ 
```

Reinitializing

This time the Git remote we just added was detected and became the default answer for the "git repository" question. Accepting this default answer meant that the `repository`, `bugs` and `homepage` fields were added to `package.json`.

A repository field in the `package.json` is an important addition when it comes to publishing open source modules since it will be rendered as a link on the modules information page on http://npmjs.com.

A repository link enables potential users to peruse the code prior to installation. Modules that can't be viewed before use are far less likely to be considered viable.

Versioning

The `npm` tool supplies other functionality to help with module creation and management workflow.

For instance the `npm version` command can allow us to manage our module's version number according to semver semantics.

> ## semver
>
> semver is a versioning standard. A version consists of three numbers separated by a dot, for example `2.4.16`. The position of a number denotes specific information about the version in comparison to other versions. The three positions are known as `MAJOR.MINOR.PATCH`. The PATCH number is increased when changes have been made that don't break existing functionality nor add any new functionality. For instance, a bug fix would be considered a patch. The MINOR number should be increased when new backwards compatible functionality is added. For instance the adding of a method. The MAJOR number increases when backwards-incompatible changes are made. See http://semver.org/ for more information.

If we were to a fix a bug we would want to increase the PATCH number. We could either manually edit the `version` field in `package.json`, setting it to `1.0.1`, or we can execute the following:

```
npm version patch
```

This will increase the version field in one command. Additionally, if our module is a Git repository, it will add a commit based on the version (in our case 'v1.0.1') which we can then immediately push.

When we ran the command, `npm` output the new version number. However we can double check the version number of our module without opening `package.json`:

```
npm version
```

This will output something similar to the following:

```
{ 'hsl-to-hex': '1.0.1',
  npm: '2.14.17',
  ares: '1.10.1-DEV',
  http_parser: '2.6.2',
  icu: '56.1',
  modules: '47',
  node: '5.7.0',
  openssl: '1.0.2f',
```

```
  uv: '1.8.0',
  v8: '4.6.85.31',
  zlib: '1.2.8' }
```

The first field is our module along with its version number.

If we added new backwards compatible functionality, we could run:

```
npm version minor
```

Now our version is `1.1.0`.

Finally for a major version bump we can run the following:

```
npm version major
```

This sets the our modules version to `2.0.0`.

Since we're just experimenting and didn't make any changes we should set our version back to `1.0.0`.

We can do this via the `npm` command as well:

```
npm version 1.0.0
```

See also

- *Installing Dependencies* in this chapter
- *Writing module code* in this chapter
- *Publishing a module* in this chapter

Installing Dependencies

In most cases, it is most wise to compose a module out of other modules.

In this recipe we're going to install a dependency.

Getting ready

For this recipe, all we need is a command prompt open in the `hsl-to-hex` folder from the **Scaffolding a module** recipe.

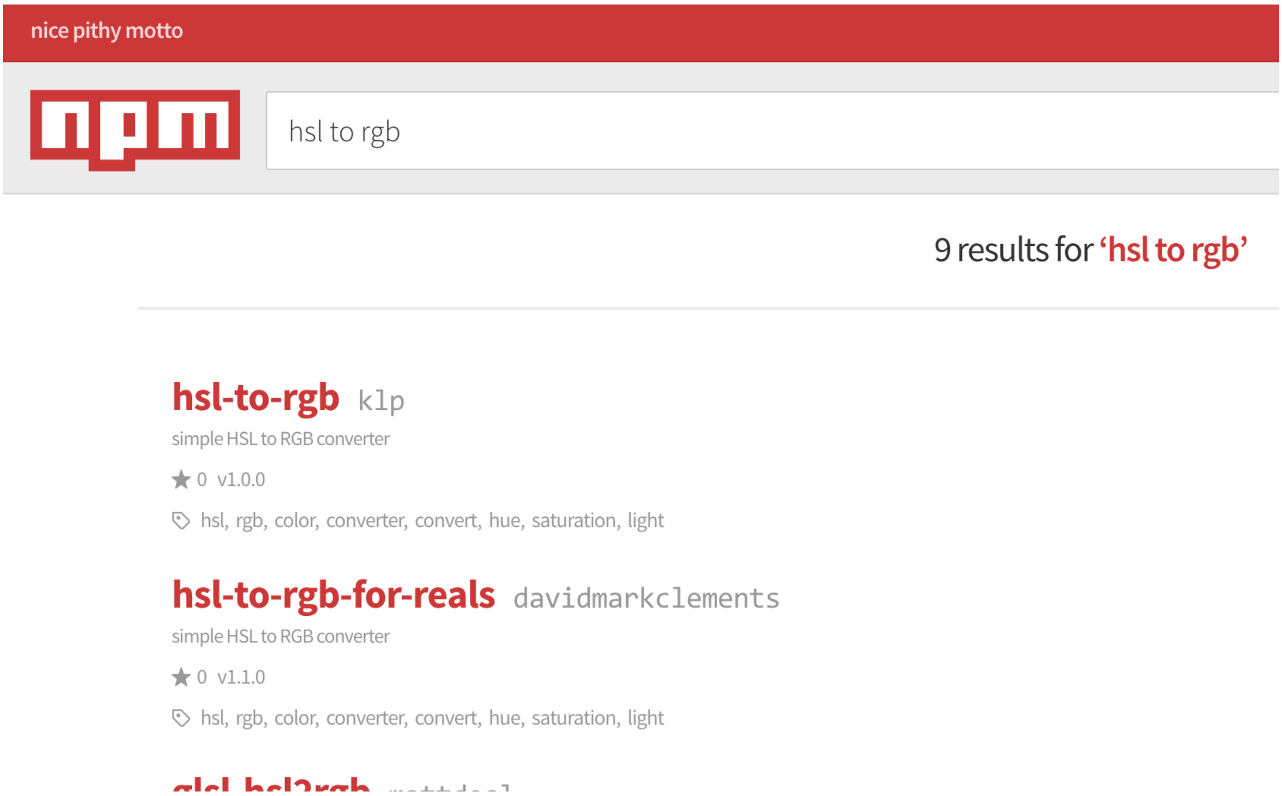
How to do it

Our `hsl-to-hex` module can be implemented in two steps

1. convert the hue degrees, saturation percentage and luminosity percentage to corresponding red, green and blue numbers between 0 and 255
2. convert the RGB values to HEX

Before we tear into writing an HSL to RGB algorithm, we should check whether this problem has already been solved.

The easiest way to check is to head to http://npmjs.com and perform a search.



Oh look somebodies already solved this

After some research we decide that the `hsl-to-rgb-for-reals` module is the best fit.

Ensuring we are in the `hsl-to-hex` folder, we can now install our dependency with the following:

```
npm install --save hsl-to-rgb-for-reals
```

Now let's take a look at the bottom of `package.json`:

```
tail package.json #linux/osx
```

```
type package.json #windows
```

Tail output should give us:

