Nix Cook Book

tropf

ABSTRACT

My personal recipes when using nix ⟨`https://nixos.org/`⟩.

1. Intro

Nix refers to the operating system NixOS, the package manager and associated repository nixpkgs, and the functional language nix itself. I use all three, because I like its focus towards reproducability.

It declarative approach breaks with many habits and also tools, so in this document I attempt to make some notes how to tackle challenges I encountered in using nix.

2. Parameterized Packages in Flakes

To create multiple versions of the same package (derivation) with different parameters use `callPackage`. Note the trailing`{}` to immedeately create a v alid derivation.

```
mypkg = pkgs.callPackage ({doCheck ? false}: stdenv.mkDerivation rec {
  inherit doCheck;
  checkTarget = "test";

  cmakeFlags = (if doCheck then [
    "-DBUILD_TESTING=TRUE"
  ] else []);
}) {};

mypkg_with_tests = selfpkgs.mypkg.override {doCheck = true;};
```

3. Include nix Include Dirs in Exported Compile Commands

When exporting cmake compile commands the nix-defined include dirs (extracted by querying g++) will not be included. You can manually add them with a (hacky) sed command.

› This modifies JSON with sed and is evil. Use at your own risk.

```
nix develop && cd build
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
sed -e "s,$(which g++),$(which g++) $(g++ -xc++ -E -Wp,-v /dev/null 2>&1 | sed -n '

# note: this is maybe already called automatically by your editor
rc -J .
```

4. Creating Debuggable Builds

By default, nix will enable all sorts of optimizations/security options ("fortifications") in C/C++ programs. This can make debugging very hard, as it optimizes code out/reorders your code, even using `Debug` cmake build type.

To disable these modifications set: `hardeningDisable = [ "all" ]`
In full context this may be used as such:

```
buildInputs = [
    cmake
    gcc
] ++ (if doCheck then [
    catch2
    cmakeCurses
    gdb
] else []);

cmakeFlags = (if doCheck then [
    "-DBUILD_TESTING=TRUE"
    "-DCMAKE_BUILD_TYPE=Debug"
] else []);

hardeningDisable = if doCheck then [ "all" ] else [];
```

5. Build without Tests, but Develop with Tests

You can specify different targets for `nix build` and `nix develop` such that they refer to different deviations:

```
defaultPackage.x86_64-linux = mypkg;
devShells.x86_64-linux.default = mypkg_with_tests;
```

6. Ensure Locales Exist

The CI infrastructure I use lacks some locales. When building plots etc., the font handling doesn't work due to these missing locales. Disturbingly, my local `nix build` invocation finds the locales and looks completely fine, while the CI version does not. To fix this set the following variables inside the `stdenv` invocation:

```
LOCALE_ARCHIVE = "${glibcLocales.override {allLocales = false; locales=["en_US.UTF-
LC_ALL = "en_US.utf8";
```

7. Load System Config From Flake

To investigate the configuration generated by a flake run (in the directory with the system's `flake.nix`):

```
$ nix repl
nix-repl> :lf .
Added 13 variables.
nix-repl> nixosConfigurations.kurt.config.programs.git.enable
true
```

8. nixos-shell Base Image

The command `nixos-shell` spawns a shell with a nixos VM. One might view it as `nix shell` on steroids. It is available on github ⟨`https://github.com/Mic92/nixos-shell/`⟩ and can be installed directly through nixpkgs. (I.e., there is no extra option/module required.)

The following is a base configuration that I commonly use. It mounts the current directory under `/mnt`, configures `nix` to enable flake support, and adds some packages. It includes a normal (non-root) user account (useful for the mount).

```
{ pkgs, ... }: {
  boot.kernelPackages = pkgs.linuxPackages_latest;

  nixos-shell.mounts = {
    mountHome = false;
    extraMounts = {
      "/mnt" = ./.;
    };
  };

  users.users.mensch.isNormalUser = true;
  nix = {
    package = pkgs.nixUnstable;
    settings.trusted-users = [ "root" "user" ];
    extraOptions = ''
      experimental-features = nix-command flakes
    '';
  };
  programs.git.enable = true;
  environment.systemPackages = with pkgs; [
    moreutils tree jq htop
  ];
}
```

9. Pin flake.lock to different channel than in flake.nix

Even though `flake.nix` may specify some channel (e.g. unstable), the hash given in `flake.lock` may point to an arbitrary commit. It will be used until `nix flake update` is called. It can be set like that:

```
nix flake update --override-input nixpkgs github:NixOS/nixpkgs/nixos-24.05
```

This can be used when you generally want unstable, but right now there is a broken package in unstable. So for the time being, you can downgrade to stable, und with the next `nix flake update` (where the package is hopefully fixed), this little hack will vanish without trace.

10. Setup remote builds

Nix supports moving builds to remotes. To enable them, setup passwordless SSH for your current user and root (when root is running the nix-daemon). On the build machine, add the build user to the nix trusted users in `/etc/nix/nix.conf`:

```
trusted-users = service
```

Then, push a build against the remote:

```
nix build -L --builders 'ssh://buildbox x86_64-linux' .#mypkgs
```

› All inputs will be copied to the builder via the network.