
scripty



What is?

Using npm-scripts has become a popular way of maintaining the various build tasks needed to develop Node.js modules. People like npm-scripts because it's simple! This is a common refrain:

Don't bother with grunt, gulp, or broccoli, just add a little script to your package.json and run it with `npm run name:of:script`

Indeed, this *is* much simpler, but it can quickly become a mess. Take a look at what happened to our testdouble.js library's package.json. Using npm-scripts for everything is simple to start with, but it can't hope to guard against the complexity that naturally accumulates over the life of a project.

We wrote scripty to help us extract our npm scripts—particularly the gnarly ones—into their own files without changing the command we use to run them. To see how to do this yourself, read on!

Install

```
1 $ npm install --save-dev scripty
```

Usage

1. From your module's root, create a `scripts` directory
2. If you want to define an npm script named "foo:bar", write an executable file at `scripts/foo/bar`
3. Feel a liberating breeze roll over your knuckles as your script is free to roam within its own file, beyond the stuffy confines of a quote-escaped string inside a pile of JSON
4. Declare your "foo:bar" script in "scripts" in your `package.json`:

```
1 "scripts": {  
2   "foo:bar": "scripty"  
3 }
```

From this point on, you can run `npm run foo:bar` and scripty will use npm's built-in `npm_lifecycle_event` environment variable to look up `scripts/foo/bar` and execute it for you.

This pattern is great for extracting scripts that are starting to become unwieldy inside your **package.json**, while still explicitly calling out the scripts that your package supports (though where to take that aspect from here is up for debate).

Advanced Usage

Ready to take things to the next level? Check this stuff out:

Passing command-line args

To pass command-line args when you're running an npm script, set them after `--` and npm will forward them to your script (and scripty will do its part by forwarding them along).

For example, if you had a script in `scripts/echo/hello`:

```
1 #!/usr/bin/env sh
2
3 echo Hello, "$1"!
```

Then you can run `npm run echo:hello -- WORLD` and see your script print `"Hello, WORLD!"`.

Batching “sub-scripts”

Let's say you have two test tasks in `scripts/test/unit` and `scripts/test/integration`:

```
1 "scripts": {
2   "test:unit": "scripty",
3   "test:integration": "scripty"
4 }
```

And you want `npm test` to simply run all of them, regardless of order. In that case, just add a `"test"` entry to your **package.json** like so:

```
1 "scripts": {
2   "test:unit": "scripty",
3   "test:integration": "scripty",
4   "test": "scripty"
5 }
```

And from then on, running `npm test` will result in scripty running all the executable files it can find in `scripts/test/*`.

Defining an explicit parent script

Suppose in the example above, it becomes important for us to run our scripts in a particular order. Or, perhaps, when running `npm test` we need to do some other custom scripting as well. Fear, not!

Without changing the JSON from the previous example:

```
1 "scripts": {
2   "test:unit": "scripty",
3   "test:integration": "scripty",
4   "test": "scripty"
5 }
```

Defining a script named `scripts/test/index` will cause scripty to only run that `index` script, as opposed to globbing for all the scripts it finds in `scripts/test/*`.

Running scripts in parallel

If you have a certain command that will match multiple child scripts (for instance, if `npm run watch` matches `scripts/watch/js` and `scripts/watch/css`), then you can tell scripty to run the sub-scripts in parallel by setting a `SCRIPTY_PARALLEL` env variable to `'true'`. This may be used to similar effect as the `npm-run-all` module.

To illustrate, to run a scripty script in parallel, you might:

```
1 $ SCRIPTY_PARALLEL=true npm run watch
```

Or, if that particular script should always be run in parallel, you can set the variable in your `package.json`:

```
1 "scripts": {
2   "watch": "SCRIPTY_PARALLEL=true scripty"
3 }
```

Which will run any sub-scripts in parallel whenever you run `npm run watch`.

Finally, if you **always** want to run scripts in parallel, any option can be set in your `package.json` under a `"scripty"` entry:

```
1 "config": {
2   "scripty": {
3     "parallel": true
4   }
5 }
```

Windows support

Windows support is provided by scripty in two ways:

1. If everything in your `scripts` directory can be safely executed by Windows, no action is needed (this is only likely if you don't have collaborators on Unix-like platforms)
2. If your project needs to run scripts in both Windows & Unix, then you may define a `scripts-win/` directory with a symmetrical set of scripts to whatever Unix scripts might be found in `scripts/`

To illustrate the above, suppose you have this bash script configured as `"test/unit"` in your package.json file and this bash script defined in `scripts/test/unit`:

```
1 #!/usr/bin/env bash
2
3 teenytest --helper test/unit-helper.js "lib/**/*.test.js"
```

In order to add Windows support, you could define `scripts-win/test/unit.cmd` with this script:

```
1 @ECHO OFF
2
3 teenytest --helper test\unit-helper.js "lib\**\*.test.js"
```

With a configuration like the above, if `npm run test:unit` is run from a Unix platform, the initial bash script in `scripts/` will run. If the same CLI command is run from Windows, however, the batch script in `scripts-win/` will be run.