```
  "bugs": {
    "url": "https://github.com/davidmarkclements/hsl-to-hex/issues"
  },
  "homepage": "https://github.com/davidmarkclements/hsl-to-hex#readme",
  "description": "",
  "dependencies": {
    "hsl-to-rgb-for-reals": "^1.1.0"
  }
}
```

We can see that the dependency we installed has been added to a `dependencies` object in the `package.json` file.

How it works

The top two results of the npm search are `hsl-to-rgb` and `hsl-to-rgb-for-reals`. The first result is unusable, because the author of the package forgot to export it and is unresponsive to fixing it. The `hsl-to-rgb-for-reals` module is a fixed version of `hsl-to-rgb`.

This situation serves to illustrate the nature of the npm ecosystem.

On the one hand there are over 200,000 modules and counting, on the other many of these modules are of low value. Nevertheless, the system is also self healing, in that if a module is broken and not fixed by the original maintainer a second developer often assumes responsibility and publishes a fixed version of the module.

When we run `npm install` in a folder with a `package.json` file, a `node_modules` folder is created (if it doesn't already exist). Then the package is downloaded from the npm registry and saved into a subdirectory of `node_modules` (for example, `node_modules/hsl-to-rgb-for-reals`).

> ## npm 2 vs npm 3
>
> Our installed module doesn't have any dependencies of its own. But if it did the sub-dependencies would be installed differently depending on whether we're using version 2 or version 3 of `npm`.
>
> Essentially `npm` 2 installs dependencies in a tree structure, for instance `node_modules/dep/node_modules/sub-dep-of-dep/node_modules/sub-dep-of-sub-dep`. Conversely `npm` 3 follows a maximally flat strategy where sub-dependencies are installed in the top level `node_modules` folder when possible. For example `node_modules/dep`, `node_modules/sub-dep-of-dep` and `node_modules/sub-dep-of-sub-dep`. This results in fewer downloads and less disk space usage. `npm` 3 resorts to a tree structure in cases where there's two version of a sub-dependency, which is why it's called a "maximally" flat strategy.
>
> Typically if we've installed Node 4 or above, we'll be using `npm` version 3.

There's more

Let's explore development dependencies, creating module management scripts and installing global modules without requiring root access.

Installing Development Dependencies

We usually need some tooling to assist with development and maintenance of a module or application. The ecosystem is full of programming support modules, from linting, to testing to browser bundling to transpilation.

In general we don't want consumers of our module to download dependencies they don't need. Similarly, if we're deploying a system built in node, we don't want to burden the continuous integration and deployment processes with superfluous, pointless work.

So we separate our dependencies into production and development categories.

When we use `npm --save install <dep>` we're installing a production module.

To install a development dependency we use `--save-dev`.

Let's go ahead and install a linter.

> **standard**
>
> `standard` is a JavaScript linter that enforces an unconfigurable rule set The premise of this approach is that we should stop using precious time up on bikeshedding about syntax.

All the code in this book uses the `standard` linter so we'll install that.

```
npm install --save-dev standard
```

> **semistandard**
>
> If the absence of semi-colons is abhorrent, we can choose to install `semistandard` instead of `standard` at this point. The lint rules match those of `standard`, with the obvious exception of requiring semi-colons. Further, any code written using standard can be reformatted to `semistandard` using the `semistandard-format` command tool. Simply `npm -g i semistandard-format` to get started with it.

Now let's take a look at the `package.json` file:

```
{
  "name": "hsl-to-hex",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "David Mark Clements",
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "git+ssh://git@github.com/davidmarkclements/hsl-to-hex.git"
  },
  "bugs": {
    "url": "https://github.com/davidmarkclements/hsl-to-hex/issues"
  },
  "homepage": "https://github.com/davidmarkclements/hsl-to-hex#readme",
  "description": "",
  "dependencies": {
    "hsl-to-rgb-for-reals": "^1.1.0"
  },
  "devDependencies": {
    "standard": "^6.0.8"
  }
```

```
}
```

We now have a `devDependencies` field alongside the `dependencies` field.

When our module is installed as a sub-dependency of another package, only the `hsl-to-rgb-for-reals` module will be installed whilst the `standard` module since will be ignored since it's irrelevant to our modules actual implementation.

If this `package.json` file represented a production system we could run the install step with the `--production` flag like so:

```
npm install --production
```

Alternatively, this can be set in production environment with the following command:

```
npm config set production true
```

Currently we can run our linter using the executable installed in the `node_modules/.bin` folder. For example:

```
./node_modules/.bin/standard
```

This is ugly and not at all ideal. See Using npm run scripts for a more elegant approach.

Using npm run scripts

Our `package.json` file currently has a `scripts` property that looks like this:

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Let's edit the `package.json` file and add another field, called `lint`.

Like so:

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
```

```
    "lint": "standard"
  },
```

Now, as long as we have `standard` installed as a development dependency of our module (see Installing Development Dependencies), we can run the following command to run a lint check on our code:

```
npm run-script lint
```

This can be shortened to:

```
npm run lint
```

When we run an npm script, the current directory's `node_modules/.bin` folder is appended to the execution contexts `PATH` environment variable. This means even if we don't have the `standard` executable in our usual system `PATH`, we can reference it in an npm script as if it was in our `PATH`.

Some consider lint checks to be a precursor to tests.

Let's alter the `scripts.test` field like so:

```
  "scripts": {
    "test": "npm run lint",
    "lint": "standard"
  },
```

> **Chaining commands**
>
> Later we could append other commands to the `test` script using the double ampersand (`&&`), to run a chain of checks. For instance, `"test": "npm run lint && tap test"`

Now to run the `test` script:

```
npm run test
```

Since the `test` script is special, we can simply run

```
npm test
```

Eliminating the need for `sudo`

The `npm` executable can install both local and global modules. Global modules are mostly installed so their as command line utilities can be used system wide.

On OSX and Linux the default `npm` set up requires `sudo` access to install a module.

For example, for following will fail on a typical OS X or Linux system with the default `npm` set up:

```
npm -g install cute-stack # <-- oh oh needs sudo
```

This is unsuitable for several reasons. Forgetting to use `sudo` becomes frustrating, we're trusting `npm` with root access and accidentally using `sudo` for a local install causes permissions problems (particularly with the npm local cache).

The `prefix` setting stores the location for globally installed modules, we can view this with:

```
npm config get prefix
```

Usually the output will be `/usr/local`. To avoid the use of `sudo` all we have to do is set ownership permissions on any subfolders in `/usr/local` used by `npm`:

```
 sudo chown -R $(whoami) $(npm config get prefix)/{lib/node_modules,bin
```

Now we can install global modules without root access:

```
npm -g install cute-stack # <-- now works without sudo
```

If changing ownership of system folders isn't feasible, we can use a second approach which involves changing the `prefix` setting to a folder in our home path:

```
mkdir ~/npm-global
npm config set prefix ~/npm-global
```

We'll also need to set our `PATH`:

```
export PATH=$PATH:~/npm-global/bin
source ~/.profile
```

The `source` essentially refreshes the terminal environment to reflect the changes we've made.

See also

- *Scaffolding a module* in this chapter
- *Writing module code* in this chapter
- *Publishing a module* in this chapter

Writing module code

Now it's time to engage in actual implementation details.

In this recipe we're going to write some code for our `hsl-to-hex` module.

Getting ready

Let's ensure that we have a folder called `hsl-to-hex`, with a `package.json` file in it. The `package.json` file should contain `hsl-to-rgb-for-reals` as a dependency. If there isn't a `node_modules` folder, we need to make sure we run `npm install` from the command line with the working directory set to the `hsl-to-hex` directories path.

To get started let's create a file called `index.js in the`hsl-to-hex` folder, then open it in our favorite text editor.

