NPM Modules

Overview

This document will cover some basic steps to get setup using git and node modules. You'll be working primarily on the command line. If this is new to you, sorry about the additional cognitive load, but I'd recommend diving in instead of looking for GUI alternatives (Confucius say: Those who seek GUI alternatives meet sticky ends). Feel free to ask on the #js-arch Slack channel if you need help with anything.

Setup

On your mac, uninstall any version of node you currently have installed, and instead use nvm (Node Version Manager). Instructions are on the GitHub page. Then use it to install a recent version of node, such as the 6.2.0 branch (`nvm install v6.2.0; nvm use 6.2.0`).

Using a virtual machine (vagrant), I don't recommend using nvm. Instead, install node at the version you want during your provisioning step using the steps outlined at https://github.com/nodesource/distributions.

Create a Module

1. Create a new empty GitHub repo `my-module` (calling it something other than my-module).
2. On your local filesystem `mkdir my-module && cd my-module && touch README.md.`
3. `git init && git add . && git commit -m "initial commit"`
4. `git remote add origin https://github.com/spoonflower/my-module && git remote -v && git push -u origin master`.
5. Create and commit a `.gitignore` file with something like

```
node_modules/
.vagrant/
.env
.DS_Store
```

Run `npm init`. This will walk you through a wizard on the command line to set up your `package.json` file. When prompted to enter an entry point, enter `todo.js`. You can use defaults for the rest of the items if you'd like, or edit them as you see fit.

One thing that I like to do that helps prevent you from accidentally publishing your package on npm is to add `"private": true` to the json.

Exploring Node Modules

Now that you've got a new repo setup, let's play around with npm a little bit. We're going to make a toy todo library. We'll just make the data handling component for now, which will work in node or the browser. Remember, although node is known for running as a server, it's perfectly adequate at doing scripting or running tests. We'll write in ES5 to keep things easily compatible with the browser.

Installing Third Party Modules

To generate unique ids for our todos, let's get a uuid module installed. Most open source modules are published in the npm library. If you haven't used an npm module before, it's good practice to check out its GitHub page, so we can have a look at it's package.json file, and also to see when it was last updated to make sure it has some activity.

Visit https://github.com/defunctzombie/node-uuid and open the package.json file. It's a good idea to check out the dependencies section. When you're thinking about the browser, and maintainence in general, extra dependencies can add weight and make upgrading down the road more difficult, so it's important to just be cognizant of what you're getting into by installing the library. A uuid library is simple enough to not need dependencies, and you'll see that it does in fact contain no "dependencies" entry. If you'd like to contrast with a more complicated module, look at the package.json for the Express Framework

Another place to check is the `"name"` key. This will be what you use to install the module. We can see that the name for the node-uuid module is 'uuid'. In your `my-module` folder, type in `npm install --save uuid`.

After the installation is complete, in your editor or file browser, open up the `node_modules/uuid` folder. You should see pretty much the same files as the GitHub page.

Also, reload your local package.json, and check the `"dependencies"` key. You should see that a new entry has been added for it, with the key name `"uuid"`. This happened because we specified the `--save` argument for the install command, which updates your local package.json automatically. If you are installing a package just for development use, such as a test suite, you can use `--save-dev` to put the entry in the `"devDependencies"` key. This will mark the module as being unnecessary for production installs.

You've just installed an external module. In node, there are also internal modules, that use the same system, but live locally within your module. To specify you are requiring an external module, don't use an absolute or relative path when requiring. `require('uuid')` loads the uuid module from your `node_modules` folder. `require('./lib/uuid')` looks for `./lib/uuid.js`, relative to the path of the script it's called in. We'll explore this more in a bit.

Folder Structure & Import/Export

For external modules, you can specify the name of the file you'd like to use as the primary entry point using the `"main"` key in package.json. Look at the uuid package for an example. You'll see it refers to 'uuid.js'. This means that after you have installed this module, when you do something like `var foo = require('uuid')`, `foo` will refer to whatever is 'exported' by the `uuid.js` file. Open up the `uuid.js` file in GitHub and see if you can predict what properties `foo` would have in the above example (hint, it's at the bottom).

In this case, it may be a little confusing if you aren't aware that functions are also objects in js. We see that the variable `uuid` is assigned to the function `v4`. It then has properties assigned to other functions, such as `v1`, `v4`, `parse`, and `unparse`. Finally, the `uuid` var is exported in the node fashion, by using `module.exports = uuid`.

Create a new file called `todo.js`, and put the following in it.

```
var uuid = require('uuid');
console.log('v4', uuid.v4());
console.log('v1', uuid.v1());
```

Now let's run it using node and see what we get. Enter `node todo.js` in your console, and you should get output that looks like

```
v4 2330c912-8cf6-430e-bd7b-4c61879e81ea
v1 8e1e1480-5e41-11e6-8d3d-ffcfb964ea53
```

Next, let's create an internal module.

1. Run `mkdir lib && echo "module.exports = 'Hello!';" > lib/hello.js`.
2. Add `console.log('Hello?', require('./lib/hello'))` to your `todo.js` file and save.
3. Run `node todo.js` again. You should see `Hello? Hello!` added to the output.

Hopefully this also helps demonstrate that you can export any type of variable in `module.exports`. Whatever you export is what will be pulled in when you do a `require`.