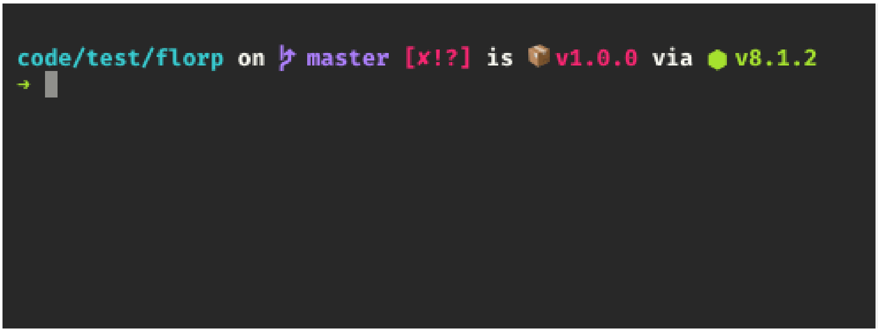
Introducing npx: an npm package runner

Those of you upgrading npm to its latest version, `npm@5.2.0`, might notice that it installs a new binary alongside the usual `npm`: `npx`.

npx is a tool intended to help round out the experience of using packages from the npm registry — the same way npm makes it super easy to install and manage dependencies hosted on the registry, npx makes it easy to use CLI tools and other executables hosted on the registry. It greatly simplifies a number of things that, until now, required a bit of ceremony to do with plain npm:

Using locally-installed tools without `npm run-script`



```
code/test/florp on master [x!?] is v1.0.0 via ● v8.1.2
➜
```

Installing cowsay as a local devDependency and running it with `$ npx cowsay`

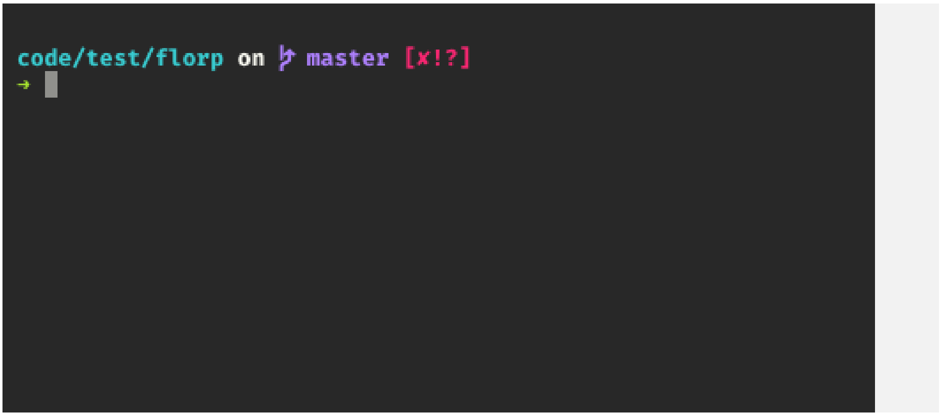
For the past couple of years, the npm ecosystem has been moving more and more towards installing tools as project-local `devDependencies`, instead of requiring users to install them globally. This means that tools like `mocha`, `grunt`, and `bower`, which were once primarily installed globally on a system, can now have their versions managed on a per-project basis. It also means that all you need to do to get an npm-based project up and running is to make sure you have node+npm on your system, clone the git repo, and do `npm it` to run `install` and `test`. Since `npm run-script` adds local binaries to path, this works just fine!

The downside is that this gives you no fast/convenient way to invoke local binaries interactively. There’s several ways to do this, and they all have some annoyance to them: you can add those tools to your `scripts`, but then you need to remember to pass arguments through using `--`, you can do shell tricks like `alias npmx=PATH=$(npm bin):$PATH`, or you can just path manually to them with `./node_modules/.bin/mocha`. These all work, but none are quite ideal.

npx gives you what I think is the best solution: `$ npx mocha` is all you need to do to use your **local** installation. If you go an extra step and configure the shell auto-fallback (more on this below), then `$ mocha` inside a project directory will do the trick for you!

For bonus points, npx has basically no overhead if invoking an already-installed binary — it's clever enough to load the code for the tool directly into the current running `node` process! This is about as fast as this sort of thing gets, and makes it a perfectly acceptable tool for scripting.

