Home Manager Manual

Next

Home Manager Manual

Version 25.05 (unstable)

Table of Contents

Preface

Introduction to Home Manager

Installing Home Manager

- Standalone installation
- NixOS module
- nix-darwin module

Using Home Manager

- Configuration Example
- Rollbacks
- Keeping your ~ safe from harm
- Graphical services
- GPU on non-NixOS systems
- Updating

Nix Flakes

- Prerequisites
- Standalone setup
- NixOS module
- nix-darwin module

Writing Home Manager Modules

Option Types

Contributing

Getting started

Guidelines

News

Tests

Third-Party Tools and Extensions

Module Collections

Frequently Asked Questions (FAQ)

Why is there a collision error when switching generation?

Why are the session variables not set?

How to set up a configuration for multiple users/machines?

Why do I get an error message about `ca.desrt.dconf` or `dconf.service`?

How do I install packages from Nixpkgs unstable?

How do I change the package used by a module?

A. Home Manager Configuration Options

B. NixOS Configuration Options

C. nix-darwin Configuration Options

D. Release Notes

Preface

This manual will eventually describe how to install, use, and extend Home Manager.

If you encounter problems then please reach out on the IRC channel #home-manager hosted by OFTC. There is also a Matrix room, which is bridged to the IRC channel. If your problem is caused by a bug in Home Manager then it should be reported on the Home Manager issue tracker.

Commands prefixed with `$ sudo` have to be run as root, either requiring to login as root user or temporarily switching to it using `sudo` for example.

Introduction to Home Manager

Home Manager is a Nix-powered tool for reproducible management of the contents of users' home directories. This includes programs, configuration files, environment variables and, well… arbitrary files. The following example snippet of Nix code:

```
programs.git = {
  enable = true;
  userEmail = "joe@example.org";
  userName = "joe";
};
```

would make available to a user the `git` executable and man pages and a configuration file `~/.config/git/config`:

```
[user]
        email = "joe@example.org"
        name = "joe"
```

Since Home Manager is implemented in Nix, it provides several benefits:

- Contents are reproducible — a home will be the exact same every time it is built, unless of course, an intentional change is made. This also means you can have the exact same home on different hosts.
- Significantly faster and more powerful than various backup strategies.
- Unlike "dotfiles" repositories, Home Manager supports specifying programs, as well as their configurations.
- Supported by http://cache.nixos.org/, so that you don't have to build from source.

- If you do want to build some programs from source, there is hardly a tool more useful than Nix for that, and the build instructions can be neatly integrated in your Home Manager usage.
- Infinitely composable, so that values in different configuration files and build instructions can share a source of truth.
- Connects you with the most extensive and most up-to-date software package repository on earth, Nixpkgs.

Installing Home Manager

Home Manager can be used in three primary ways:

1. Using the standalone `home-manager` tool. For platforms other than NixOS and Darwin, this is the only available choice. It is also recommended for people on NixOS or Darwin that want to manage their home directory independently of the system as a whole. See Standalone installation for instructions on how to perform this installation.
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`. See NixOS module for a description of this setup.
3. As a module within a nix-darwin system configuration. This allows the user profiles to be built together with the system when running `darwin-rebuild`. See nix-darwin module for a description of this setup.

In this chapter we describe how to install Home Manager in the standard way using channels. If you prefer to use Nix Flakes then please see the instructions in nix flakes.

Table of Contents

Standalone installation

NixOS module

nix-darwin module

Standalone installation

1. Make sure you have a working Nix installation. Specifically, make sure that your user is able to build and install Nix packages. For example, you should be able to successfully run a command like `nix-instantiate '<nixpkgs>' -A hello` without having to switch to the root user. For a multi-user install of Nix this means that your user must be covered by the `allowed-users` Nix option. On NixOS you can control this option using the `nix.settings.allowed-users` system option.

2. Add the appropriate Home Manager channel. If you are following Nixpkgs master or an unstable channel you can run

   ```
   $ nix-channel --add https://github.com/nix-community/home-manager/archive/maste
   $ nix-channel --update
   ```

   and if you follow a Nixpkgs version 24.11 channel you can run

   ```
   $ nix-channel --add https://github.com/nix-community/home-manager/archive/relea
   $ nix-channel --update
   ```

3. Run the Home Manager installation command and create the first Home Manager generation:

   ```
   $ nix-shell '<home-manager>' -A install
   ```

   Once finished, Home Manager should be active and available in your user environment.

4. If you do not plan on having Home Manager manage your shell configuration then you must source the

   ```
   $HOME/.nix-profile/etc/profile.d/hm-session-vars.sh
   ```

   file in your shell configuration. Alternatively source

   ```
   /etc/profiles/per-user/$USER/etc/profile.d/hm-session-vars.sh
   ```

   when managing home configuration together with system configuration.

   This file can be sourced directly by POSIX.2-like shells such as Bash or Z shell. Fish users can use utilities such as foreign-env or babelfish.

For example, if you use Bash then add

```
. "$HOME/.nix-profile/etc/profile.d/hm-session-vars.sh"
```

to your `~/.profile` file.

If instead of using channels you want to run Home Manager from a Git checkout of the repository then you can use the home-manager.path option to specify the absolute path to the repository.

Once installed you can see Using Home Manager for a more detailed description of Home Manager and how to use it.

NixOS module

Home Manager provides a NixOS module that allows you to prepare user environments directly from the system configuration file, which often is more convenient than using the `home-manager` tool. It also opens up additional possibilities, for example, to automatically configure user environments in NixOS declarative containers or on systems deployed through NixOps.

To make the NixOS module available for use you must `import` it into your system configuration. This is most conveniently done by adding a Home Manager channel to the root user. For example, if you are following Nixpkgs master or an unstable channel, you can run

```
$ sudo nix-channel --add https://github.com/nix-community/home-manager/archive/maste
$ sudo nix-channel --update
```

and if you follow a Nixpkgs version 24.11 channel, you can run

```
$ sudo nix-channel --add https://github.com/nix-community/home-manager/archive/relea
$ sudo nix-channel --update
```

It is then possible to add

```
imports = [ <home-manager/nixos> ];
```

to your system `configuration.nix` file, which will introduce a new NixOS option called `home-manager.users` whose type is an attribute set that maps user names to Home Manager configurations.

For example, a NixOS configuration may include the lines

```
users.users.eve.isNormalUser = true;
home-manager.users.eve = { pkgs, ... }: {
  home.packages = [ pkgs.atool pkgs.httpie ];
  programs.bash.enable = true;

  # The state version is required and should stay at the version you
  # originally installed.
  home.stateVersion = "24.11";
};
```

and after a `sudo nixos-rebuild switch` the user eve's environment should include a basic Bash configuration and the packages atool and httpie.

