

Project Layout

I'm a polyglot programmer. I work in a variety of languages but mostly in C#, Typescript, and Rust. Every few years, I try a new language to see if I can pick up new ideas or if one "fits" my current mental state better. This is also why I've done a lot dozens of other languages; I would say I know over thirty languages but I'm only a "master" in a handful.

I also flit from project to project. I have my writing and games. I have little one-off programs and ones that I hope will become a major thing. But, like everything else in my life, I'm "gloriously unfocused" on my tasks which means I have to minimize the speed that I get into a project before the muse escapes me.

Tools Selection

One of the earliest approaches I had to try getting a proper environment at the per-project level was asdf. It worked out fairly well for a few years, but then I noticed that my various novels and stories were getting fragile. There were limitations that asdf couldn't handle easily which meant I needed something more reliable. That led me into Nix which is my current setup because entering the directory sets up that project's settings while still giving me the reproducibility I need for my novels.

D. Moonfire

This means that most of my projects now have a `./flake.nix` and a `./flake.lock` in the root level.

Building, Releasing, and Actions

Because I've fallen in love with Semantic Releases and Conventional Commits, a lot of my processes are built around those. In earlier projects, that usually meant that almost every project also included Node in some form so I could use semantic-release. That also meant I could use `package.json` to handle versioning.

Though, recent thoughts have suggested that I need to break that "one tool fits all" approach. Mostly it came while working on Nitride and this website. I found myself trying to have "one build system" to create everything related to the site, including handling Javascript and CSS/SASS. Those are two very complicated projects in C#, so I realize it made sense that instead of creating a Nitride task to call webpack, I really should just call `webpack` directly. In other words, the Unix philosophy.

This is where being a polyglot and using different tools comes into play. I have a website that does C#, Typescript, and SASS at the same time. Which one is the "root", which command drives everything? What about a Rust project? Or something else?

Shell Scripts

That has kind of led me to my current approach. Instead of always packaging Node in my projects, I really should have a standard location to handle the various actions/targets that apply to any project. Right now, that seems to be shell scripts.

With shell scripts, I just have to know that `./scripts/build.sh` will do whatever is needed to build the target. Same with `./scripts/test.sh` and `./scripts/release.sh`. A Rust project may call Cargo, a .NET project will call `dotnet`, and polyglot will call any and all needed to build it.

This will give me room to experiment. If I decide I want to play with Cake for my .NET projects, then it still works out because it is just a consistent place. If I want to use Lefthook instead of Husky, I can.

I also went with `.sh` suffixes on the files because while I mostly code in Linux, I also want to support Powershell and Windows. That way, it is also clear that `build.sh` and `build.ps1` probably result in the same end-result, but specific for that language. (I know Powershell runs on Linux too.)

Obviously, some documentation would be required, but that could be a `README.md` file in that directory. That will look nice in GitLab and give documentation.

Paths

Fortunately, I use direnv and nix-direnv frequently in my development. This loads the flake.nix file as soon as I enter the directory and sets up the tools I need. It also gives me a chance to modify the PATH variable but only for that directory which means I can add the ./scripts folder into the path and have it available anywhere inside the project.

```
export PATH=$PWD/scripts:$PATH
use flake || use nix
```

When working with packaging systems such as Node that also include scripts, I also add those into the path. In both cases, \$PWD is always the directory with the .envrc file, even if I change directly into somewhere deeper into the project tree; using \$PWD/scripts means that the build.sh command is available anywhere.

```
export PATH=$PWD/scripts:$PWD/node_modules/.bin:$PATH
use flake || use nix
```

Boilerplates

Over the half year or so that I've been using this, I found that I was introducing a few new patterns into my scripts. Mostly these were to support CI/CD environments but also because I like feedback that scripts are doing something.

The most notable aspects were to almost always move into the root directory of the project.

```
#!/usr/bin/env bash
cd $(dirname $(dirname $0))
```

In the above case, \$0 is the name of the script. The first dirname gets me into the ./scripts folder, the second gets me into the root. That means that even if I call this script from deep inside the project, the paths are always relative to the project root.

The other is to set up logging so I have something to see what is going on. This is useful for the CI process, but also just so I know something is working properly. I ended up using a consistent start to the scripts to help me identify where the build process was.

```
log() { echo "[INFO] $(basename $0): $@"; }
log "running tests/grepping/going/running-tests.sh"
some testing code
```

D. Moonfire

When run, it looks like this:

```
$ test.sh
test.sh: running tests/gre&#173;go&#173;ri&#173;an-tests.scm
.....
-----
Ran 38 tests in 0.001s

OK
$
```

Each script usually has their only Unicode character, which also gives the logs a nice colorful appearance and really makes it easier to see where things are going. I ended up using a Bash function for this because it simplified the calls into a simple log mes­sage and made it easier to function.

Sadly, Bash doesn't have a good packaging story, so I just copy/paste this into the top of every script along with the `#!/usr/bin/env bash` shebang. Overall, it seems to work and I've been pretty happy with it since.