How to do it

The first thing we'll want to do in our `index.js` file is specify any dependencies we'll be using.

In our case, there's only one dependency

```
var toRgb = require('hsl-to-rgb-for-reals')
```

Typically all dependencies should be declared at the top of the file.

Now let's define an API for our module, we're taking hue, saturation and luminosity values and outputting a CSS compatible hex string.

Hue is in degrees, between 0 and 359. Since degrees are cyclical in nature, we'll support numbers greater than 359 or less than 0 by "spinning" them around until they fall within the 0 to 359 range.

Saturation and luminosity are both percentages, we'll represent these percentages with whole numbers between 0 and 100. For these numbers we'll need to enforce a maximum and a minimum, anything below 0 will become 0, anything above 100 will become 100.

Let's write some utility functions to handle this logic:

```
function max (val, n) {
  return (val > n) ? n : val
}

function min (val, n) {
  return (val < n) ? n : val
}

function cycle (val) {
  // for safety:
  val = max(val, 1e7)
  val = min(val, -1e7)
  // cycle value:
  while (val < 0) { val += 360 }
  while (val > 359) { val -= 360 }
  return val
}
```

Now for the main piece, the `hsl` function:

```
function hsl (hue, saturation, luminosity) {
  // resolve degrees to 0 - 359 range
  hue = cycle(hue)

  // enforce constraints
  saturation = min(max(saturation, 100), 0)
  luminosity = min(max(luminosity, 100), 0)

  // convert to 0 to 1 range used by hsl-to-rgb-for-reals
  saturation /= 100
```

```
  luminosity /= 100

  // let hsl-to-rgb-for-reals do the hard work
  var rgb = toRgb(hue, saturation, luminosity)

  // convert each value in the returned RGB array
  // to a 2 character hex value, join the array into
  // a string, prefixed with a hash
  return '#' + rgb
    .map(function (n) {
      return (256 + n).toString(16).substr(-2)
    })
    .join('')
}
```

In order to make our code into a bona fide module we have to export it:

```
module.exports = hsl
```

We can run a few sanity checks to ensure our code is working.

Maximum saturation and luminosity should be white (`#ffffff`), regardless of hue. So, with our current working directory set to our modules folder, let's try the following:

```
node -p "require('./')(0, 100, 100)"
```

This should print `#ffffff` .

> ## The -p flag
>
> The `-p` flag tells node to evaluate the supplied string and print the result to the terminal.

Okay that was easy. Let's try another test. A saturation of 0% and a luminosity of 50% should create red, green and blue values that are half way between 0 and 256 (128). In hex this is `80` , so the following should output `#808080` :

```
node -p "require('./')(0, 0, 50)"
```

We've checked luminosity and saturation, let's finish by ensuring that hue input works as expected.

Hue represents the color spectrum, starting and finishing with red, defined in degree points.

Hue degrees

As we know, setting both saturation and luminosity to 100% will always result in white. After 50% luminosity colours beyond the defined hue will be added to further increase the brightness of the colour. This means that we should get a pure hue by setting saturation to 100% and luminosity to 50%.

So, the following should output `#ff0000` (red)

```
node -p "require('./')(0, 100, 50)"
```

A hue of 240 should give exact blue (`#0000ff`):

```
node -p "require('./')(240, 100, 50)"
```

And 180 should result in cyan (`#00ffff`):

```
node -p "require('./')(180, 100, 50)"
```

How it works

The algorithmic heavy lifting is performed by our dependency `hsl-to-rgb-for-reals` . This is often the case in the landscape of Node's ecosystem. Many fundamental computer science problems have already been solved (often multiple times) by third party contributors.

Our `index.js` exports a single function, the `hsl` function. This function applies sanity to the inputs (like rotating 360 degrees back to 0 and enforcing minimum and maximums) and then convert the output from decimal values to hex values, prefixing them with a hash (`#`).

Since the `hsl-to-rgb-for-reals` module returns an array of values between 0 and 255, we can use the native `map` method to iterate over each element in the array and convert it from base 10 to base 16. Then we `join` the resulting array into a string.

In our quick command line checks, we call the `node` binary with the `-p` flag. This simply evaluates a supplied expression, and outputs its value. In each case the expression involves calling `require`.

The `require` function is central to Node's module system, when it's called the module system performs a series of steps.

First `require` has to locate the module according to the supplied argument. Depending on the input, the module may be a local file, a core module or a separately installed module.

We supplied a path `'./'`, so the `require` function attempts to load the current directory as a module. In order to do this it looks for a `package.json` file, and looks up the `main` field in the `package.json` file. The `main` field in our `package.json` file is `'index.js'`, so `require` recognises this file as the modules entry point. In the absence of a `package.json` file or `main` field, `require` also defaults to `index.js` as the entry point.

Once an entry point file has been identified, Node synchronously loads it into a string. The modules code is wrapped with the following:

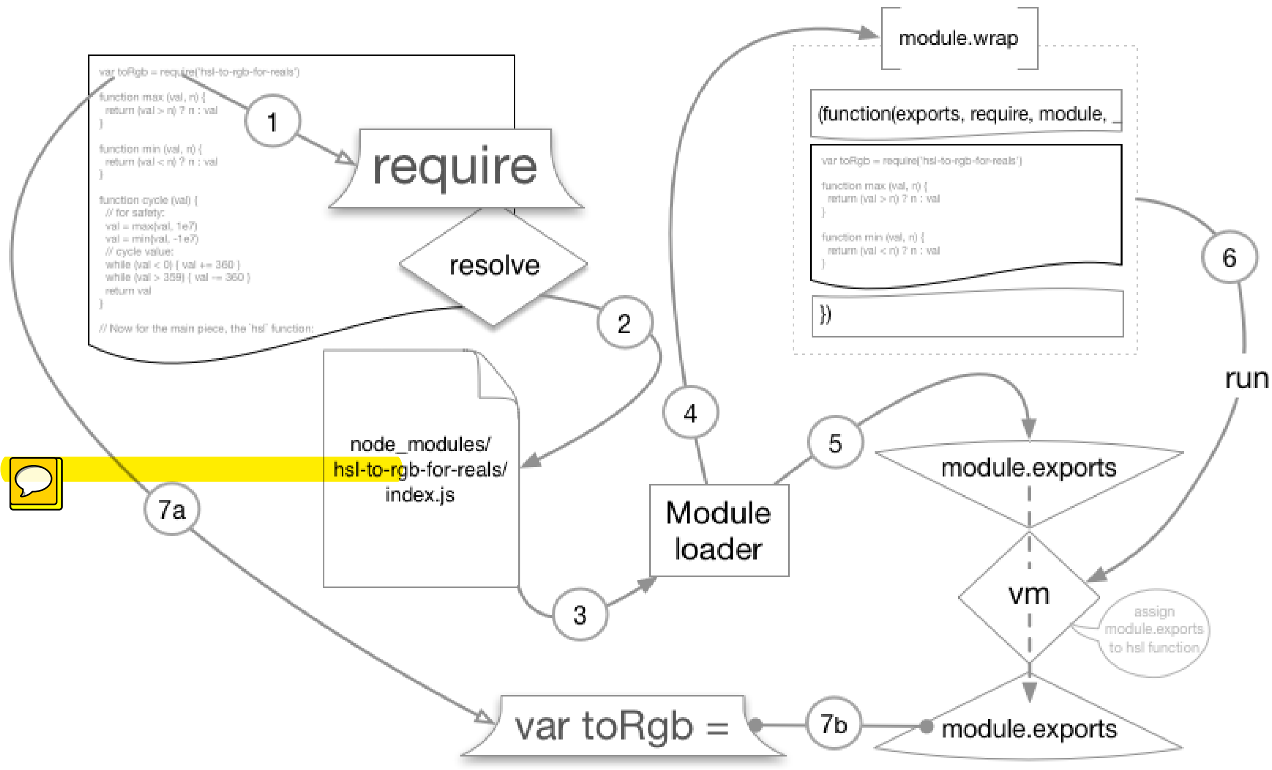
```
(function (exports, require, module, __filename, __dirname) {
  /* module code here */
})
```

