Nix - Managing emacs environments with Nix

Adam Hose

November 28, 2018

Who am I?

- UNIX lover who lives inside Emacs
- I'm working as a Devops Engineer at Tweag I/O in London
- First used Nix back in 2015
- NixOS member since 2017

Nix - what is it?

- A package manager
- A build system
- A language (DSL for package management)

What do we expect from package managers?

- Manages software builds
  - Build manifests
- Manages package repositories
  - Debian: universe/multiverse/non-free
  - Elpa/Melpa
- Create redistributable packages
- Dependency management
- Upgrades/downgrades

Problems with traditional package management

- Underspecified dependencies
  - E.g. Program `x` actually depends on `y` but not in the manifest
  - Package `x` depends on native dependency `y` (not handled by `M-x package-install`)
- Rolling back?
  - No atomicity (state limbo)
- Major upgrades
  - Often breaks entirely
  - Broken ABIs
- Trust but verify
  - `nix-build --check`
- Cross compilation

How nix deals with these issues

- No unspecified dependencies
  - Dependency not in inputs? Not available at build time.
- Immutable package store
  - No more in-place upgrades
- Atomic installs/uninstalls/upgrades/downgrades
  - Using symlinks/environment variables
- Pure package builds

Nixpkgs - The packages

- Available on Github `https://github.com/nixos/nixpkgs`
- Huge package tree

More packages than Debian/Ubuntu/Arch

- Very up to date

Packages are ~85% up to date

- Mostly free software
- Accepts unfree packages (but must be user enabled)
- Pull request based workflow on Github
  - Around 300-400 monthly contributors
  - Last week we saw ~500 commits from ~130 authors
- Some fully autogenerated ecosystems
  - Emacs(!).
    - Elpa/Melpa/Org With separation of melpa stable
  - Haskell
- Comes with lots of abstractions
  - Language specific
  - Source fetchers
  - Library functionality

Nix - An introduction

- Reproducible deterministic builds
  - Easier to debug
  - No more "works on my machine"
- Packages built in isolation (sandboxed)
  - Only specified inputs are available
- All inputs are hashed
  - If any input changes it is considered to be a distinct evaluation
- All outputs are stored by hash

`/nix/store/<hash>-packagename-version/`

- Source based with binary cache
- Unprivileged installs
- Both Linux (`x86_64` / `aarch64`) and `OSX` are fully supported

Nix - the language

- Purely functional
  - Always returns the same answer given the same inputs
  - Evaluation has no side effects
- Lazy eval - Like Haskell!
  - A good fit for package trees where you want to go from a few leafs (user installed packages) to many dependencies
- Untyped - With a few exceptions
  - paths, urls, bool, int, lists, functions and attrsets
- Lambda calculus based syntax

Nix - the language

- Hello world

```
let
  name = "Emacs Stockholm";
"Hello ${name}"
```

- Functions

```
let
  fn = (a: b: a + b);
in fn 5 5
```

- Expressions

```
let
  x = if x > 5 then x else throw "x is too small";
in x 5
```

Nix - the language

- Attribue sets (maps)

```
{
  foo="bar";
}
```

- Lists

```
[ "foo" "bar" ]
```

- Currying (partial application)

```
let
  mul = (a: b: a * b);
  mul5 = mul 5;
in mul5 5;
```

Nix - The build system

- Clear separation between build time and runtime
- Each package is composed of a `derivation`
  - A derivation is the package description
  - Lists all input `derivations` (packages)
  - A `derivation` can depend on one or more `outputs`
- One build results in one or more `outputs`
  - `dev`
  - `man`
  - `bin`
  - `out`

Nix - The build process

`/nix/store/` 2i4vyzq4i9j7l8d2g3fdal97h4mi5sy3 `-openssh-7.7/`

- The OpenSSH derivation + all of it's input are instantiated
- A hash is calculated over the instantiated derivation
- A nix build environment (sandbox) is created for the package
- Each build `phase` from the derivation runs. `unpackPhase`, `patchPhase`, `buildPhase`, `installPhase`, etc
- All binaries are patched
  - Shared libraries point to absolute store path
  - Shebangs are patched
- Package is being written to the nix store