Specifying custom script directories

By default, scripty will search for scripts in `scripts/` relative to your module root (and if you're running windows, it'll check `scripts-win/` first). If you'd like to customize the base directories scripty uses to search for your scripts, add a `"scripty"` object property to your package.json like so:

```
1 "config": {
2   "scripty": {
3     "path": "../core/scripts",
4     "windowsPath": "../core/scripts-win"
5   }
6 }
```

You can configure either or both of `"path"` and `"windowsPath"` to custom locations of your choosing. This may be handy in situations where multiple projects share the same set of scripts.

Sharing scripts via node modules

You can configure scripty to include certain node modules into its executable search space. This is beneficial if you would like to create a centralized place for your scripts and then share them across multiple projects. To include modules add a "scripty" object property, `modules`, to your `package.json` like so:

```
1 "config": {
2   "scripty": {
3     "modules": ["packageA", "packageB"]
4   }
5 }
```

Each node module must contain a `scripts` directory. Below is an example directory structure:

```
1 root/
2   scripts/
3     foo
4   node_modules/
5     packageA/
6       scripts/
7         foo
8         bar
9     packageB/
10      scripts/
11        bar
12        baz
```

In the above example the resolution of `foo` would resolve to `root.scripts.foo`. Local scripts take priority over ones defined in modules. The resolution of `bar` would resolve to `root.node_modules.packageA.scripts.bar` as `packageA` was the first module defined in the `scripty.modules` config.

Dry runs

To perform a dry run of your scripts—something that's handy to check which scripts will run from a particular command without actually executing potentially destructive scripts, you can set an environment variable like so:

```
1 $ SCRIPTY_DRY_RUN=true npm run publish:danger:stuff
```

This will print the path and contents of each script the command would execute in the order they would be executed if you were to run the command normally.

Worth mentioning, like all options this can be set in `package.json` under a "scripty" entry:

```
1 "config": {
2   "scripty": {
3     "dryRun": true
4   }
5 }
```

Log output

Scripty is now quieter by default. The output can be configured to a level of `verbose`, `info`, `warn`, or `error`. Any logs equal to or higher than the setting are shown. All logs are printed to STDERR (to aid in redirection and piping).

```
1 $ SCRIPTY_LOG_LEVEL=verbose npm run publish:danger:stuff
```

This will print the path and contents of each script the command executes.

If you always want scripty to run your scripts at a certain level, you can set it in your package.json under a `"scripty"` entry:

```
1 "config": {
2   "scripty": {
3     "logLevel": "warn"
4   }
5 }
```

`SCRIPTY_SILENT` and `SCRIPTY_QUIET` are aliases for `SCRIPTY_LOG_LEVEL=silent`
`SCRIPTY_VERBOSE` is an alias for `SCRIPTY_LOG_LEVEL=verbose` (also `"silent": true`, etc in package.json#scripty)

`SCRIPTY_DRY_RUN=true` implies log level `info`

Explicit setting from `logLevel` takes precedence; otherwise, conflicting values between `silent/verbose/dryRun` will respect the highest level. If no setting is provided, scripty will infer its log level from npm's log level.

Likely questions

- **Is this pure magic?** - Nope! For once, instilling some convention didn't require any clever metaprogramming, just environment variables npm already sets; try running `printenv` from a script some time!
- **Why isn't my script executing?** - If your script isn't executing, make sure it's **executable**! In UNIX, this can be accomplished by running `chmod +x scripts/path/to/my/script` (permissions will also be stored in git)

-
- **How can I expect my users to understand what this does?** Documenting your project's use of `scripty` in the `README` is probably a good idea. Here's some copy pasta if you don't feel like writing it up yourself:

npm scripts

MyProject uses `scripty` to organize npm scripts. The scripts are defined in the `scripts` directory. In `package.json` you'll see the word `scripty` as opposed to the script content you'd expect. For more info, see `scripty`'s GitHub.

{{ insert table containing script names and what they do, e.g. this }}

Code of Conduct

This project follows Test Double's code of conduct for all community interactions, including (but not limited to) one-on-one communications, public posts/comments, code reviews, pull requests, and GitHub issues. If violations occur, Test Double will take any action they deem appropriate for the infraction, up to and including blocking a user from the organization's repositories.