If `nixos-rebuild switch` does not result in the environment you expect, you can take a look at the output of the Home Manager activation script output using

```
$ systemctl status "home-manager-$USER.service"
```

If you do not plan on having Home Manager manage your shell configuration then you must add either

```
. "$HOME/.nix-profile/etc/profile.d/hm-session-vars.sh"
```

or

```
. "/etc/profiles/per-user/$USER/etc/profile.d/hm-session-vars.sh"
```

to your shell configuration, depending on whether home-manager.useUserPackages is enabled. This file can be sourced directly by POSIX.2-like shells such as Bash or Z shell. Fish users can use utilities such as foreign-env or babelfish.

By default packages will be installed to `$HOME/.nix-profile` but they can be installed to `/etc/profiles` if

```
home-manager.useUserPackages = true;
```

is added to the system configuration. This is necessary if, for example, you wish to use `nixos-rebuild build-vm`. This option may become the default value in the future.

By default, Home Manager uses a private `pkgs` instance that is configured via the `home-manager.users.<name>.nixpkgs` options. To instead use the global `pkgs` that is configured via the system level `nixpkgs` options, set

```
home-manager.useGlobalPkgs = true;
```

This saves an extra Nixpkgs evaluation, adds consistency, and removes the dependency on `NIX_PATH`, which is otherwise used for importing Nixpkgs.

Home Manager will pass `osConfig` as a module argument to any modules you create. This contains the system's NixOS configuration.

```
{ lib, pkgs, osConfig, ... }:
```

Once installed you can see Using Home Manager for a more detailed description of Home Manager and how to use it.

nix-darwin module

Home Manager provides a module that allows you to prepare user environments directly from the nix-darwin configuration file, which often is more convenient than using the `home-manager` tool.

To make the NixOS module available for use you must `import` it into your system configuration. This is most conveniently done by adding a Home Manager channel. For example, if you are following Nixpkgs master or an unstable channel, you can run

```
$ nix-channel --add https://github.com/nix-community/home-manager/archive/master.tar
$ nix-channel --update
```

and if you follow a Nixpkgs version 24.11 channel, you can run

```
$ nix-channel --add https://github.com/nix-community/home-manager/archive/release-24
$ nix-channel --update
```

It is then possible to add

```
imports = [ <home-manager/nix-darwin> ];
```

to your nix-darwin `configuration.nix` file, which will introduce a new NixOS option called `home-manager` whose type is an attribute set that maps user names to Home Manager configurations.

For example, a nix-darwin configuration may include the lines

```
users.users.eve = {
  name = "eve";
  home = "/Users/eve";
};
home-manager.users.eve = { pkgs, ... }: {
  home.packages = [ pkgs.atool pkgs.httpie ];
  programs.bash.enable = true;

  # The state version is required and should stay at the version you
  # originally installed.
  home.stateVersion = "24.11";
};
```

and after a `darwin-rebuild switch` the user eve's environment should include a basic Bash configuration and the packages atool and httpie.

If you do not plan on having Home Manager manage your shell configuration then you must add either

```
. "$HOME/.nix-profile/etc/profile.d/hm-session-vars.sh"
```

or

```
. "/etc/profiles/per-user/$USER/etc/profile.d/hm-session-vars.sh"
```

to your shell configuration, depending on whether home-manager.useUserPackages is enabled. This file can be sourced directly by POSIX.2-like shells such as Bash or Z shell. Fish users can use utilities such as foreign-env or babelfish.

By default user packages will not be ignored in favor of `environment.systemPackages`, but they will be installed to `/etc/profiles/per-user/$USERNAME` if

```
home-manager.useUserPackages = true;
```

is added to the nix-darwin configuration. This option may become the default value in the future.

By default, Home Manager uses a private `pkgs` instance that is configured via the `home-manager.users.<name>.nixpkgs` options. To instead use the global `pkgs` that is configured via the system level `nixpkgs` options, set

```
home-manager.useGlobalPkgs = true;
```

This saves an extra Nixpkgs evaluation, adds consistency, and removes the dependency on `NIX_PATH`, which is otherwise used for importing Nixpkgs.

Home Manager will pass `osConfig` as a module argument to any modules you create. This contains the system's nix-darwin configuration.

```
{ lib, pkgs, osConfig, ... }:
```

Once installed you can see Using Home Manager for a more detailed description of Home Manager and how to use it.

Using Home Manager

Your use of Home Manager is centered around the configuration file, which is typically found at `~/.config/home-manager/home.nix` in the standard installation or `~/.config/home-manager/flake.nix` in a Nix flake based installation.

> *The default configuration used to be placed in `~/.config/nixpkgs`¸ so you may see references to that elsewhere. The old directory still works but Home Manager will print a warning message when used.*

This configuration file can be *built* and *activated*.

Building a configuration produces a directory in the Nix store that contains all files and programs that should be available in your home directory and Nix user profile, respectively. The build step also checks that the configuration is valid and it will fail with an error if you, for example, assign a value to an option that does not exist or assign a value of the wrong type. Some modules also have custom assertions that perform more detailed, module specific, checks.

Concretely, if your configuration contains

```
programs.emacs.enable = "yes";
```

then building it, for example using `home-manager build`, will result in an error message saying something like

```
$ home-manager build
error: A definition for option `programs.emacs.enable' is not of type `boolean'. Def
- In `/home/jdoe/.config/home-manager/home.nix': "yes"
(use '--show-trace' to show detailed location information)
```

The message indicates that you must provide a Boolean value for this option, that is, either `true` or `false`. The documentation of each option will state the expected type, for programs.emacs.enable you will see “Type: boolean”. You there also find information about the default value and a description of the option. You can find the complete option documentation in Home Manager Configuration Options or directly in the terminal by running

```
man home-configuration.nix
```

Once a configuration is successfully built, it can be activated. The activation performs the steps necessary to make the files, programs, and services available in your user environment. The `home-manager switch` command performs a combined build and activation.

Table of Contents

Configuration Example

Rollbacks

Keeping your ~ safe from harm

Graphical services

GPU on non-NixOS systems

Updating

Configuration Example

A fresh install of Home Manager will generate a minimal `~/.config/home-manager/home.nix` file containing something like

```
{ config, pkgs, ... }:

{
  # Home Manager needs a bit of information about you and the
  # paths it should manage.
  home.username = "jdoe";
  home.homeDirectory = "/home/jdoe";

  # This value determines the Home Manager release that your
  # configuration is compatible with. This helps avoid breakage
  # when a new Home Manager release introduces backwards
  # incompatible changes.
  #
  # You can update Home Manager without changing this value. See
  # the Home Manager release notes for a list of state version
  # changes in each release.
  home.stateVersion = "24.11";

  # Let Home Manager install and manage itself.
  programs.home-manager.enable = true;
}
```

You can use this as a base for your further configurations.

