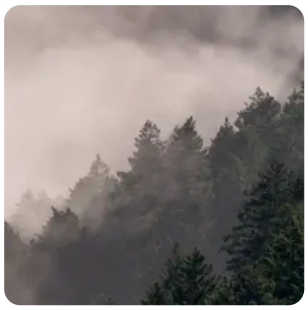

Drake Rossman
twitter.com/drakerossman
mastodon.social/@drakerossman

Friday, May 24, 2024

Why and How to Add Home Manager to NixOS

This post is part of a series about NixOS. The series is done in support of my upcoming Opus Magnum, "Practical NixOS: the Book", and you can read the detailed post here.

Should you find this article lacking some information, be sure to tell me in the comments or elsewhere, since it is a living document and I am very keen on expanding it.

This article's full code alongside the rest of configuration is available on GitHub.

▼ Table of Contents (Click to Hide/Show)

What is Home Manager

Home Manager is a tool for managing user environments using the Nix package manager.

That's it, pretty succint.

But why? Isn't nix already doing that for you? Well, there are several reasons why.

First of all, Home Manager provides an abstraction to manage user-specific configuration. By that I mean dotfiles and packages, which are unique to a single user on your nix-enabled machine. If you happen to have a necessity for such a thing, you're better off using Home Manager, else you would end up implementing parts of Home Manager by yourself.

Next, Home Manager would be the only sensible choice for a **non-NixOS** nix-enabled system to be declaratively-managed. For MacOS though, Nix-darwin exists, but we'll take look at that one in another article.

Last but not least, Home Manager provides a lot of configurable options, which simplify your nix life a lot, e.g. it has a one-liner not present in NixOS Options to enable Wayland. These nice things could also have been a part of Nixpkgs or NixOS Options, and that also leads us to another question:

Why Home Manager is not Part of NixOS or Nixpkgs

The best answer you may get comes from a NixOS Discourse thread, with replies from Nix creator Eelco Dolstra and Home Manager creator Robert Helgesson (rycee).

Long story short: Robert Helgesson took inspiration from an already existing tool called `nixuser` which served a similar purpose, and, in his own words,

> *"As it turned out, the basic functionality of installing packages, generating configurations, linking files into $HOME, and so on is actually very simple and it was pretty easy to just implement everything from scratch."*

So, for the historical reasons Home Manager exists as a separate project, and it was also later added to the Nix Community GitHub Organization which is the umbrella org for the officially endorsed Nix tools.

Eelco Dolstra says, that many things in nixpkgs find themselves as they are because Nix/NixOS was (back then) lacking proper modularity (and it is now solved with flakes), that led to tightly-coupled systems. Having Home Manager as a separate project makes complete sense in that regard, and allows for "both quality and speed of development". Also think Conway's Law.

There are talks however that it would be beneficial to merge some parts of Home Manager into nixpkgs, e.g. some general utility functions.

How to Add Home Manager to Your System

As per Home Manager docs, there are 3 possible ways to install it:

1. Using the standalone Home Manager tool. For platforms other than NixOS and Darwin (MacOS), this is the only available choice.
2. As a module within a NixOS system configuration. This allows the user profiles to be built together with the system when running `nixos-rebuild`.
3. As a module within a nix-darwin (MacOS) system configuration. This allows the user profiles to be built together with the system when running `darwin-rebuild`.

3rd option is out of scope for us, since we assume for now that you're on NixOS. 1st one does not apply to us either, since for the sake of simplicity and convenience on a single-user machine that our setup is, we would like to manage both the system and the home configurations with the same and single application of the `nixos-rebuild` command. Thus, we are left with the 2nd option - using Home Manager as a NixOS module.

Note, that we'll be using a flaked-out NixOS system which we have configured previously. You may want to refer to the previous post on how to enable flakes if you haven't yet, and take the initial configuration from there. The final result for the NixOS system with flakes and Home Manager enabled is always available on the GitHub.

Using Home Manager with flake gives us the possibility to not use a separate tool to perform upgrades.

First, let's add Home Manager urls to the inputs. In your `/etc/nixos/flake.nix`:

```
inputs = {
  nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  home-manager.url = "github:nix-community/home-manager";
  home-manager.inputs.nixpkgs.follows = "nixpkgs";
};
```

The line `home-manager.url = "github:nix-community/home-manager";` should be pretty obvious, as it fetches the flake of `home-manager` from GitHub, evaluates it and makes whatever outputs are available within that remote flake for our local flake.

The `home-manager.inputs.nixpkgs.follows = "nixpkgs";` is less straightforward, but this is actually a flake built-in functionality to reduce the number of inputs a flake depends on. In our case, both local flake and the remote `home-manager` flake depend on `nixpkgs`.