The resulting string is passed through the `vm` module's `runInThisContext` method, which essentially tells the JavaScript engine to compile the string into a function. This function is then called with the five parameters dictated in the wrapper (`exports`, `require`, `module`, `__filename`, `__dirname`). The `exports` argument is an empty object, the `module` argument is an object with an `exports` property pointing to the `exports` object. So there are two reference to the initial exports object: the `exports` parameter and the `module.exports` property.

The value returned from `require` is the `module.exports` property.

In our code we overwrote the `module.exports` property with the `hsl` function, which is why we can call the result of `require` immediately (for example `require('./')(180, 100, 50)`).

The following diagram serves to visualize the module loading process at a high level:



How Node loads modules

There's more

Let's add tests for our module, and then take a look at some newer language features which we can use in our test writing.

Adding tests

If bugs arise, or we decide to make changes, or extend functionality it would be nice if we could run a single command that runs some checks against our code so we can be confident that we're not unintentionally breaking anything. We could lump all of our `node -p` checks from the main recipe into a single bash (or batch) file, but there's a more standard and elegant approach.

Let's write some tests.

First we'll need a test library, let's use the `tap` module. The `tap` tool is simple, doesn't require it's own test runner, has built in coverage analysis and outputs TAP, the Test Anything Protocol, which is used across many languages.

```
npm install --save-dev tap
```

Remember we're installing with `--save-dev` because this dependency would not be required in production.

Now, assuming we're in the `hsl-to-hex-folder`, let's create a `test` folder

```
mkdir test
```

> ### Test writing
>
> For an excellent article on test writing and TAP output, check out Eric Elliots blog post Why I use Tape instead of Mocha & so should you at https://medium.com/javascript-scene/6aa105d8eaf4

Now, in the `test` folder let's create an `index.js` file, with the following code:

```
var hsl = require('../')
var test = require('tap').test

test('pure white', function (assert) {
  var expected = '#ffffff'
  var actual = hsl(0, 100, 100)
  var it = 'max saturation and luminosity should return pure white'
  assert.is(actual, expected, it)
  assert.end()
})

test('medium gray', function (assert) {
  var expected = '#808080'
  var actual = hsl(0, 0, 50)
  var it = '0% saturation, 50% luminosity should be medium gray'
  assert.is(actual, expected, it)
  assert.end()
})

test('hue - red', function (assert) {
  var expected = '#ff0000'
  var actual = hsl(0, 100, 50)
  var it = '0deg should be red'
  assert.is(actual, expected, it)
  assert.end()
})

test('hue - blue', function (assert) {
```

```
  var expected = '#0000ff'
  var actual = hsl(240, 100, 50)
  var it = '240deg should be blue'
  assert.is(actual, expected, it)
  assert.end()
})

test('hue - cyan', function (assert) {
  var expected = '#00ffff'
  var actual = hsl(180, 100, 50)
  var it = '180deg should be cyan'
  assert.is(actual, expected, it)
  assert.end()
})

test('degree overflow', function (assert) {
  var expected = hsl(1, 100, 50)
  var actual = hsl(361, 100, 50)
  var it = '361deg should be the same as 1deg'
  assert.is(actual, expected, it)
  assert.end()
})

test('degree underflow', function (assert) {
  var expected = hsl(-1, 100, 50)
  var actual = hsl(359, 100, 50)
  var it = '-1deg should be the same as 359deg'
  assert.is(actual, expected, it)
  assert.end()
})

test('max constraint', function (assert) {
  var expected = hsl(0, 101, 50)
  var actual = hsl(0, 100, 50)
  var it = '101% should be the same as 100%'
  assert.is(actual, expected, it)
  assert.end()
})

test('max constraint', function (assert) {
  var expected = hsl(0, -1, 50)
  var actual = hsl(0, 0, 50)
  var it = '-1% should be the same as 0%'
  assert.is(actual, expected, it)
  assert.end()
})
```

In the `package.json` file we'll edit the `scripts.test` field to read thusly:

```
    "test": "npm run lint && tap --cov test",
```

We can see if our tests are passing by running `npm test`

```
npm test
```

We also get to see a coverage report which was enabled with the `--cov` flag.

> ### coverage
>
> Coverage is a percentage of the amount of logic paths that were touched by our tests. This can be measured in several ways, for instance did we cover all the if/else branches? Did we cover every line of code? This can provide a sort of quality rating for our tests. However there are two things to consider when it comes to coverage. First, 100% coverage does not equate to 100% of possible scenarios. There could be some input that causes our code to crash or freeze. For example, what if we passed in a hue of `Infinity`. In our case we've handled that scenario, but haven't tested it. Yet we have 100% coverage. Secondly, in many real world cases, getting the last 20% of coverage can become the most resource intensive part of development and it's debatable whether that last 20% will deliver on the time and effort investment required.

Modernizing syntax

Recent Node.js versions support modern JavaScript syntax (known as EcmaScript 6), let's use some of these shiny new JavaScript features to improve the tests we wrote in the previous section.

> ### EcmaScript 6
>
> Learn more about EcmaScript 6 at http://es6-features.org/

For this to work, we'll need at least Node version 5 installed, preferable Node version 6 or greater.

> ### Managing Node Versions
>
> Check out nvm or n for an easy way to switch between node versions

Node v6 and above should support all of the syntax we'll be using, Node v5 will support all of it, as long as we pass a special flag.

> ### Transpilation
>
> For version below Node 5, or to use syntax that currently isn't available in even in the latest versions of Node we can fall back to transpilation. Transpilation is essentially compiling a later version of a language into an earlier version. This is beyond our scope, but check out for more information on how to transpile.

If we're using Node version 5 we'll need to adjust the `test` field in the `package.json` file like so:

```
"test": "npm run lint && tap --node-arg='--harmony-destructuring' --cov
```

The `--node-arg` is supplied to the `tap` test runner to pass through a Node specific flag which will be applied via tap when it runs out tests.

In this case we passed the `--harmony-destructuring` flag, this turns on an experimental syntax in Node version 5 (as mentioned, we don't need to do this for Node v6 and up).

> ### Syntax Switches
>
> Get a full list of experimental syntax and behaviours by running `node --v8-options | grep harmony` or if we're on windows: `node --v8-options | findstr harmony`.