> *If you are not very familiar with the Nix language and NixOS modules then it is encouraged to start with small and simple changes. As you learn you can gradually grow the configuration with confidence.*

As an example, let us expand the initial configuration file to also install the htop and fortune packages, install Emacs with a few extra packages available, and enable the user gpg-agent service.

To satisfy the above setup we should elaborate the `home.nix` file as follows:

```
{ config, pkgs, ... }:

{
  # Home Manager needs a bit of information about you and the
  # paths it should manage.
  home.username = "jdoe";
  home.homeDirectory = "/home/jdoe";

  # Packages that should be installed to the user profile.
  home.packages = [
    pkgs.htop
    pkgs.fortune
  ];

  # This value determines the Home Manager release that your
  # configuration is compatible with. This helps avoid breakage
  # when a new Home Manager release introduces backwards
  # incompatible changes.
  #
  # You can update Home Manager without changing this value. See
  # the Home Manager release notes for a list of state version
  # changes in each release.
  home.stateVersion = "24.11";

  # Let Home Manager install and manage itself.
  programs.home-manager.enable = true;

  programs.emacs = {
    enable = true;
    extraPackages = epkgs: [
      epkgs.nix-mode
      epkgs.magit
    ];
  };

  services.gpg-agent = {
    enable = true;
    defaultCacheTtl = 1800;
    enableSshSupport = true;
  };
}
```

- Nixpkgs packages can be installed to the user profile using home.packages.
- The option names of a program module typically start with `programs.<package name>`.
- Similarly, for a service module, the names start with `services.<package name>`. Note in some cases a package has both programs *and* service options – Emacs is such an example.

To activate this configuration you can run

```
home-manager switch
```

or if you are not feeling so lucky,

```
home-manager build
```

which will create a `result` link to a directory containing an activation script and the generated home directory files.

Rollbacks

While the `home-manager` tool does not explicitly support rollbacks at the moment it is relatively easy to perform one manually. The steps to do so are

1. Run `home-manager generations` to determine which generation you wish to rollback to:

```
$ home-manager generations
2018-01-04 11:56 : id 765 -> /nix/store/kahm1rxk77mnvd2l8pfvd4jkkffk5ijk-home-m
2018-01-03 10:29 : id 764 -> /nix/store/2wsmsliqr5yynqkdyjzb1y57pr5q2lsj-home-m
2018-01-01 12:21 : id 763 -> /nix/store/mv960kl9chn2lal5q8lnqdp1ygxngcd1-home-m
2017-12-29 21:03 : id 762 -> /nix/store/6c0k1r03fxckql4vgqcn9ccb616ynb94-home-m
2017-12-25 18:51 : id 761 -> /nix/store/czc5y6vi1rvnkfv83cs3rn84jarcgsgh-home-m
…
```

2. Copy the Nix store path of the generation you chose, e.g.,

   ```
   /nix/store/mv960kl9chn2lal5q8lnqdp1ygxngcd1-home-manager-generation
   ```

   for generation 763.

3. Run the `activate` script inside the copied store path:

   ```
   $ /nix/store/mv960kl9chn2lal5q8lnqdp1ygxngcd1-home-manager-generation/activate
   Starting home manager activation
   …
   ```

Keeping your ~ safe from harm

To configure programs and services Home Manager must write various things to your home directory. To prevent overwriting any existing files when switching to a new generation, Home Manager will attempt to detect collisions between existing files and generated files. If any such collision is detected the activation will terminate before changing anything on your computer.

For example, suppose you have a wonderful, painstakingly created `~/.config/git/config` and add

```
{
  # …

  programs.git = {
    enable = true;
    userName = "Jane Doe";
    userEmail = "jane.doe@example.org";
  };
```

```
  # …
}
```

to your configuration. Attempting to switch to the generation will then result in

```
$ home-manager switch
…
Activating checkLinkTargets
Existing file '/home/jdoe/.config/git/config' is in the way
Please move the above files and try again
```

Graphical services

Home Manager includes a number of services intended to run in a graphical session, for example `xscreensaver` and `dunst`. Unfortunately, such services will not be started automatically unless you let Home Manager start your X session. That is, you have something like

```
{
  # …

  services.xserver.enable = true;

  # …
}
```

in your system configuration and

```
{
  # …

  xsession.enable = true;
  xsession.windowManager.command = "…";

  # …
}
```

in your Home Manager configuration.

GPU on non-NixOS systems

To access the GPU, programs need access to OpenGL and Vulkan libraries. While this works transparently on NixOS, it does not on other Linux systems. A solution is provided by NixGL, which can be integrated into Home Manager.

To enable the integration, import NixGL into your home configuration, either as a channel, or as a flake input passed via `extraSpecialArgs`. Then, set the `nixGL.packages` option to the package set provided by NixGL.

Once integration is enabled, it can be used in two ways: as Nix functions for wrapping programs installed via Home Manager, and as shell commands for running programs installed by other means (such as `nix shell`). In either case, there are several wrappers available. They can be broadly categorized

- by vendor: as Mesa (for Free drivers of all vendors) and Nvidia (for Nvidia-specific proprietary drivers).
- by GPU selection: as primary and secondary (offloading).

For example, the `mesa` wrapper provides support for running programs on the primary GPU for Intel, AMD and Nouveau drivers, while the `mesaPrime` wrapper does the same for the secondary GPU.

Note: when using Nvidia wrappers together with flakes, your home configuration will not be pure and needs to be built using `home-manager switch --impure`. Otherwise, the build will fail, complaining about missing attribute `currentTime`.

Wrapper functions are available under `config.lib.nixGL.wrappers`. However, it can be more convenient to use the `config.lib.nixGL.wrap` alias, which can be configured to use any of the wrappers. It is intended to provide a customization point when the same home configuration is used across several machines with different hardware. There is also the `config.lib.nixGL.wrapOffload` alias for two-GPU systems.

Another convenience is that all wrapper functions are always available. However, when `nixGL.packages` option is unset, they are no-ops. This allows them to be used even when the home configuration is used on NixOS machines. The exception is the `prime-offload` script which ignores `nixGL.packages` and is installed into the environment whenever `nixGL.prime.installScript` is set. This script, which can be used to start a program on a secondary GPU, does not depend on NixGL and is useful on NixOS systems as well.

