How I use nix

Yann Esposito

[2020-06-14 Sun]

Have you ever written a small script and you update your system and this stop working? Have you copied your tool/script to another machine it doesn't work because some dependency is missing? Have you tried to sync your dotfiles to another env and there are a few details not working? Some missing dependency? If the answer is yes, then nix can help.

Scripts

Suppose you want to write a portable script. For example, the script I use to minify my CSS. Here it is:

```
#!/usr/bin/env nix-shell
#!nix-shell --pure
#!nix-shell -i bash
#!nix-shell -I nixpkgs="https://github.com/NixOS/nixpkgs/archive/19.09.tar.gz"
#!nix-shell -p bash minify

minify "$1" > "$2"
```

So let's analyze each line of the header block:

`#!/usr/bin/env nix-shell` basic, use `nix-shell` to run the script.

`#!nix-shell --pure` only use dependencies installed in this nix shell environment. A bit as if the PATH environment variable was emptied.

`#!nix-shell -i bash` tell `nix-shell` to run `bash`

`#!nix-shell -I nixpkgs="https://github.com/NixOS/nixpkgs/archive/19.09.tar.gz"` pin the nixpkgs using this archive.

`#!nix-shell -p bash minify` install `bash` and `minify` in the nix shell.

Now if the script is run on a machine with `nix` installed you can be pretty sure it will work as expected. Even if I update my OS and I forget about this script for a few years. As long as I can install nix on the new system and I could download the tar file the script will be run the same way as the day I wrote it.

Remark: You can use any shell (like `fish`, `zsh`) but also other languages `python`, `haskell`, etc...

Temporary working env

Quite often, I need to do something, and run a specific command that need me to install a very specific command. And I'm pretty sure I will not use this tool ever again.

For those cases, what I do, is generally run my command directly with a fresh `nix-shell`.

```
> nix-shell -p httpie
[nix-shell:~]$ ... here I can use httpie ...
```

If I don't use `httpie` for a while it will be garbage collected eventually.

Home Manager

A few years ago I used `brew` to install the tools I need. With `nix` you can install a new tool with `nix-env -i` instead of `brew install`. Still recently I prefer to use home-manager.

The main advantage is that it is even more reproductible and can easily be shared accross different machines.

Mainly when I need a new binary I add it in a description list in the file `~/.config/nixpkgs/home.nix`. It looks like this:

```
home.packages = with pkgs; [
  # emacs
  emacsMacport
  imagemagick
  gnupg
  # shell
  direnv
  ...
];
```

then I simply run `home-manager switch` and I've got all those tools in my env.

Pinning the packages

```
{ config, pkgs, ... }:
let
  # ...
  pkgs = import (fetchGit {
    name = "nixpkgs20";
    url = "https://github.com/NixOS/nixpkgs";
    # obtained via
    # git ls-remote https://github.com/NixOS/nixpkgs nixpkgs-20.03-darwin
    ref = "refs/heads/nixpkgs-20.03-darwin";
    rev = "58f884cd3d89f47672e649c6edfb2382d4afff6a";
```

```
  }) {};
  # ...
in {
  # ...
}
```

Specific tools

There are a few noticiable artifacts here:

The first one is `weechat` is a very specify build of weechat with the plugin I need. For that I created a new directory `weechat-with-weeslack` containing a `default.nix`:

```
{ pkgs, ...}:
pkgs.weechat.override {
  configure = { availablePlugins, ... }: {
    # plugins = with availablePlugins; [ python perl guile ];
    scripts = with pkgs.weechatScripts; [ wee-slack ];
  };
}
```

And in my `home.nix` I use:

```
weechat-with-weeslack = import ./weechat-with-weeslack {
  inherit pkgs;
};
```

Even if this looks cryptic. The important detail is just that there exists a way to say to nix I'd like to use weechat (an IRC client) with the wee-slack client (which uses python). And nix handle the rest for me without any conflict.

Another nice tool is `sws`

I use macOS so even though I'm using a darwin focused nixpkgs sometimes a few package can be broken and can't be installed.

That occurred with `sws` during the upgrade to 20.03 on darwin. This is a simple tool that need haskell to be compiled locally and installed. Here is how I could install it:

```
let
  ...
  rel19 = import (fetchGit {
    name = "nixpkgs19";
    url = "https://github.com/NixOS/nixpkgs";
    ref = "refs/heads/nixpkgs-19.09-darwin";
    rev = "2f9bafaca90acd010cccd0e79e5f27aa7537957e";
  }) {};
  haskellDeps = ps: with ps; [
    base
```

```
    protolude
    tidal
    shake
    rel19.haskellPackages.sws
    ];
  ghc = pkgs.haskellPackages.ghcWithPackages haskellDeps;
  ...
in
  home.packages = with pkgs; [
    ...
    ghc
    ...
  ]
```

So I used the older version from 19.09.

Dev environment

When working on a project. You can produce a pretty good local environment. For example, for my blog, I only use emacs and a few shell scripts. Still I needed to fix a few binaries, like the correct `date` via `coreutils`. And also I use `html-xml-utils` to easily deal with html/xml parsing. I use it to generate my RSS xml file.

So I have a `shell.nix` files at the root of my project:

```
{ pkgs ? import (fetchTarball https://github.com/NixOS/nixpkgs/archive/19.09.tar.gz) {} }:
  let my_aspell = pkgs.aspellWithDicts(p: with p; [en fr]);
  in
  pkgs.mkShell {
    buildInputs = [ pkgs.coreutils
                    pkgs.html-xml-utils
                    pkgs.zsh
                    pkgs.perl
                    pkgs.perlPackages.URI
                    pkgs.minify
                    pkgs.haskellPackages.sws
                    pkgs.cacert
                  ];
  }
```

So I just need to launch `nix-shell` and I have my environment.

A nice addition is to use direnv[1] which support `nix-shell` by putting `use_nix` inside the `.envrc` at the root of the project. But by default invoking `nix-shell` can take a few seconds everytime. But we can do even better by using lorri[2].

[1] https://direnv.net
[2] https://github.com/target/lorri

I start the lorri daemon in my `StartupItems` mainly I simply created the file `~/Library/LaunchAgents/com.github.target.lorri.plist`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.github.target.lorri</string>
    <key>ProgramArguments</key>
    <array>
        <string>/bin/zsh</string>
        <string>-i</string>
        <string>-c</string>
        <string>$HOME/.nix-profile/bin/lorri daemon</string>
    </array>
    <key>StandardOutPath</key>
    <string>/var/tmp/lorri.log</string>
    <key>StandardErrorPath</key>
    <string>/var/tmp/lorri.log</string>
    <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
</dict>
</plist>
```

And started the daemon with:

```
launchctl load ~/Library/LaunchAgents/com.github.target.lorri.plist
```

`lorri` takes care of keeping a cache and watch my configuration change by project. This makes the call to `direnv` almost instantaneous and seemless. I just changed the content of my `.envrc` with:

```
eval "$(lorri direnv)"
```

And of course this would work the same way with more complex `shell.nix`. Typically for Haskell projects.

Install

So you would like to use nix too?

First, let's start by the bad news. Recent macOS security policy made nix a bit harder to install on a mac. See macOS Installation instructions.

Once you have nix installed you should update the nix-channel. Mainly a nix-channels is where are the definitions of all the packages. See nixOS documentation.