Flakes and Developer Environments (Nix From First Principles: Flake Edition #8)

Nov 07
2022

#Nix

This is part 8 of the Nix from First Principles: Flake Edition series.

While packages are the most common type of item you'll find contained in flakes, another type of item is a *development shell*. This is an environment that you can enter with the `nix develop` command. By default, each package will provide a development shell which is the build environment used to build it. By running just `nix develop` without specifying which shell is wanted, `nix` will put you in a `bash` shell with the environment from the default package.

However, sometimes you may wish to set up additional tools in a development environment, for example you may have scripts written in Python and want to have Python available as for a developer to use, yet you do not want to include Python as a dependency in your built package. Or you might want to set some environment variables so that logging is set to debug mode in your development environment.

You can customise the `nix develop` environment with the `devShells` attribute set in a flake. The helper function `pkgs.mkShell` is useful here. This is an extended version of the `mkDerivation` function that has been used in previous examples, except this version is specialised for shells. The following flake shows an example:

flake.nix

```
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";

  outputs = { self, nixpkgs }:
    let
      pkgs = import nixpkgs { system = "x86_64-linux"; };
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
        packages = [
          # Made available on the CLI
          pkgs.cargo
          pkgs.rustc
          pkgs.python3
        ];
        RUST_LOG = 1; # Set as environment variable
      };

      # You could also define extra shells or packages here
```

```
    };
}
```

Now if you run `nix develop` in the same directory as this flake, you'll be left at a bash shell with cargo, python3 and rustc available. If you run the command `env | grep RUST`, you can observe the following output:

```
$ env | grep RUST
RUST_LOG=1
```

Or `which python3`:

```
/nix/store/xcaaly5shfy227ffs8nipxrd49b56iqq-python3-3.10.8/bin/pyth
```

You could even define multiple shells. For example, lets say you wanted to support Python 3.8 through 3.10 in a single project. `nixpkgs` provides python artifacts in specific names like `python38` if you want specific versions. So you could build a flake which defines your python versions for each one, like so:

```
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";

  outputs = { self, nixpkgs }:
    let
      pkgs = import nixpkgs { system = "x86_64-linux"; };
      # A list of shell names and their Python versions
      pythonVersions = {
        python38 = pkgs.python38;
        python39 = pkgs.python39;
        python310 = pkgs.python310;
        default = pkgs.python310;
      };
      # A function to make a shell with a python version
      makePythonShell = shellName: pythonPackage: pkgs.mkShell {
        # You could add extra packages you need here too
        packages = [ pythonPackage ];
        # You can also add commands that run on shell startup with
        shellHook = ''
          echo "Now entering ${shellName} environment."
        '';
      };
    in
    {
      # mapAttrs runs the given function (makePythonShell) against
```

```
        # in the attribute set (pythonVersions) and returns a new set
        devShells.x86_64-linux = builtins.mapAttrs makePythonShell py
      };
}
```

Now if you run the command `nix develop .#python38`, you will end up in a shell that contains Python 3.8, something you can confirm with `python --version`.

```
$ nix develop .#python38
Now entering python38 environment.
$ python --version
Python 3.8.15
```

You can also inspect all items that are available in the flake with the command `nix flake show`:

```
$ nix flake show
git+file:///home/tony/code/nix-guide?dir=8-dev-envs%2fmanypython
└───devShells
    └───x86_64-linux
        ├───default: development environment 'nix-shell'
        ├───python310: development environment 'nix-shell'
        ├───python38: development environment 'nix-shell'
        └───python39: development environment 'nix-shell'
```

But what if I don't want bash?

Of course, you might prefer other shells than `bash` - however, Nix seems hardcoded to use `bash`. So you might wonder, how can you override the shell? For example, you might prefer the `zsh` or `fish` shells.

The first method of doing this requires no external tools, just `Nix` itself. You install your desired shell in the your `devShell`, and execute it in the `shellHook`. Here's an example that starts the fish shell.

```
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";

  outputs = { self, nixpkgs }:
    let
      pkgs = import nixpkgs { system = "x86_64-linux"; };
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
```

```
          packages = [
            pkgs.fish
          ];
          # Note that `shellHook` still uses bash syntax.
          # This starts fish, then exists the bash shell when fish ex
          shellHook = ''
            fish && exit
          '';
        };
      };
  }
```

While the big advantage here is that you don't need any tools apart from `nix`, one big disadvantage is that your shell of choice needs to be included in each development environment, and other users of your flakes are forced to use your preferred shell also.

Another option is to run `nix develop -c fish`. This overrides the command on shell launch to launch your preferred shell. (Thanks to u/Mysteriox7 on reddit for bringing this to my attention). This is an improvement over forcing all users of your flake to use your preferred shell, but does need to be typed each time.

With both approaches however, one thing that can be a pain is they require typing that command every time you enter a project which you want to use `nix develop` for.

Luckily, there are third party tools to the rescue.

nix-direnv

`nix-direnv` is a integration between `nix` and the `direnv` tool. `direnv` is a command to run commands whenever you enter a directory with a `.envrc` file. `nix-direnv` builds upon this to load a cached `nix` environment into your currently open shell. Effectively, the first time you open a project with `nix-direnv`, it will build your development environment, record the environment changes this process makes then apply those to your current shell. On opening the project again in the future, it will load the cached environment, so it doesn't need to execute your nix scripts again if they are unchanged,

To use this, you'll need to install direnv and nix-direnv. Follow their installation instructions at the links provided. Then to set up your flake project with direnv, you just need to create a `.envrc` file with one command, `use flake`

.envrc

```
use flake
```

Alternatively, you can pass it a flake reference if you want to use a different devShell. For example, my hosted version of the Python 3.8 development shell can be used with

```
use flake "gitlab:tonyfinn/nix-guide?dir=8-dev-envs/manypython#pyth
```

That's it for this part of the Nix guide. Next time I'll cover another type of flake output - runnable commands.