Below is an abbreviated example for an Optimus laptop that makes use of both Mesa and Nvidia wrappers, where the latter is used in dGPU offloading mode. It demonstrates how to wrap `mpv` to run on the integrated Intel GPU, wrap FreeCAD to run on the Nvidia dGPU, and how to install the wrapper scripts. It also wraps Xonotic to run on the dGPU, but uses the wrapper function directly for demonstration purposes.

```
{ config, lib, pkgs, nixgl, ... }:
{
  nixGL.packages = nixgl.packages;
  nixGL.defaultWrapper = "mesa";
  nixGL.offloadWrapper = "nvidiaPrime";
  nixGL.installScripts = [ "mesa" "nvidiaPrime" ];

  programs.mpv = {
    enable = true;
    package = config.lib.nixGL.wrap pkgs.mpv;
  };

  home.packages = [
    (config.lib.nixGL.wrapOffload pkgs.freecad)
    (config.lib.nixGL.wrappers.nvidiaPrime pkgs.xonotic)
  ];
}
```

The above example assumes a flake-based setup where `nixgl` was passed from the flake. When using channels, the example would instead begin with

```
{ config, lib, pkgs, ... }:
{
  nixGL.packages = import <nixgl> { inherit pkgs; };
  # The rest is the same as above
  ...
```

Updating

If you have installed Home Manager using the Nix channel method then updating Home Manager is done by first updating the channel. You can then switch to the updated Home Manager environment.

```
$ nix-channel --update
…
unpacking channels...
$ home-manager switch
```

Nix Flakes

Home Manager is compatible with Nix Flakes. But please be aware that this support is still experimental and may change in backwards incompatible ways.

Just like in the standard installation you can use the Home Manager flake in three ways:

1. Using the standalone `home-manager` tool. For platforms other than NixOS and Darwin, this is the only available choice. It is also recommended for people on NixOS or Darwin that want to manage their home directory independently of the system as a whole. See Standalone setup for instructions on how to perform this installation.
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`. See NixOS module for a description of this setup.
3. This allows the user profiles to be built together with the system when running `darwin-rebuild`. See nix-darwin module for a description of this setup.

Table of Contents

Prerequisites

Standalone setup

NixOS module

nix-darwin module

Prerequisites

- Install Nix 2.4 or later, or have it in `nix-shell`.
- Enable experimental features `nix-command` and `flakes`.
    - When using NixOS, add the following to your `configuration.nix` and rebuild your system.

```
nix = {
  package = pkgs.nixFlakes;
  extraOptions = ''
```

```
    experimental-features = nix-command flakes
  '';
};
```

- If you are not using NixOS, add the following to `nix.conf` (located at `~/.config/nix/` or `/etc/nix/nix.conf`).

  ```
  experimental-features = nix-command flakes
  ```

  You may need to restart the Nix daemon with, for example, `sudo systemctl restart nix-daemon.service`.

- Alternatively, you can enable flakes on a per-command basis with the following additional flags to `nix` and `home-manager`:

  ```
  $ nix --extra-experimental-features "nix-command flakes" <sub-commands>
  $ home-manager --extra-experimental-features "nix-command flakes" <sub-comm
  ```

- Prepare your Home Manager configuration (`home.nix`).

  Unlike the channel-based setup, `home.nix` will be evaluated when the flake is built, so it must be present before bootstrap of Home Manager from the flake. See Configuration Example for introduction about writing a Home Manager configuration.

Standalone setup

To prepare an initial Home Manager configuration for your logged in user, you can run the Home Manager `init` command directly from its flake.

For example, if you are using the unstable version of Nixpkgs or NixOS, then to generate and activate a basic configuration run the command

```
$ nix run home-manager/master -- init --switch
```

For Nixpkgs or NixOS version 24.11 run

```
$ nix run home-manager/release-24.11 -- init --switch
```

This will generate a `flake.nix` and a `home.nix` file in `~/.config/home-manager`, creating the directory if it does not exist.

If you omit the `--switch` option then the activation will not happen. This is useful if you want to inspect and edit the configuration before activating it.

```
$ nix run home-manager/$branch -- init
$ # Edit files in ~/.config/home-manager
$ nix run home-manager/$branch -- init --switch
```

Where `$branch` is one of `master` or `release-24.11`.

After the initial activation has completed successfully then building and activating your flake-based configuration is as simple as

```
$ home-manager switch
```

It is possible to override the default configuration directory, if you want. For example,

```
$ nix run home-manager/$branch -- init --switch ~/hmconf
$ # And after the initial activation.
$ home-manager switch --flake ~/hmconf
```

The flake inputs are not automatically updated by Home Manager. You need to use the standard `nix flake update` command for that.

If you only want to update a single flake input, then the command `nix flake lock --update-input <input>` can be used.

You can also pass flake-related options such as `--recreate-lock-file` or `--update-input <input>` to `home-manager` when building or switching, and these options will be forwarded to `nix build`. See the NixOS Wiki page for details.

NixOS module

To use Home Manager as a NixOS module, a bare-minimum `flake.nix` would be as follows:

```
{
  description = "NixOS configuration";

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

  outputs = inputs@{ nixpkgs, home-manager, ... }: {
    nixosConfigurations = {
      hostname = 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.jdoe = import ./home.nix;

            # Optionally, use home-manager.extraSpecialArgs to pass
            # arguments to home.nix
          }
        ];
      };
    };
  };
}
```

The Home Manager configuration is then part of the NixOS configuration and is automatically rebuilt with the system when using the appropriate command for the system, such as `nixos-rebuild switch --flake <flake-uri>`.

You can use the above `flake.nix` as a template in `/etc/nixos` by

```
$ nix flake new /etc/nixos -t github:nix-community/home-manager#nixos
```

nix-darwin module

The flake-based setup of the Home Manager nix-darwin module is similar to that of NixOS. The `flake.nix` would be:

```
{
  description = "Darwin configuration";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
    darwin.url = "github:lnl7/nix-darwin";
    darwin.inputs.nixpkgs.follows = "nixpkgs";
    home-manager.url = "github:nix-community/home-manager";
    home-manager.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = inputs@{ nixpkgs, home-manager, darwin, ... }: {
    darwinConfigurations = {
      hostname = darwin.lib.darwinSystem {
        system = "x86_64-darwin";
        modules = [
          ./configuration.nix
          home-manager.darwinModules.home-manager
          {
            home-manager.useGlobalPkgs = true;
            home-manager.useUserPackages = true;
            home-manager.users.jdoe = import ./home.nix;

            # Optionally, use home-manager.extraSpecialArgs to pass
            # arguments to home.nix
          }
        ];
      };
    };
  };
}
```

and it is also rebuilt with the nix-darwin generations. The rebuild command here may be `darwin-rebuild switch --flake <flake-uri>`.