Now let's rewrite out test code like so:

```
const hsl = require('../')
const {test} = require('tap')

test('pure white', ({is, end}) => {
  const expected = '#ffffff'
  const actual = hsl(0, 100, 100)
  const it = `
    max saturation and luminosity should return pure white
  `
  is(actual, expected, it)
  end()
})
```

```
test('medium gray', ({is, end}) => {
  const expected = '#808080'
  const actual = hsl(0, 0, 50)
  const it = `
    0% saturation, 50% luminosity should be medium gray
  `
  is(actual, expected, it)
  end()
})

test('hue', ({is, end}) => {
  {
    const expected = '#ff0000'
    const actual = hsl(0, 100, 50)
    const it = `
      0deg should be red
    `
    is(actual, expected, it)
  }
  {
    const expected = '#0000ff'
    const actual = hsl(240, 100, 50)
    const it = `
      240deg should be blue
    `
    is(actual, expected, it)
  }
  {
    const expected = '#00ffff'
    const actual = hsl(180, 100, 50)
    const it = `
      180deg should be cyan
    `
    is(actual, expected, it)
  }
  end()
})

test('degree overflow/underflow', ({is, end}) => {
  {
    const expected = hsl(1, 100, 50)
    const actual = hsl(361, 100, 50)
    const it = `
      361deg should be the same as 1deg
    `
    is(actual, expected, it)
  }
  {
    const expected = hsl(-1, 100, 50)
    const actual = hsl(359, 100, 50)
    const it = `
```

```
      -1deg should be the same as 359deg
    `
    is(actual, expected, it)
  }
  end()
})

test('max constraint', ({is, end}) => {
  {
    const expected = hsl(0, 101, 50)
    const actual = hsl(0, 100, 50)
    const it = `
      101% should be the same as 100%
    `
    is(actual, expected, it)
  }
  {
    const expected = hsl(0, -1, 50)
    const actual = hsl(0, 0, 50)
    const it = `
      -1% should be the same as 0%
    `
    is(actual, expected, it)
  }
  end()
})
```

Here we've used several EcmaScript 6 features, all of which are available out-of-the-box in Node 6.

The features we've used are

- destructuring assignment (enabled with --harmony-destructuring on Node v5)
- arrow functions
- template strings (also known as template literals)
- `const` and block scope

Destructuring is an elegant shorthand for taking property from an object and loading into a variable.

We first use destructuring assignment early on in our rewritten tests when we take `test` method from the exported `tap` object and load it into the `test` `const`, like so:

```
const {test} = require('tap')
```

This is equivalent to:

```
const test = require('tap').tap
```

On one line this doesn't deliver much value but destructuring reveals its terse simplicity when we wish to extract several properties from an object and assign to variables of the same name.

For instance, the following:

```
var foo = myObject.foo
var bar = myObject.bar
var baz = myObject.baz
```

Can be achieved in one line with destructuring, with less noise and (subjectively) greater readability. Like so:

```
var {foo, bar, baz} = myObject
```

We also destructure the `assert` object in each of our `test` callbacks.

For instance the top line of first test looks like this:

```
test('pure white', ({is, end}) => {
```

Parameter destructuring allows us to focus only on the properties we're interested in using for that function. It presents a clear contract to whatever's calling our function. Namely, for our case, the input object should have `is` and `end` properties.

> ### Destructuring
>
> Find more detail about destructuring at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Each of the callbacks supplied to our test are arrow functions, which look like this:

```
var fn = (some, params) => { return 'something' }
```

When it makes sense, arrow functions we can also omit the braces and the `return` keyword:

```
[1,2,3,4,5].map(n => n * n) // [1, 4, 9, 16, 25]
```

We use arrow functions purely for aesthetics, removing noise enhances the focus of our code.

> ## Arrow Functions
>
> We should note that arrow functions behave differently to normal functions, in particular when it comes to the `this` context. Find out more about arrow functions at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Template strings are denoted with backticks (`` ` ``) and can be multiline. We use these for our `it` constants purely for the multiline capabilities, since describing behaviour can often take up more than 80-100 columns, or more than one line. Template strings (the clue being in the name), also supply interpolation, like so:

```
var name = 'David'
console.log(`Hi name is ${name}`)
```

> ## Template strings
>
> Find out more about template strings at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Finally, EcmaScript 6 supplies two additional assignment keywords to the JavaScript lexicon: `let` and `const`. In JavaScript the lifetime of `var` assigned reference occurs within the closest parent function. The `let` and `const` keywords introduce the more traditional block-scoped behavior, where the references lifetime is relative to the closest block (for example, as denoted by the braces in a `for` loop). We used this to our advantage and merged some of the tests together. For instance, `hue - blue`, `hue - red` and `hue - cyan` could all be separate assertions in a single `hue` test. Block scoping makes it easy to maintain the repeating pattern of `expected`, `actual`, and `it` without clashing with neighbouring assertions. The `let` keyword is similar to `var` in that it allows for reassignment. The `const` keyword is a constant *reference* to a value (it does **not** make the value a constant). We use `const` throughout since we have no intention of reassigning our references, and if that occurred it would be a bug (in which case our tests would conveniently throw).

> ### Block Scope In JavaScript
>
> For more info about block scope a look at Dr. Axel Rauschmayer's article at http://www.2ality.com/2015/02/es6-scoping.html
>
> ### Public Code and EcmaScript 6
>
> In the next recipe we'll look at how to publish a module. If we plan to make our module for public consumption, using language features which aren't available across different Node versions prohibits the user base of our module. Generally, it's better to use language features which cater to the lowest common denominator and only carefully using newer parts of the language once market share of that feature has been determined at an acceptably high level.

See also

- *Scaffolding a module* in this chapter
- *Installing Dependencies* in this chapter
- *Publishing a module* in this chapter
- *Debugging Node with Chrome Devtools* in **Chapter 1 Debugging Processes**

Publishing a module

In this recipe we'll prepare our module to be published then publish it as a scoped package.

Getting ready

We're going to publish our `hsl-to-hex` module we've been working on in previous recipes. We'll also want the (original) tests we wrote in the Adding tests portion of the the There's more section of the Writing module code recipe.

If we don't have an npmjs.org account we'll need to head over to https://www.npmjs.com/signup and get an account. Keep the npm username handy, we're going to need it.

How to do it

If we've just signed up for an npm account (as explained in the previous Getting ready section) we'll want to authorise our `npm` client with npmjs.org.

On the command line, we simply need to run:

```
npm login
```

Then supply the username, password, and email address we signed up with.

Every module should have a Readme explaining how it works.

Let's create a `Readme.md` file with the following markdown:

````
# hsl-to-hex

Convert HSL colors to RGB colors in hex format.

## Install

```sh
npm install --save @davidmarkclements/hsl-to-hex
```

## API

```
require('hsl-to-hex') => Function
hsl(hue, saturation, luminosity)` => String
```

## Example
````

```
var hsl = require('hsl-to-hex')
var hue = 133
var saturation = 40
var luminosity = 60
var hex = hsl(hue, saturation, luminosity)
console.log(hex) // #70c282
```

License

ISC

In the install section of the readme, we should replace `@davidmarkclements/hsl-to-hex` with out own username.

> ### Markdown
>
> Markdown is a lightweight documentation syntax, see https://guides.github.com/features/mastering-markdown/

As a courtesy, we'll also take the example we wrote in the Readme and put it in an `example.js` file.

Let's create a file called `example.js` with the following contents:

```
var hsl = require('./')
var hue = 133
var saturation = 40
var luminosity = 60
var hex = hsl(hue, saturation, luminosity)
console.log(hex) // #70c282
```

Notice how we've made a minor adjustment to the example code, instead of requiring `hsl-to-hex` we're requiring `./`. This ensures that the `example.js` file will run.