Should you omit this line, the result would be evaluated using 2 possibly distinct `nixpkgs` with one of them being referenced in the remote `flake.lock` on Home Manager's GitHub repo, and the other one in the local `flake.lock` for your own flake. This won't cause any trouble, but would affect the size of the data stored in the Nix store, as well as increase the required bandwidth and evaluation time, so if you wan't to save these, you would use `home-manager.inputs.nixpkgs.follows = "nixpkgs";` . It may cause some problems though, if, say, Home Manager depends on something that is not present in the `nixpkgs` from your (possibly outdated) `flake.lock` , but such scenario is unlikely.

Next, let's make `home-manager` availabe within the scope of our flake:

```
outputs = inputs@{
  self,
  nixpkgs,
  home-manager,
  ...
}: {
  # omitted
}
```

What does it do? Per flake docs,

> *"outputs: A function that, given an attribute set containing the outputs of each of the input flakes keyed by their identifier, yields the Nix values provided by this flake."*

So, `outputs = {}: {}` is a function assigned to a key within the attribute set (which is the flake itself).

The first pair of curly brackets `{}:` contains the function arguments, with the outputs provided by the other flakes (`nixpkgs` and `home-manager` found under respective urls listed in `inputs`).

The second pair of curly brackets `{}` is an attribute set, which is also the `outputs` function body. It contains all the key-value pairs which are to be made available for the reference as a flake output, either for standalone consumption, or as an input within some other flake.

So, if you check the Home Manager flake source, you'll see a lot of possible outputs, like `nixosModules` , `darwinModules` , `defaultTemplate` , etc.

Now, we want to reference one of these outputs within our local flake, and we do it exactly like this:

```
  modules = [
    ./configuration.nix
    home-manager.nixosModules.home-manager
    {
      home-manager.useGlobalPkgs = true;
      home-manager.useUserPackages = true;
      home-manager.users.theNameOfTheUser = import ./home.nix;
    }
  ];
```

In this code, the interesting part is `home-manager.nixosModules.home-manager` - it is a utility function which is usable as a NixOS module, and it accepts some attribute set as function arguments. `nixosModules.home-manager` is defined in this file.

The rest are the options provided by the said function, as per their source definition:

- `home-manager.useGlobalPkgs = true;` - use the system configuration's `pkgs` argument in Home Manager. This disables the Home Manager options `nixpkgs`
- `home-manager.useUserPackages = true;` - install the user packages through the `users.users.theNameOfTheUser.packages` option
- `home-manager.users.theNameOfTheUser = import ./home.nix;` load `home.nix` to be used as the configuration for the user `theNameOfTheUser`

Instead of importing a file (which may be) called `home.nix` (or some other name), we may directly inline it into the `flake.nix` for the sake of a smaller config, simplicity, and visibility:

```
  home-manager.users.theNameOfTheUser = { pkgs, ... }: {
    home.username = "theNameOfTheUser";
    home.homeDirectory = "/home/theNameOfTheUser";
    programs.home-manager.enable = true;
    home.packages = with pkgs; [
      # your desired nixpkgs here
    ];
    home.stateVersion = "23.11";
  };
```

The functionalities of these options are self-evident from the names, except for the `home.stateVersion = "23.11";` . From this option's definition:

> *"It is occasionally necessary for Home Manager to change configuration defaults in a way that is incompatible with stateful data. This could, for example, include switching*

> *the default data format or location of a file. The state version indicates which default settings are in effect and will therefore help avoid breaking program configurations. Switching to a higher state version typically requires performing some manual steps, such as data conversion or moving files."*

Let's add a couple of packages via Home Manager's provided options, so these packages would be made available as user-specific and not system-wide:

```
home.packages = with pkgs; [
    thunderbird
    keepassxc
  ];
```

Both Thunderbird and KeePassXC are available in nixpkgs!

The whole flake now looks like this:

```
{
  description = "flake for yourHostNameGoesHere with Home Manager enabled";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
    home-manager.url = "github:nix-community/home-manager";
    home-manager.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = inputs@{
    self,
    nixpkgs,
    home-manager,
    ...
  }: {
    nixosConfigurations = {
      yourHostNameGoesHere = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ./configuration.nix
          home-manager.nixosModules.home-manager
          {
            home-manager.useGlobalPkgs = true;
            home-manager.useUserPackages = true;
            home-manager.users.theNameOfTheUser = { pkgs, ... }: {
              home.username = "theNameOfTheUser";
              home.homeDirectory = "/home/theNameOfTheUser";
              programs.home-manager.enable = true;
```

```
                home.packages = with pkgs; [
                  thunderbird
                  keepassxc
                ];
                home.stateVersion = "23.11";
              };
            }
          ];
        };
      };
    };
  }
```

Finally, just do the usual `nixos-rebuild --flake .#yourHostNameGoesHere switch` unser sudo from within the `/etc/nixos` directory, wait till Nix is finished rebuilding and applying the configuration, and we're done!

You now have a flaked NixOS with user environment managed by Home Manager!

This article's full code alongside the rest of configuration is available on GitHub.

Discuss on Twitter

Click Here to Load Comments

TAGS
NIXOS NIX

← Back to the blog