You can use the above `flake.nix` as a template in `~/.config/darwin` by

```
$ nix flake new ~/.config/darwin -t github:nix-community/home-manager#nix-darwin
```

Writing Home Manager Modules

The module system in Home Manager is based entirely on the NixOS module system so we will here only highlight aspects that are specific for Home Manager. For information about the module system as such please refer to the Writing NixOS Modules chapter of the NixOS manual.

Table of Contents

Option Types

Option Types

Overall the basic option types are the same in Home Manager as NixOS. A few Home Manager options, however, make use of custom types that are worth describing in more detail. These are the option types `dagOf` and `gvariant` that are used, for example, by programs.ssh.matchBlocks and dconf.settings.

`hm.types.dagOf`

Options of this type have attribute sets as values where each member is a node in a directed acyclic graph (DAG). This allows the attribute set entries to express dependency relations among themselves. This can, for example, be used to control the order of match blocks in a OpenSSH client configuration or the order of activation script blocks in home.activation.

A number of functions are provided to create DAG nodes. The functions are shown below with examples using an option `foo.bar` of type `hm.types.dagOf types.int`.

`hm.dag.entryAnywhere (value: T) : DagEntry<T>`

Indicates that `value` can be placed anywhere within the DAG. This is also the default for plain attribute set entries, that is

```
foo.bar = {
  a = hm.dag.entryAnywhere 0;
}
```

and

```
foo.bar = {
  a = 0;
}
```

are equivalent.

`hm.dag.entryAfter (afters: list string) (value: T) : DagEntry<T>`

Indicates that `value` must be placed *after* each of the attribute names in the given list. For example

```
foo.bar = {
  a = 0;
  b = hm.dag.entryAfter [ "a" ] 1;
}
```

would place `b` after `a` in the graph.

`hm.dag.entryBefore (befores: list string) (value: T) : DagEntry<T>`

Indicates that `value` must be placed *before* each of the attribute names in the given list. For example

```
foo.bar = {
  b = hm.dag.entryBefore [ "a" ] 1;
  a = 0;
}
```

would place `b` before `a` in the graph.

`hm.dag.entryBetween (befores: list string) (afters: list string) (value: T) : DagEntry<T>`

Indicates that `value` must be placed *before* the attribute names in the first list and *after* the attribute names in the second list. For example

```
foo.bar = {
  a = 0;
  c = hm.dag.entryBetween [ "b" ] [ "a" ] 2;
  b = 1;
}
```

would place `c` before `b` and after `a` in the graph.

There are also a set of functions that generate a DAG from a list. These are convenient when you just want to have a linear list of DAG entries, without having to manually enter the relationship between each entry. Each of these functions take a `tag` as argument and the DAG entries will be named `${tag}-${index}`.

`hm.dag.entriesAnywhere (tag: string) (values: [T]) : Dag<T>`

Creates a DAG with the given values with each entry labeled using the given tag. For example

```
foo.bar = hm.dag.entriesAnywhere "a" [ 0 1 ];
```

is equivalent to

```
foo.bar = {
  a-0 = 0;
  a-1 = hm.dag.entryAfter [ "a-0" ] 1;
}
```

`hm.dag.entriesAfter (tag: string) (afters: list string) (values: [T]) : Dag<T>`

Creates a DAG with the given values with each entry labeled using the given tag. The list of values are placed are placed *after* each of the attribute names in `afters`. For example

```
foo.bar =
  { b = 0; }
  // hm.dag.entriesAfter "a" [ "b" ] [ 1 2 ];
```

is equivalent to

```
foo.bar = {
  b = 0;
  a-0 = hm.dag.entryAfter [ "b" ] 1;
  a-1 = hm.dag.entryAfter [ "a-0" ] 2;
}
```

`hm.dag.entriesBefore (tag: string) (befores: list string) (values: [T]) : Dag<T>`

Creates a DAG with the given values with each entry labeled using the given tag. The list of values are placed *before* each of the attribute names in `befores`. For example

```
foo.bar =
  { b = 0; }
  // hm.dag.entriesBefore "a" [ "b" ] [ 1 2 ];
```

is equivalent to

```
foo.bar = {
  b = 0;
  a-0 = 1;
  a-1 = hm.dag.entryBetween [ "b" ] [ "a-0" ] 2;
}
```

`hm.dag.entriesBetween (tag: string) (befores: list string) (afters: list string) (values: [T]) : Dag<T>`

Creates a DAG with the given values with each entry labeled using the given tag. The list of values are placed *before* each of the attribute names in `befores` and *after* each of the attribute names in `afters`. For example

```
foo.bar =
  { b = 0; c = 3; }
  // hm.dag.entriesBetween "a" [ "b" ] [ "c" ] [ 1 2 ];
```

is equivalent to

```
foo.bar = {
  b = 0;
  c = 3;
  a-0 = hm.dag.entryAfter [ "c" ] 1;
  a-1 = hm.dag.entryBetween [ "b" ] [ "a-0" ] 2;
}
```

`hm.types.gvariant`

This type is useful for options representing GVariant values. The type accepts all primitive GVariant types as well as arrays, tuples, “maybe” types, and dictionaries.

Some Nix values are automatically coerced to matching GVariant value but the GVariant model is richer so you may need to use one of the provided constructor functions. Examples assume an option `foo.bar` of type `hm.types.gvariant`.

`hm.gvariant.mkBoolean (v: bool)`

Takes a Nix value `v` to a GVariant `boolean` value (GVariant format string `b`). Note, Nix booleans are automatically coerced using this function. That is,

```
foo.bar = hm.gvariant.mkBoolean true;
```

is equivalent to

```
foo.bar = true;
```

`hm.gvariant.mkString (v: string)`

Takes a Nix value `v` to a GVariant `string` value (GVariant format string `s`). Note, Nix strings are automatically coerced using this function. That is,

```
foo.bar = hm.gvariant.mkString "a string";
```

is equivalent to

```
foo.bar = "a string";
```

`hm.gvariant.mkObjectpath (v: string)`

Takes a Nix value `v` to a GVariant `objectpath` value (GVariant format string `o`).

`hm.gvariant.mkUchar (v: string)`

Takes a Nix value `v` to a GVariant `uchar` value (GVariant format string `y`).

`hm.gvariant.mkInt16 (v: int)`

Takes a Nix value `v` to a GVariant `int16` value (GVariant format string `n`).

`hm.gvariant.mkUint16 (v: int)`

Takes a Nix value v to a GVariant `uint16` value (GVariant format string q).

`hm.gvariant.mkInt32 (v: int)`

Takes a Nix value v to a GVariant `int32` value (GVariant format string `i`). Note, Nix integers are automatically coerced using this function. That is,

```
foo.bar = hm.gvariant.mkInt32 7;
```

is equivalent to

```
foo.bar = 7;
```

`hm.gvariant.mkUint32 (v: int)`

Takes a Nix value v to a GVariant `uint32` value (GVariant format string u).