Now we'll make some final touches the `package.json` file.

First we'll reinitialize the module:

```
npm init
```

Following this command, we can simply press Enter in response to all questions. The output of `npm init` should show that a `description` field has been added, with its content taken from the `readme.md` file.

```
"description": "Convert HSL colors to RGB colors in hex format.",
```

Now let's open the `package.json` file and change the `name` field, by prefixing it with an at (`@`) symbol, followed by our npm username, followed by a forward slash (`/`). For instance:

```
"name": "@davidmarkclements/hsl-to-hex",
```

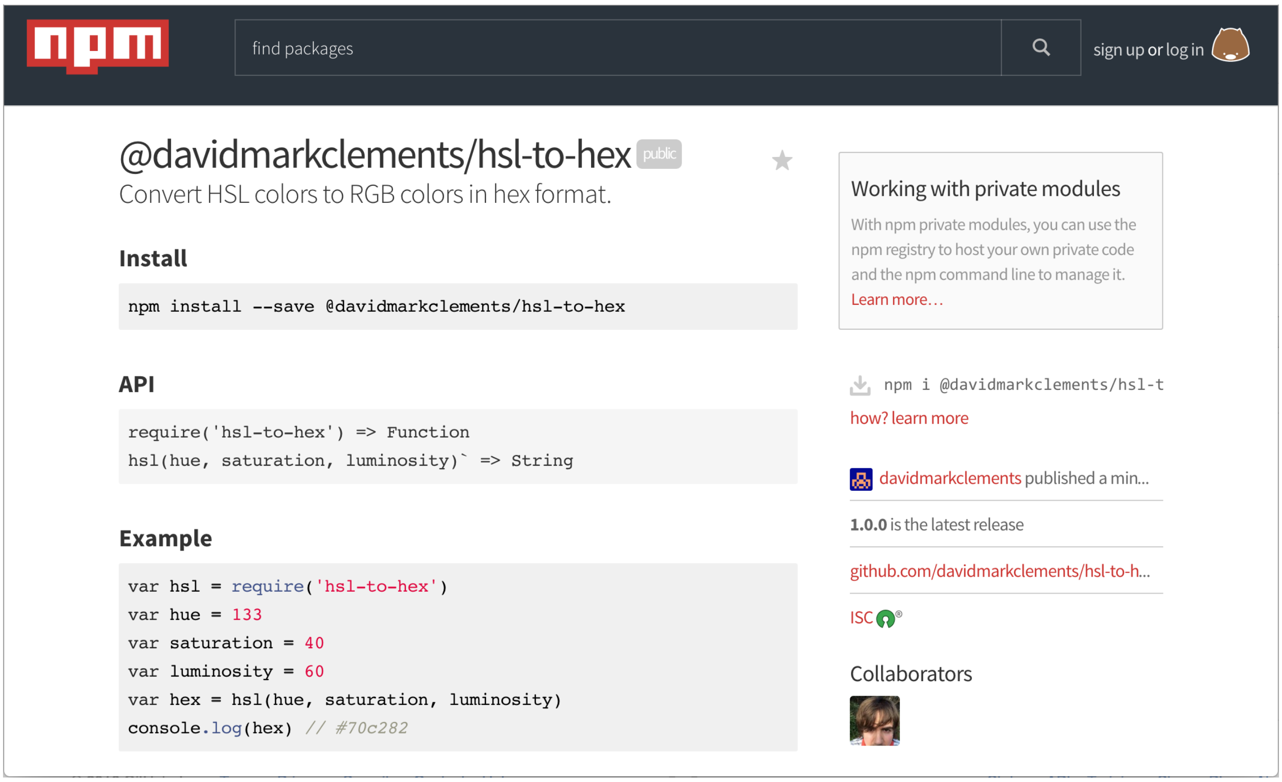
Of course, instead of using `@davidmarkclements` we'll use whatever username we supplied to `npm login`.

> ### Extra Credit: Push to GitHub
>
> If we followed the Reinitializing portion of the There's More section in the Scaffolding a module recipe we could also take this opportunity to push to GitHub just before we publish. This can be useful in helping users explore code and clone our repo to execute the example, run tests or even fix bugs or add features which can be contributed back to our module.
>
> To do this we can run
>
> ```
> git add .
> git commit -m 'v1.0.0'
> git push
> ```

Finally, we're ready to publish:

```
npm publish --access=public
```

We should now be able to navigate to https://www.npmjs.com/package/@davidmarkclements/hsl-to-hex (where `@davidmarkclements` is the username we're using) and view the modules npm page.



How it works

The npm registry allows for global and scoped package names. We converted our module's name to a scoped namespace in order to avoid naming conflicts. This meant we had to pass the `--access=public` flag to along with the `npm publish` command. If we we're publishing a module to the global namespace this wouldn't be required since any modules in the global namespace are public. Scoped packages, however, allow for both private and public publishing where restricted access is the default.

When we ran the `npm publish` command, the `npm` tool packaged up our module and sent it to the npm registry. The npm registry stored our module and analysed the `package.json` file to and `readme.md` file to create a page for our module on npmjs.com.

Subsequently, now that our module is in the registry it can be installed as a dependency of other modules and applications.

For instance:

```
mkdir my-app
cd my-app
npm init
```

```
npm install @davidmarkclements/hsl-to-hex
```

There's more

Prior to publishing a module (or deploying a system), there's several things we want to check so that our module isn't dead on arrival, or worse: alive and dangerous. Let's explore some tools and approaches for handling module security, broken dependencies and automated checks.

Detecting Vulnerabilities

Whilst, by no means, a substitute for thorough penetration testing the `auditjs` tool can help to catch some security holes in our modules and applications.

Let's check it out:

```
npm i -g auditjs
```

Now let's run a security sweep.

In our `hsl-to-hex` module folder, we simply execute the following:

```
auditjs
```

When we run `auditjs` the entire `node_modules` tree is scanned against the OSS Index. The OSS Index contains recorded security vulnerabilities from various sources including the National Vulnerability Datase, the Node Security Project and http://npmjs.com itself. Additionally, `auditjs` will check the current Node version in use for any security announcement.

> ### OSS Index
>
> For more on the OSS Index visit https://ossindex.net/

Extraneous Dependencies

It can be all too easy to publish modules without necessary dependencies.

Let's say we install the `debug` module because we want to instrument our code with debug messages.

In the `hsl-to-hex` folder we could run:

```
npm install debug
```

> ## Debugging
>
> The purpose of this section is to demonstrate extraneous dependencies. The fact we're using the `debug` module is peripheral, however we will be going into detail on debugging in **Chapter 9 Debugging Systems**.

Now let's use debug in our `index.js` file.

At the top of the file, we can create a debug logger:

```
var toRgb = require('hsl-to-rgb-for-reals')
var debug = require('debug')('hsl-to-hex')
```

Now let's add some debug output in each of our functions.

For the `max` function we'll add the following:

```
function max (val, n) {
  debug('ensuring ' + val + 'is no more than ' + n)
/*...snip..*/
```

Likewise for the `min` function:

```
function min (val, n) {
  debug('ensuring ' + val + 'is no less than ' + n)
/*...snip..*/
```

And finally the `cycle` function:

```
function cycle (val) {
  debug('resolving ' + val + ' within the 0-359 range')
/*...snip..*/
```

Ok, looks good.