Nix - installing packages

- Install a package into your user profile

```
nix-env -iA nixpkgs.emacs
```

- Global package installs

```
# /etc/nixos/configuration.nix
environment.systemPackages = [
  pkgs.emacs
];
```

Nix - magical superpowers

- Start a new shell with a package

```
nix-shell -p emacs25
```

- Magical superpowers

```
nix-shell -p 'python3.withPackages(ps: with ps; [
  ipython tensorflow numpy requests
])' --run ipython
```

Nix - magical superpowers

- Self-documenting scripts

```
#!/usr/bin/env nix-shell
#! nix-shell -i python3 -p python3 python3Packages.requests
import requests

if __name__ == '__main__':
    print(requests.get('https://www.gnu.org'))
```

Nix - magical superpowers

- Overrides are a breeze

```
somePackage.overrideAttrs(oldAttrs: {
  name = "overriden-${oldAttrs.version}";

  buildInputs = oldAttrs.buildInputs ++ [ pkgs.poppler ];

  patches = [ (fetchpatch {
    url = "https://github.com/path/to.patch";
    sha256 = "1n1x1f7xgci7wqm0xjbxxlxxd1kq3866a3xnv7dfz2512z6051fw";
  }) ];
})
```

Managing your emacs configuration - Raw nix style

```
with import <nixpkgs> {};

let
  # Decide which emacs package we want to use
  package = emacs26;
  # Get the emacs packages attribute sets
  emacsPackages = emacsPackagesNgGen package;
  # Assign the function that we will use to create our env
  emacsWithPackages = emacsPackages.emacsWithPackages;
# Finally, create the environment
in emacsWithPackages (epkgs: [ epkgs.magit ])
```

Managing your emacs configuration - Raw nix style (nix-shell)

```
with import <nixpkgs>;

let
  emacsEnv = emacsWithPackages (epkgs: with epkgs; [
    pdf-tools
    magit
  ]);
in mkShell {
  buildInputs = [ emacsEnv ];
  shellHook = ''
    export EDITOR=${emacsEnv}/bin/emacs
  '';
}
```

NixOS style

```
# Note: Makes systemd user service
{ config, pkgs, ...}:
{
  services.emacs = {
    enable = true;
    defaultEditor = true;
    package = (emacsWithPackages (epkgs: with epkgs; [
      pdf-tools
      magit
    ]));
  };
}
```

Home-manager

- Home-manager is a tool for managing user environments with Nix
- It's like NixOS but for user envs
  - Manage services/dotfiles
- Should work on *most* distros
- But is of course best together with NixOS
- Use either standalone (`home-manager switch`) or as a NixOS module (`nixos-rebuild switch`)

Home-manager

```
{ pkgs, ... }:
{
  home.file.".emacs".source = pkgs.runCommand "config.el" {} ''
    cp ${./dotfiles/emacs/config.org} config.org
    ${pkgs.emacs}/bin/emacs --batch ./config.org -f org-babel-tangle
    mv config.el $out
  '';
  home.sessionVariables.EDITOR = "emacsclient";
    programs.emacs = {
      enable = true;
      package = emacs26;
      extraPackages = epkgs: with epkgs; [
        webpaste
        go-mode
        exwm
      ];
  };
}
```

Blending emacs and nix

- `direnv` - load an environment from `.envrc`
- `emacs-direv` - integrate this into emacs
- Direnv supports nix!

`.envrc`

```
use nix
```

`shell.nix`

```
with import <nixpkgs> {};
mkShell { buildInputs = [ golint ]; }
```

Cool side-notes

- Nix is used and contributed to by Emacs maintainers (John Wiegley & others)
- Emacs uses Hydra - the NixOS CI
- There is a GNU distribution based on the same principles called `GuixSD`
  - Using Guile (scheme) as it's configuration language

Takeaways

- Have a rough idea of the Nix/Emacs ecosystem
- Know how to start using Nix
- Understand why the Nix Way is the future of package management
- Managing emacs with nix makes your life easier and more awesome

Show and tell time!

Questions?