Executing one-off commands



```
code/test/florp on ᚠ master [x!?]
→
```

`$ npx create-react-app my-cool-new-app` installs a temporary create-react-app and calls it, without polluting global installs or requiring more than one step!

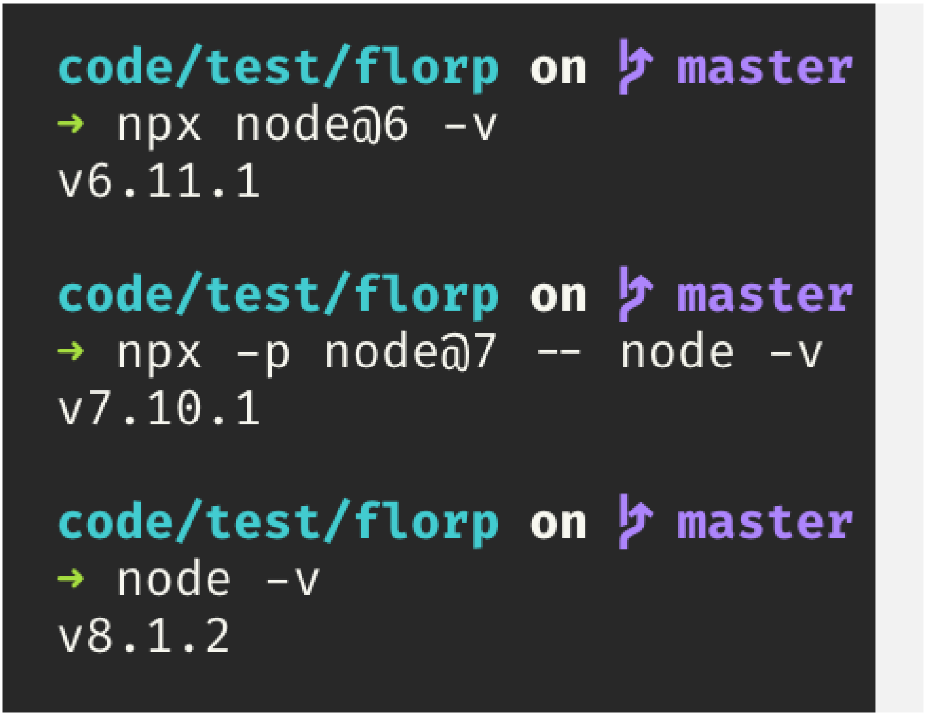
Have you ever run into a situation where you want to try some CLI tool, but it's annoying to have to install a global just to run it once? npx is great for that, too. Calling `npx <command>` when `<command>` isn't already in your `$PATH` will automatically install a package with that name from the npm registry for you, and invoke it. When it's done, the installed package won't be anywhere in your globals, so you won't have to worry about pollution in the long-term.

This feature is ideal for things like generators, too. Tools like `yeoman` or `create-react-app` only ever get called once in a blue moon. By the time you run them again, they'll already be far out of date, so you end up having to run an install every time you want to use them anyway.

As a tool maintainer, I like this feature a lot because it means I can just put `$ npx my-tool` into the `README.md` instructions, instead of trying to get people over the hurdle of actually installing it. To be frank, saying "oh just copy-paste this one commands, it's zero commitment" is more palatable to users who are unsure about whether to use a tool or not.

Here's some other fun packages that you might want to try using with `npx`: `happy-birthday`, `benny-hill`, `workin-hard`, `cowsay`, `yo`, `create-react-app`, `npm-check`. There's even an entire `awesome-npx` repo! Go ahead! A command to get a full-fledged local REST server runningis small enough to fit in a tweet.