`hm.gvariant.mkInt64 (v: int)`

Takes a Nix value v to a GVariant `int64` value (GVariant format string x).

`hm.gvariant.mkUint64 (v: int)`

Takes a Nix value v to a GVariant `uint64` value (GVariant format string `t`).

`hm.gvariant.mkDouble (v: double)`

Takes a Nix value v to a GVariant `double` value (GVariant format string d). Note, Nix floats are automatically coerced using this function. That is,

```
foo.bar = hm.gvariant.mkDouble 3.14;
```

is equivalent to

```
foo.bar = 3.14;
```

`hm.gvariant.mkArray type elements`

Builds a GVariant array containing the given list of elements, where each element is a GVariant value of the given type (GVariant format string `a${type}`). The `type` value can be constructed using

- `hm.gvariant.type.string` (GVariant format string `s`)
- `hm.gvariant.type.boolean` (GVariant format string `b`)
- `hm.gvariant.type.uchar` (GVariant format string `y`)
- `hm.gvariant.type.int16` (GVariant format string `n`)
- `hm.gvariant.type.uint16` (GVariant format string `q`)
- `hm.gvariant.type.int32` (GVariant format string `i`)
- `hm.gvariant.type.uint32` (GVariant format string `u`)
- `hm.gvariant.type.int64` (GVariant format string `x`)
- `hm.gvariant.type.uint64` (GVariant format string `t`)
- `hm.gvariant.type.double` (GVariant format string `d`)
- `hm.gvariant.type.variant` (GVariant format string `v`)
- `hm.gvariant.type.arrayOf type` (GVariant format string `a${type}`)
- `hm.gvariant.type.maybeOf type` (GVariant format string `m${type}`)
- `hm.gvariant.type.tupleOf types` (GVariant format string `(${lib.concatStrings types})`)
- `hm.gvariant.type.dictionaryEntryOf [keyType valueType]` (GVariant format string `{${keyType}${valueType}}`)

where `type` and `types` are themselves a type and list of types, respectively.

`hm.gvariant.mkEmptyArray type`

An alias of `hm.gvariant.mkArray type []`.

`hm.gvariant.mkNothing type`

Builds a GVariant maybe value (GVariant format string `m${type}`) whose (non-existent) element is of the given type. The `type` value is constructed as described for the `mkArray` function above.

`hm.gvariant.mkJust element`

Builds a GVariant maybe value (GVariant format string `m${element.type}`) containing the given GVariant element.

`hm.gvariant.mkTuple elements`

Builds a GVariant tuple containing the given list of elements, where each element is a GVariant value.

`hm.gvariant.mkVariant element`

Builds a GVariant variant (GVariant format string v) which contains the value of a GVariant element.

`hm.gvariant.mkDictionaryEntry [key value]`

Builds a GVariant dictionary entry containing the given list of elements (GVariant format string `{${key.type}${value.type}}`), where each element is a GVariant value.

Contributing

Contributions to Home Manager are very welcome. To make the process as smooth as possible for both you and the Home Manager maintainers we provide some guidelines that we ask you to follow. See Getting started for information on how to set up a suitable development environment and Guidelines for the actual guidelines.

This text is mainly directed at those who would like to make code contributions to Home Manager. If you just want to report a bug then first look among the already open issues, if you find one matching yours then feel free to comment on it to add any additional information you may have. If no matching issue exists then go to the new issue page and write a description of your problem. Include as much information as you can, ideally also include relevant excerpts from your Home Manager configuration.

Table of Contents

Getting started

Guidelines

News

Tests

Getting started

If you have not previously forked Home Manager then you need to do that first. Have a look at GitHub’s Fork a repo for instructions on how to do this.

Once you have a fork of Home Manager you should create a branch starting at the most recent `master` branch. Give your branch a reasonably descriptive name. Commit your changes to this branch and when you are happy with the result and it fulfills Guidelines then push the branch to GitHub and create a pull request.

Assuming your clone is at `$HOME/devel/home-manager` then you can make the `home-manager` command use it by either

1. overriding the default path by using the `-I` command line option:

   ```
   $ home-manager -I home-manager=$HOME/devel/home-manager
   ```

   or, if using flakes:

   ```
   $ home-manager --override-input home-manager ~/devel/home-manager
   ```

   or

2. changing the default path by ensuring your configuration includes

   ```
   programs.home-manager.enable = true;
   programs.home-manager.path = "$HOME/devel/home-manager";
   ```

   and running `home-manager switch` to activate the change. Afterwards, `home-manager build` and `home-manager switch` will use your cloned repository.

The first option is good if you only temporarily want to use your clone.

Guidelines

Maintain backward compatibility

Keep forward compatibility in mind

Add only valuable options

Add relevant tests

Add relevant documentation

Add yourself as a module maintainer

Format your code

Format your commit messages

Format your news entries

Use conditional modules and news

Mind the license

Commits

Example commit

Code Style

If your contribution satisfy the following rules then there is a good chance it will be merged without too much trouble. The rules are enforced by the Home Manager maintainers and to a lesser extent the Home Manager CI system.

If you are uncertain how these rules affect the change you would like to make then feel free to start a discussion in the #home-manager IRC channel, ideally before you start developing.

Maintain backward compatibility

Your contribution should not cause another user’s existing configuration to break unless there is a very good reason and the change should be announced to the user through an assertion or similar.

Remember that Home Manager is used in many different environments and you should consider how your change may effect others. For example,

- Does your change work for people that do not use NixOS? Consider other GNU/Linux distributions and macOS.
- Does your change work for people whose configuration is built on one system and deployed on another system?

Keep forward compatibility in mind

The master branch of Home Manager tracks the unstable channel of Nixpkgs, which may update package versions at any time. It is therefore important to consider how a package update may affect your code and try to reduce the risk of breakage.

The most effective way to reduce this risk is to follow the advice in Add only valuable options.

Add only valuable options

When creating a new module it is tempting to include every option supported by the software. This is *strongly* discouraged. Providing many options increases maintenance burden and risk of breakage considerably. This is why only the most important software options should be modeled explicitly. Less important options should be expressible through an `extraConfig` escape hatch.

A good rule of thumb for the first implementation of a module is to only add explicit options for those settings that absolutely must be set for the software to function correctly. It follows that a module for software that provides sensible default values for all settings would require no explicit options at all.

If the software uses a structured configuration format like a JSON, YAML, INI, TOML, or even a plain list of key/value pairs then consider using a `settings` option as described in Nix RFC 42.

Add relevant tests

If at all possible, make sure to add new tests and expand existing tests so that your change will keep working in the future. See Tests for more information about the Home Manager test suite.

All contributed code *must* pass the test suite.