Let's run our tests to make sure everything is working:

```
npm test
```

We can also check out debug logs work by running

```
DEBUG=hsl-to-hex node example.js
```

If all went well we could be fooled into believing that we are ready to publish a new version of our module. However if we did, it would be broken on arrival.

This is because we neglected to add the `debug` module to the `package.json` file. We omitted the `--save` flag.

The tests and code work for us because the `debug` module is installed, but npm does not know to install it for users of our module because we haven't told npm to do so.

A good habit to get into is running `npm ls` before publishing, like so:

```
npm ls
```

This will output a dependency tree for all our production and development modules. More importantly, it also identifies "extraneous" dependencies. That is, dependencies that are in the `node_modules` folder, but aren't specified in the `package.json` file.

In our case, the bottom of the `npm ls` output will say:

```
npm ERR! extraneous: debug@2.2.0 /Users/davidclements/z/Node-Cookbook-3
```

To fix the extraneous dependency we can manually edit the `package.json` file `dependencies` field, or we can simply run the install command again with the `--save` flag. Like so:

```
npm install --save debug
```

Prepublish

Along with arbitrary commands, and the special `test` field, the `scripts` section of the `package.json` file supports hooks.

The `prepublish` field can be useful for catching mistakes before we send our code out into the world.

> ## npm scripts
>
> Take a look at https://docs.npmjs.com/misc/scripts for list of all supported npm scripts.

Let's add a `prepublish` script, that runs `npm ls` and `npm test`.

```
"scripts": {
  "prepublish": "npm ls && npm test",
  "test": "npm run lint && tap --cov test",
  "lint": "standard"
},
```

Now, before each publish npm should check for extraneous dependencies, run our linter, and run our test suite automatically before each publish.

Let's test it real quick by adding an extraneous dependency as in the previous section.

This time our extraneous dependency can be `clockface`:

```
npm install clockface
```

Okay, now we'll bump the patch version:

```
npm version patch
```

Then we'll try to publish:

```
npm publish
```

At this point, npm should fail to publish and write an `npm-debug.log` to our modules folder.

When `npm ls` fails, it exits with a non-zero exit code. The npm task will respect this exit code and similarly fail. A similar situation would occurs if one of our tests were failing, or if our code contained syntax that broke linting rules.

Finally, let's fix our "mistake" and publish for real:

In this case, we don't need `clockface`, so we'll simply remove it:

```
rm -fr node_modules/clockface
```

And publish:

```
npm publish
```

Now we should see a successful `npm ls` command and we should see linting and tests passing, followed by a successful publish of the next version of our module.

Decentralized Publishing

In the last few decades great strides have been made in the area of distributed computing. An interesting outcome is the prospect that modules can be shared within the community without a central registry.

It's possible today to publish a module to IPFS (the InterPlanetary FileSystem). IPFS is an immutable peer-to-peer file system composed of various innovations in cryptography and decentralized networks from the last 30 years. Including technologies found in `git`, BitTorrent, the Tor network, and BitCoins blockchain.

Let's publish our module to IPFS!

First we'll need to install IPFS. Let's head to https://ipfs.io/docs/install/ and follow the instructions to install IPFS on our system.

Once installed, we need to initialize:

````
ipfs init
``

Then start the IPFS service:

```sh
ipfs daemon
````
````

There's a handy npm module called `stay-cli` that makes it trivial to publish to and install modules from IPFS.

Let's open another terminal and install `stay-cli`:

```
npm install -g stay-cli
```

The `stay-cli` will prepare our module for decentralized publishing by injecting a `prepublish` script into our `package.json`. If we followed the previous section (Prepublish) we'll already have a `prepublish` script.

That being the case we'll need to alter the `scripts` section in our `package.json` file to:

```
  "scripts": {
    "check": "npm ls && npm test",
    "test": "npm run lint && tap --cov test",
    "lint": "standard"
  }
```

Notice that we've renamed the "prepublish" field to "check".

Next in our module's folder, we can run the following:

```
stay init
```

This will place a `publish-dep.sh` in our module's directory and alter our `package.json` file to look thusly:

```
  "scripts": {
    "check": "npm ls && npm test",
    "test": "npm run lint && tap --cov test",
    "lint": "standard",
    "prepublish": "./publish-dep.sh"
  },
```

Now let's make sure our pre-publish checks happen, by editing the `prepublish` field like so:

```
    "prepublish": "npm run check && ./publish-dep.sh"
```

Let's bump our module's patch version, since we've made edits:

```
npm version patch
```

Now we're ready to publish our module to the IPFS peer-to-peer network.

```
npm publish
```

This will publish to both npmjs.org and IPFS.

In the publish output we should see something like:

```
## Publishing dependency
Published as QmPqxGscbc6Qv9zdN3meifT7TRfBJKXT4VVRrQZ4skbFZ5
```

We can use the supplied hash to install our `hsl-to-hex` module from IPFS.

Let's try that out. First, let's create a new folder called stay-play:

```
mkdir stay-play
```

Now we'll initialize a new module:

```
npm init
```

Next, we need to add the decentralized dependency:

```
stay add hsl-to-hex@QmPqxGscbc6Qv9zdN3meifT7TRfBJKXT4VVRrQZ4skbFZ5
```

We can replace the hash (beginning `Qmpqx`) with the hash of our own published module when we ran `npm publish`.

This will add an `esDependencies` field to our `package.json` file, which looks like this:

```
  "esDependencies": {
    "hsl-to-hex": "QmPqxGscbc6Qv9zdN3meifT7TRfBJKXT4VVRrQZ4skbFZ5"
  }
```

Finally we can grab install distributed dependencies listed in the `esDepdendencies` field with:

```
stay install
```

In all likelihood this will install `hsl-to-hex` from our own system, since we're the closest network node to ourself. As an experiment, we could always try passing on the hash to a friend and seeing if they can install it.

See also

- *Scaffolding a module* in this chapter
- *Installing Dependencies* in this chapter
- *Writing module code* in this chapter
- *Using a private repository* in this chapter
- *Detecting dependency vulnerabilities* in **Chapter 8 Dealing with Security**

Using a private repository

There can be multiple reasons for using a private repository. From a personal perspective, it can be a useful caching mechanism or test-bed when run locally. From an organizational perspective it's usually about control.

Whilst open source has been fundamental to advancements in every industry that has been touched by the digital era, there's still a case for in-house only code. In some cases, it may be that code is specific to an organization, or reveals internal details that should be trade secrets. In other cases it may be an archaic though impassable proprietary culture. At any rate, when living in a gated community it makes all the more sense to share resources.

In this recipe we'll investigate setting up a personal module registry which could be deployed as an internal registry to provide a platform for code-reuse across an organization.

Getting ready

Setting up a private repository has become very easy, to get ready for this recipe simply have a terminal open and ready.

How to do it

We're going to use a tool called `sinopia`.

We install it with npm:

```
npm install -g sinopia
```

Next simply start `sinopia` as a service:

```
sinopia &> /dev/null &
```

Or if we're on Windows:

```
START /B "" sinopia >nul 2>nul
```

In order to publish to our local registry, we'll need to configure npm to point at Sinopia's HTTP endpoint.