Run commands with different Node.js versions



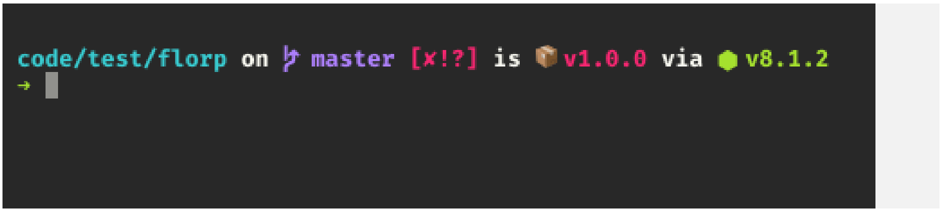
```
code/test/florp on master
➜ npx node@6 -v
v6.11.1

code/test/florp on master
➜ npx -p node@7 -- node -v
v7.10.1

code/test/florp on master
➜ node -v
v8.1.2
```

`npx -p node@<version> node -v` can be used to do one-off runs of node versions.

As it turns out, there's this cool package by Aria Stewartcalled `node` on the npm registry. This means that you can very easily try out node commands using different node versions, without having to use a version manager like `nvm`, `nave`, or `n`. All you need is a stock `npm@5.2.0`installation!

The `-p` option for npx allows you to specify packages to install and add to the running `$PATH`, so it means you can do fun things such as: `$ npx -p node@6 npm it` to install and test your current npm package as if you were running `node@6` globally. I use this all the time myself — and even recently had to use it a lot with one project, due to one of my testing libraries breaking under `node@8`. It's been a real life-saver, and I've found it much easier to use for this sort of use-case than version managers, which I always somehow find a way to break or misconfigure.

Developing `npm run-script`s interactively



```
code/test/florp on  master [x!?] is 📦 v1.0.0 via ⬢ v8.1.2
➜ ▌
```

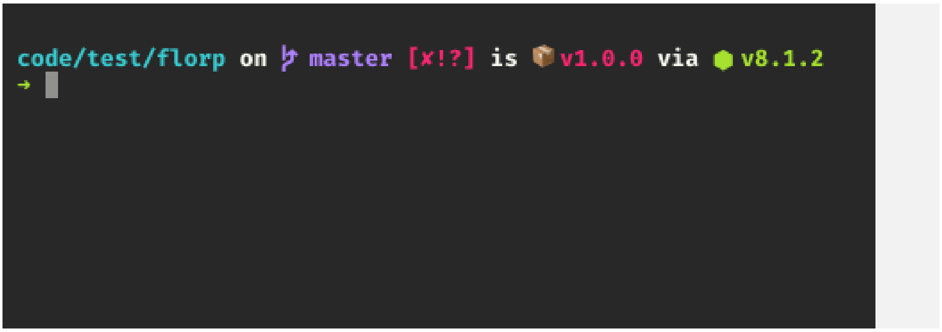
`$ npx -p cowsay -p lolcatjs -c 'echo "$npm_package_name@$npm_package_version" | cowsay | lolcatjs'` installs both cowsay and lolcatjs, and gives the script access to a slew of `$npm_` variables from run scripts.

A lot of npm users these days take advantage of the really cool `run-script` feature. Not only do they arrange your `$PATH` such that local binaries are accessible, but they also add a whole slew of environment variables that you can access in those scripts! You can see what these extra variables are with `$ npm run env | grep npm_`.

This can make it tricky to develop and test out run scripts — and it means that even with tricks like `$(npm bin)/some-bin`, you still won't have access to those magical env vars while working interactively.

But wait! npx has yet another trick up its sleeve: when you use the `-c` option, the script written inside the string argument will have full access to the same env variables as a regular run script! You can even use pipes and multiple commands with a single `npx` invocation!

Share gist-based scripts with friends and loved ones!



```
code/test/florp on  master [x!?] is 📦v1.0.0 via ⬢ v8.1.2
➜ 
```