Add relevant documentation

Many code changes require changing the documentation as well. The documentation is written in Nixpkgs-flavoured Markdown. All text is hosted in Home Manager's Git repository.

The HTML version of the manual containing both the module option descriptions and the documentation of Home Manager can be generated and opened by typing the following in a shell within a clone of the Home Manager Git repository:

```
$ nix-build -A docs.html
$ xdg-open ./result/share/doc/home-manager/index.html
```

When you have made changes to a module, it is a good idea to check that the man page version of the module options looks good:

```
$ nix-build -A docs.manPages
$ man ./result/share/man/man5/home-configuration.nix.5.gz
```

Add yourself as a module maintainer

Every new module *must* include a named maintainer using the `meta.maintainers` attribute. If you are a user of a module that currently lacks a maintainer then please consider adopting it.

If you are present in the nixpkgs maintainer list then you can use that entry. If you are not then you can add yourself to `modules/lib/maintainers.nix` in the Home Manager project.

As a maintainer you are expected to respond to issues and pull-requests associated with your module.

Maintainers are encouraged to join the IRC or Matrix channel and participate when they have opportunity.

Format your code

Make sure your code is formatted as described in Code Style. To maintain consistency throughout the project you are encouraged to browse through existing code and adopt its style also in new code.

Format your commit messages

Similar to Format your code we encourage a consistent commit message format as described in Commits.

Format your news entries

If your contribution includes a change that should be communicated to users of Home Manager then you can add a news entry. The entry must be formatted as described in News.

When new modules are added a news entry should be included but you do not need to create this entry manually. The merging maintainer will create the entry for you. This is to reduce the risk of merge conflicts.

Use conditional modules and news

Home Manager includes a number of modules that are only usable on some of the supported platforms. The most common example of platform specific modules are those that define systemd user services, which only works on Linux systems.

If you add a module that is platform specific then make sure to include a condition in the `loadModule` function call. This will make the module accessible only on systems where the condition evaluates to `true`.

Similarly, if you are adding a news entry then it should be shown only to users that may find it relevant, see News for a description of conditional news.

Mind the license

The Home Manager project is covered by the MIT license and we can only accept contributions that fall under this license, or are licensed in a compatible way. When you contribute self written code and documentation it is assumed that you are doing so under the MIT license.

A potential gotcha with respect to licensing are option descriptions. Often it is convenient to copy from the upstream software documentation. When this is done it is important to verify that the license of the upstream documentation allows redistribution under the terms of the MIT license.

Commits

The commits in your pull request should be reasonably self-contained, that is, each commit should make sense in isolation. In particular, you will be asked to amend any commit that introduces syntax errors or similar problems even if they are fixed in a later commit.

The commit messages should follow the seven rules, except for "Capitalize the subject line". We also ask you to include the affected code component or module in the first line. That is, a commit message should follow the template

```
{component}: {description}

{long description}
```

where `{component}` refers to the code component (or module) your change affects, `{description}` is a very brief description of your change, and `{long description}` is an optional clarifying description. As a rare exception, if there is no clear component, or your change affects many components, then the `{component}` part is optional. See example_title for a commit message that fulfills these requirements.

Example commit

The commit 69f8e47e9e74c8d3d060ca22e18246b7f7d988ef contains the commit message

```
starship: allow running in Emacs if vterm is used

The vterm buffer is backed by libvterm and can handle Starship prompts
without issues.
```

which ticks all the boxes necessary to be accepted in Home Manager.

Finally, when adding a new module, say `programs/foo.nix`, we use the fixed commit format `foo: add module`. You can, of course, still include a long description if you wish.

Code Style

The code in Home Manager is formatted by the nixfmt tool and the formatting is checked in the pull request tests. Run the `format` tool inside the project repository before submitting your pull request.

Keep lines at a reasonable width, ideally 80 characters or less. This also applies to string literals.

We prefer `lowerCamelCase` for variable and attribute names with the accepted exception of variables directly referencing packages in Nixpkgs which use a hyphenated style. For example, the Home Manager option `services.gpg-agent.enableSshSupport` references the `gpg-agent` package in Nixpkgs.

News

Home Manager includes a system for presenting news to the user. When making a change you, therefore, have the option to also include an associated news entry. In general, a news entry should only be added for truly noteworthy news. For example, a bug fix or new option does generally not need a news entry.

If you do have a change worthy of a news entry then please add one in `news.nix` but you should follow some basic guidelines:

- The entry timestamp should be in ISO-8601 format having "+00:00" as time zone. For example, "2017-09-13T17:10:14+00:00". A suitable timestamp can be produced by the command

  ```
  $ date --iso-8601=second --universal
  ```

- The entry condition should be as specific as possible. For example, if you are changing or deprecating a specific option then you could restrict the news to those users who actually use this option.
- Wrap the news message so that it will fit in the typical terminal, that is, at most 80 characters wide. Ideally a bit less.
- Unlike commit messages, news will be read without any connection to the Home Manager source code. It is therefore important to make the message understandable in isolation and to those who do not have knowledge of the Home Manager internals. To this end it should be written in more descriptive, prose like way.
- If you refer to an option then write its full attribute path. That is, instead of writing

  ```
  The option 'foo' has been deprecated, please use 'bar' instead.
  ```

  it should read

  ```
  The option 'services.myservice.foo' has been deprecated, please
  use 'services.myservice.bar' instead.
  ```

- A new module, say `foo.nix`, should always include a news entry that has a message along the lines of

  ```
  A new module is available: 'services.foo'.
  ```

  If the module is platform specific, e.g., a service module using systemd, then a condition like

  ```
  condition = hostPlatform.isLinux;
  ```

  should be added. If you contribute a module then you don’t need to add this entry, the merger will create an entry for you.

Tests

Home Manager includes a basic test suite and it is highly recommended to include at least one test when adding a module. Tests are typically in the form of "golden tests" where, for example, a generated configuration file is compared to a known correct file.

It is relatively easy to create tests by modeling the existing tests, found in the `tests` project directory. For a full reference to the functions available in test scripts, you can look at NMT’s bash-lib.

The full Home Manager test suite can be run by executing

```
$ nix-shell --pure tests -A run.all
```

in the project root. List all test cases through

```
$ nix-shell --pure tests -A list
```

and run an individual test, for example `alacritty-empty-settings`, through

```
$ nix-shell --pure tests -A run.alacritty-empty-settings
```

However, those invocations will impurely source the system’s nixpkgs, and may cause failures. To run against the nixpkgs from the flake.lock, use instead e.g.

```
$ nix develop --ignore-environment .#all
```

Third-Party Tools and Extensions

Here is a collection of tools and extensions that relate to Home Manager. Note, these are maintained outside the regular Home Manager flow so quality and support may vary wildly. If you encounter problems then please raise them in the corresponding project, not as issues in the Home Manager tracker.