We can do so with the following command:

```
npm set registry http://localhost:4873
```

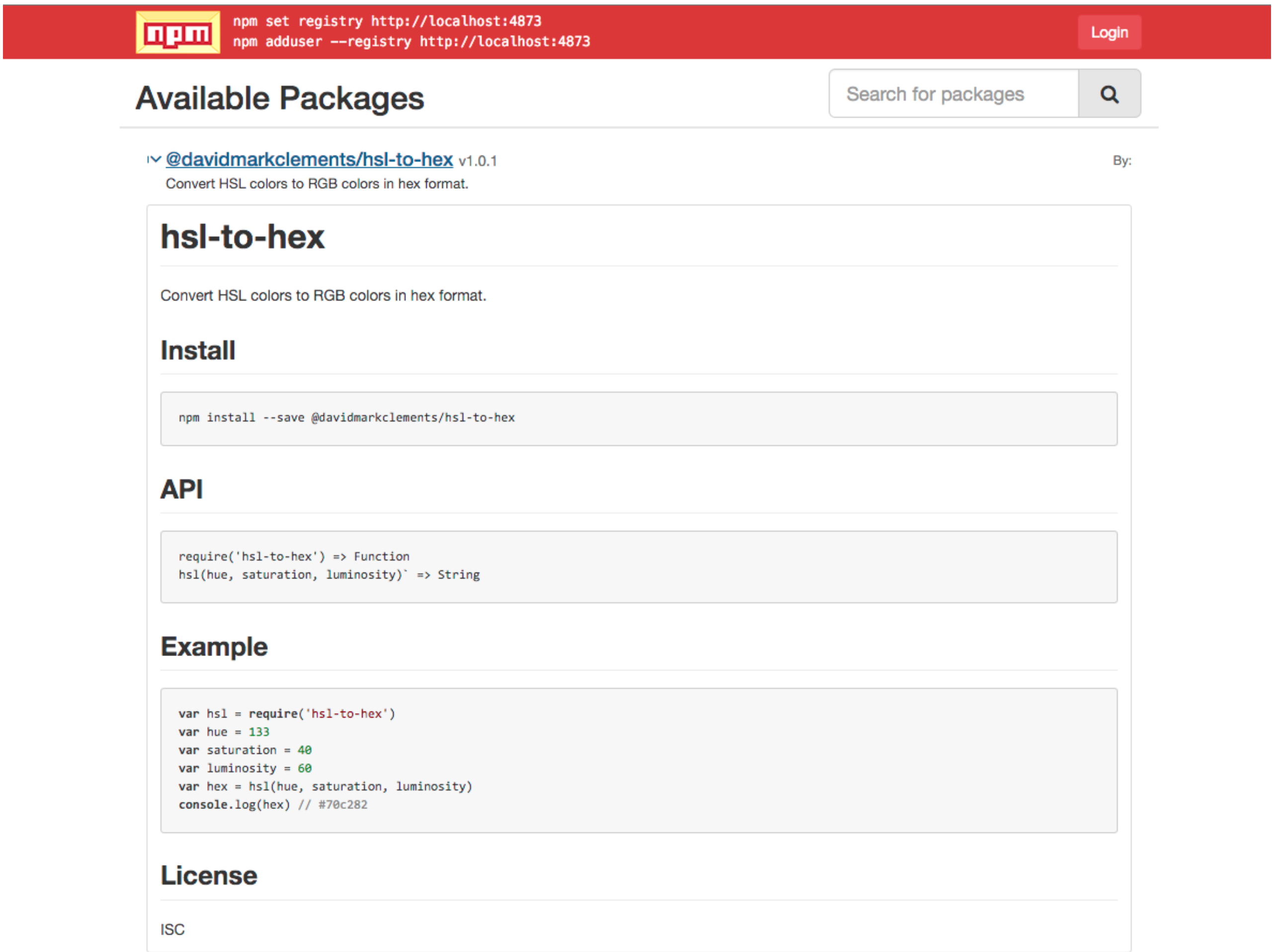
> ### Hosting Sinopia
>
> In order to host Sinopia either over the public internet, within cloud infrastructure on in-house metal several Sinopia settings would need to be configured. These settings are typically located `~/.config/sinopia/config.yaml`, although we can pass our own file with `sinopia my-config.yaml`. In particular, we would want to set an admin password hash, restrict access in the packages field and listen on host `0.0.0.0`. We can find out more about configuration options here https://github.com/rlidwka/sinopia/blob/master/conf/full.yaml. Additionally, there are ready made Chef, Puppet and Docker setups for easy deployment, see the Sinopia readme at http://npm.im/sinopia for more details.

Finally, let's enter the directory of the `hsl-to-hex` module we've been building throughout this chapter, and publish it locally.

From the `hsl-to-hex` module folder, we run:

```
npm publish
```

We can see if this worked by navigating to http://localhost:4873 in the browser.



Sinopia Module Viewer

Stop sinopia

We can stop Sinopia with `killall sinopia` or on Windows `taskkill /IM sinopia.cmd`

Revert to public registry

Set the npm registry back to default with the following:

```
npm config delete registry
```

How it works

The npm client simply sends HTTP requests to the npm registry.

For instance, an `npm publish` command causes a tarball of our module to be sent to the registry endpoint via an HTTP PUT request.

Sinopia supports a subset of the endpoints and verbs used by the npm registry, but beyond that it works quite differently.

The npm registry was originally built as a CouchDB application that held all meta-data and tarballs in one database, but now runs as a distributed microservice based system made up of many pieces.

Sinopia on the other hand, is a RESTful Node application which essentially stores modules to the file system (in exactly the same form as we create them, as a folder with `package.json` and source files).

> ### Sinopia storage
>
> We can try `grep storage ~/.config/sinopia/config.yaml` to find out where Sinopia stores modules on our file system. If we're using Windows we can use `findstr storage %homedrive%%homepath%/.config/sinopia/config.yaml`.

There's more

As we round off this chapter, we're going to explore other aspects of private repositories such as localized caching and associating scopes to registries.

Module caching

Sinopia also acts as a localized cache. If our registry endpoint is set to the Sinopia server we simply `npm install` a package and it will be downloaded from the npm registry then saved to sinopias storage. The next time we install the same module (and version), it will come from Sinopia.

If we're only interested in localized caching of npm modules, an interesting alternative to Sinopia is `registry-static`.

The `registry-static` tool is much simpler, it simply replicates the entire npm registry (around 300GB of data) to a flat file structure.

Scope Registries

We can associate registries to module namespaces. An example of a module namespace (or scope) is found in our module's name `@davidmarkclements/hsl-to-hex` where `@davidmarkclements` is the scope.

If we haven't already done so, we can revert to the default registry with the following command:

```
npm config delete registry
```

Let's say we want to associate only the `@ncb` scope with our local Sinopia server. We can run the following command

```
npm set @ncb:registry http://localhost:4873
```

Now let's alter the `name` field of `package.json` like so:

```
  "name": "@ncb/hsl-to-hex",
```

Now we can publish to our local registry:

```
npm publish
```

If we navigate to http://localhost:4873 we should see the `@ncb/hsl-to-hex` module listed.

See also

- *Publishing a module* in this chapter
- *Scaffolding a module* in this chapter
- *Installing Dependencies* in this chapter
- *Writing module code* in this chapter