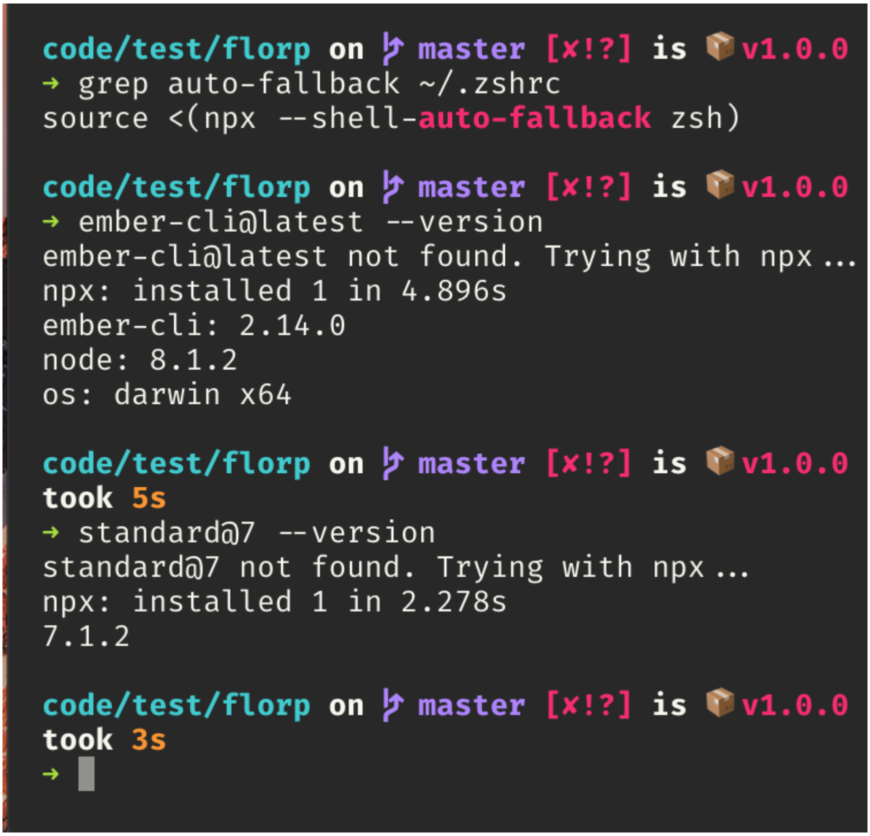
It's become pretty common to use gist.github.com to share all sorts of utility scripts, instead of setting up entire git repos, releasing new tools, etc.

With npx, you can take it a step further: since npx accepts any specifier that npm itself does, you can create a gist that people can invoke directly, with a single command!

Try it out yourself with https://gist.github.com/zkat/4bc19503fe9e9309e2bfaa2c58074d32!

Note: Stay safe out there! Always make sure to read through gists when executing them like this, much like you would when running `.sh` scripts!

Bonus Round: shell auto-fallback



```
code/test/florp on ⎇ master [✘!?] is 📦v1.0.0
➜ grep auto-fallback ~/.zshrc
source <(npx --shell-auto-fallback zsh)

code/test/florp on ⎇ master [✘!?] is 📦v1.0.0
➜ ember-cli@latest --version
ember-cli@latest not found. Trying with npx...
npx: installed 1 in 4.896s
ember-cli: 2.14.0
node: 8.1.2
os: darwin x64

code/test/florp on ⎇ master [✘!?] is 📦v1.0.0
took 5s
➜ standard@7 --version
standard@7 not found. Trying with npx...
npx: installed 1 in 2.278s
7.1.2

code/test/florp on ⎇ master [✘!?] is 📦v1.0.0
took 3s
➜
```

Putting the npx auto fallback in .zshrc means you can do `$ ember-cli@latest …` without referencing npx at all!

This awesome feature, added by Félix Saparelli, means that for many of these use cases, you never even need to call `npx` directly! The main difference between regular npx usage and the fallback is that the fallback doesn't install new packages unless you use the `pkg@version` syntax: a safety net against potentially-dangerous typosquatting.

Setting up the auto-fallback is straightforward: look in the npx documentation for the command to use for your current shell, add it to `.bashrc` / `.zshrc` / `.fishrc`, then restart your shell (or use `source` or some other mechanism to refresh the shell).

Now, you can do things like `$ standard@8 --version` to try out different versions of things, and if you're inside an npm project, `$ mocha` will automatically fall back to the locally-installed version of mocha, provided it's not already installed globally.

Do It Live!

You can get npx **now** by installing `npm@5.2.0` or later — or, if you don't want to use npm, you can install the standalone version of npx! It's totally compatible with other package managers, since any npm usage is only done for internal operations. Oh, and it's available in **10 different languages**, thanks to contributions by a bunch of early adopters from all over the world, with `--help` and all system messages translated and automatically available based on system locale! There's also an `awesome-npx` repo with examples of things that work great with npx!