If you have made something interesting related to Home Manager then you are encouraged to create a PR that expands this chapter.

Table of Contents

Module Collections

Module Collections

- xhmm — extra Home Manager modules

  A collection of modules maintained by Anselm Schüler.

- Stylix — System-wide colorscheming and typography

  Configure your applications to get coherent color scheme and font.

Frequently Asked Questions (FAQ)

Table of Contents

Why is there a collision error when switching generation?

Why are the session variables not set?

How to set up a configuration for multiple users/machines?

Why do I get an error message about `ca.desrt.dconf` or `dconf.service`?

How do I install packages from Nixpkgs unstable?

How do I change the package used by a module?

Why is there a collision error when switching generation?

Home Manager currently installs packages into the user environment, precisely as if the packages were installed through `nix-env --install`. This means that you will get a collision error if your

Home Manager configuration attempts to install a package that you already have installed manually, that is, packages that shows up when you run `nix-env --query`.

For example, imagine you have the `hello` package installed in your environment

```
$ nix-env --query
hello-2.10
```

and your Home Manager configuration contains

```
home.packages = [ pkgs.hello ];
```

Then attempting to switch to this configuration will result in an error similar to

```
$ home-manager switch
these derivations will be built:
  /nix/store/xg69wsnd1rp8xgs9qfsjal017nf0ldhm-home-manager-path.drv
[…]
Activating installPackages
replacing old ‘home-manager-path’
installing ‘home-manager-path’
building path(s) ‘/nix/store/b5c0asjz9f06l52l9812w6k39ifr49jj-user-environment’
Wide character in die at /nix/store/64jc9gd2rkbgdb4yjx3nrgc91bpjj5ky-buildenv.pl lin
collision between ‘/nix/store/fmwa4axzghz11cnln5absh31nbhs9lq1-home-manager-path/bin
builder for ‘/nix/store/b37x3s7pzxbasfqhaca5dqbf3pjjw0ip-user-environment.drv’ faile
error: build of ‘/nix/store/b37x3s7pzxbasfqhaca5dqbf3pjjw0ip-user-environment.drv’ f
```

The solution is typically to uninstall the package from the environment using `nix-env --uninstall` and reattempt the Home Manager generation switch.

You could also opt to unistall *all* of the packages from your profile with `nix-env --uninstall '*'`.

Why are the session variables not set?

Home Manager is only able to set session variables automatically if it manages your Bash, Z shell, or fish shell configuration. To enable such management you use programs.bash.enable, programs.zsh.enable, or programs.fish.enable.

If you don’t want to let Home Manager manage your shell then you will have to manually source the `~/.nix-profile/etc/profile.d/hm-session-vars.sh` file in an appropriate way. In Bash and Z shell this can be done by adding

```
. "$HOME/.nix-profile/etc/profile.d/hm-session-vars.sh"
```

to your `.profile` and `.zshrc` files, respectively. The `hm-session-vars.sh` file should work in most Bourne-like shells. For fish shell, it is possible to source it using the foreign-env plugin

```
fenv source "$HOME/.nix-profile/etc/profile.d/hm-session-vars.sh" > /dev/null
```

How to set up a configuration for multiple users/machines?

A typical way to prepare a repository of configurations for multiple logins and machines is to prepare one "top-level" file for each unique combination.

For example, if you have two machines, called "kronos" and "rhea" on which you want to configure your user "jane" then you could create the files

- `kronos-jane.nix`,
- `rhea-jane.nix`, and
- `common.nix`

in your repository. On the kronos and rhea machines you can then make `~jane/.config/home-manager/home.nix` be a symbolic link to the corresponding file in your configuration repository.

The `kronos-jane.nix` and `rhea-jane.nix` files follow the format

```
{ ... }:

{
  imports = [ ./common.nix ];

  # Various options that are specific for this machine/user.
}
```

while the `common.nix` file contains configuration shared across the two logins. Of course, instead of just a single `common.nix` file you can have multiple ones, even one per program or service.

You can get some inspiration from the Post your home-manager home.nix file! Reddit thread.

Why do I get an error message about `ca.desrt.dconf` or `dconf.service`?

You are most likely trying to configure something that uses dconf but the DBus session is not aware of the dconf service. The full error you might get is

```
error: GDBus.Error:org.freedesktop.DBus.Error.ServiceUnknown: The name ca.desrt.dcor
```

or

```
error: GDBus.Error:org.freedesktop.systemd1.NoSuchUnit: Unit dconf.service not founc
```

The solution on NixOS is to add

```
programs.dconf.enable = true;
```

to your system configuration.

How do I install packages from Nixpkgs unstable?

If you are using a stable version of Nixpkgs but would like to install some particular packages from Nixpkgs unstable – or some other channel – then you can import the unstable Nixpkgs and refer to its packages within your configuration. Something like

```
{ pkgs, config, ... }:

let
```

```
  pkgsUnstable = import <nixpkgs-unstable> {};

in

{
  home.packages = [
    pkgsUnstable.foo
  ];

  # …
}
```

should work provided you have a Nix channel called `nixpkgs-unstable`.

You can add the `nixpkgs-unstable` channel by running

```
$ nix-channel --add https://nixos.org/channels/nixpkgs-unstable nixpkgs-unstable
$ nix-channel --update
```

Note, the package will not be affected by any package overrides, overlays, etc.

How do I change the package used by a module?

By default Home Manager will install the package provided by your chosen `nixpkgs` channel but occasionally you might end up needing to change this package. This can typically be done in two ways.

1. If the module provides a `package` option, such as `programs.beets.package`, then this is the recommended way to perform the change. For example,

   ```
   programs.beets.package = pkgs.beets.override { pluginOverrides = { beatport.ena
   ```

   See Nix pill 17 for more information on package overrides. Alternatively, if you want to use the `beets` package from Nixpkgs unstable, then a configuration like

```
{ pkgs, config, ... }:

let

  pkgsUnstable = import <nixpkgs-unstable> {};

in

{
  programs.beets.package = pkgsUnstable.beets;

  # …
}
```

should work OK.

2. If no `package` option is available then you can typically change the relevant package using an overlay.

   For example, if you want to use the `programs.skim` module but use the `skim` package from Nixpkgs unstable, then a configuration like

```
{ pkgs, config, ... }:

let

  pkgsUnstable = import <nixpkgs-unstable> {};

in

{
  programs.skim.enable = true;

  nixpkgs.overlays = [
    (self: super: {
      skim = pkgsUnstable.skim;
    })
  ];

  # …
}
```

   should work OK.

Next

Appendix A. Home Manager Configuration Options