NixOS Wiki

Flakes

Nix flakes is an experimental feature (https://nixos.org/manual/nix/stable/contributing/experimental-Features.html) that was introduced with Nix 2.4 (see release notes (https://nixos.org/manual/nix/unstable/release-notes/rl-2.4.html)).

Introduction

Nix flakes enforce a uniform structure for Nix projects, pin versions of their dependencies in a lock file, and make it more convenient to write reproducible Nix expressions.

- A flake (https://nixos.org/manual/nix/unstable/command-ref/new-cli/nix3-flake.html#description) refers to a file-system tree whose root directory contains the Nix file specification called `flake.nix`.
- The contents of `flake.nix` file follow a uniform naming schema for declaring packages and their dependencies in the Nix language.
- Flakes introduce a URL-like syntax (https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-references) for specifying remote sources.
- To simplify the long URL syntax with shorter names, flakes uses a registry (https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-registry.html) of symbolic identifiers.
- Flakes also allow for locking references and versions that can then be queried and updated programmatically.
- An experimental command-line interface (https://nixos.org/manual/nix/stable/command-ref/new-cli/nix.html) accepts flake references for expressions that build, run, and deploy packages.

Enable flakes temporarily

When using any `nix` command, add the following command-line options:

```
--experimental-features 'nix-command flakes'
```

Enable flakes permanently in NixOS

Add the following to the NixOS configuration:

```
nix.settings.experimental-features = [ "nix-command" "flakes" ];
```

Other Distros, with Home-Manager

Add the following to your home-manager config:

```
nix.settings.experimental-features = [ "nix-command" "flakes" ];
```

Other Distros, without Home-Manager

Note: The Determinate Nix Installer (https://github.com/DeterminateSystems/nix-installer) enables flakes by default.

Add the following to `~/.config/nix/nix.conf` or `/etc/nix/nix.conf`:

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

Basic Usage of Flake

Before running any nix commands at this point, please note the two warnings below: one for encryption and the other for git.

Encryption WARNING

Warning: Since contents of flake files are copied to the world-readable Nix store folder, do not put any unencrypted secrets in flake files. You should instead use a secret managing scheme.

Git WARNING

For flakes in git repos, only files in the working tree will be copied to the store.

Therefore, if you use `git` for your flake, ensure to `git add` any project files after you first create them.

See also https://www.tweag.io/blog/2020-05-25-flakes/

Generate flake.nix file

To start the basic usage of flake, run the flake command in the project directory:

```
nix flake init
```

Flake schema

The flake.nix file is a Nix file but that has special restrictions (more on that later).

It has 4 top-level attributes:

- `description` is a string describing the flake.
- `inputs` is an attribute set of all the dependencies of the flake. The schema is described

below.

- `outputs` is a function of one argument that takes an attribute set of all the realized inputs, and outputs another attribute set whose schema is described below.
- `nixConfig` is an attribute set of values which reflect the values given to nix.conf (https://nixos.org/manual/nix/stable/command-ref/conf-file.html). This can extend the normal behavior of a user's nix experience by adding flake-specific configuration, such as a binary cache.

Input schema

The nix flake inputs manual (https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-inputs).

The nix flake references manual (https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-references).

The inputs attribute defines the dependencies of the flake. For example, nixpkgs has to be defined as a dependency for a system flake in order for the system to build properly.

Nixpkgs can be defined using the following code:

```
inputs.nixpkgs.url = "github:NixOS/nixpkgs/<branch name>";
```

For any repository with its own flake.nix file, the website must also be defined. Nix knows where the nixpkgs repository is, so stating that it's on GitHub is unnecessary.

For example, adding Hyprland as an input would look something like this:

```
inputs.hyprland.url = "github:hyprwm/Hyprland";
```

If you want to make Hyprland follow the nixpkgs input to avoid having multiple versions of nixpkgs, this can be done using the following code:

```
inputs.hyprland.inputs.nixpkgs.follows = "nixpkgs";
```

Using curly brackets({}), we can shorten all of this and put it in a table. The code will look something like this:

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/<branch name>";
  hyprland = {
    url = "github:hyprwm/Hyprland";
    inputs.nixpkgs.follows = "nixpkgs";
  };
};
```

Output schema

This is described in the nix package manager src/nix/flake-check.md (https://github.com/NixOS/nix/blob/master/src/nix/flake-check.md).

Once the inputs are resolved, they're passed to the function `outputs` along with with `self`, which is the directory of this flake in the store. `outputs` returns the outputs of the flake, according to the following schema.

Where:

- <system> is something like "x86_64-linux", "aarch64-linux", "i686-linux", "x86_64-darwin"
- <name> is an attribute name like "hello".
- <flake> is a flake name like "nixpkgs".
- <store-path> is a /nix/store.. path

```
{ self, ... }@inputs:
{
  # Executed by `nix flake check`
  checks."<system>"."<name>" = derivation;
  # Executed by `nix build .#<name>`
  packages."<system>"."<name>" = derivation;
  # Executed by `nix build .`
  packages."<system>".default = derivation;
  # Executed by `nix run .#<name>`
  apps."<system>"."<name>" = {
    type = "app";
    program = "<store-path>";
  };
  # Executed by `nix run . -- <args?>`
  apps."<system>".default = { type = "app"; program = "..."; };

  # Formatter (alejandra, nixfmt or nixpkgs-fmt)
  formatter."<system>" = derivation;
  # Used for nixpkgs packages, also accessible via `nix build .#<name>`
  legacyPackages."<system>"."<name>" = derivation;
  # Overlay, consumed by other flakes
  overlays."<name>" = final: prev: { };
  # Default overlay
  overlays.default = final: prev: { };
  # Nixos module, consumed by other flakes
  nixosModules."<name>" = { config, ... }: { options = {}; config = {}; };
  # Default module
  nixosModules.default = { config, ... }: { options = {}; config = {}; };
  # Used with `nixos-rebuild switch --flake .#<hostname>`
  # nixosConfigurations."<hostname>".config.system.build.toplevel must be a derivation
  nixosConfigurations."<hostname>" = {};
  # Used by `nix develop .#<name>`
  devShells."<system>"."<name>" = derivation;
  # Used by `nix develop`
  devShells."<system>".default = derivation;
  # Hydra build jobs
  hydraJobs."<attr>"."<system>" = derivation;
  # Used by `nix flake init -t <flake>#<name>`
  templates."<name>" = {
    path = "<store-path>";
    description = "template description goes here?";
  };
  # Used by `nix flake init -t <flake>`
  templates.default = { path = "<store-path>"; description = ""; };
}
```

You can also define additional arbitrary attributes, but these are the outputs that Nix knows about.

nix run

When output apps.<system>.myapp is not defined, nix run myapp runs <packages or

`legacyPackages.<system>.myapp>/bin/<myapp.meta.mainProgram or myapp.pname or myapp.name (the non-version part)>`

Using flakes with stable Nix

There exists the flake-compat (https://github.com/edolstra/flake-compat) library that you can use to shim `default.nix` and `shell.nix` files. It will download the inputs of the flake, pass them to the flake's `outputs` function and return an attribute set containing `defaultNix` and `shellNix` attributes. The attributes will contain the output attribute set with an extra `default` attribute pointing to current platform's `defaultPackage` (resp. `devShell` for `shellNix`).

Place the following into `default.nix` (for `shell.nix`, replace `defaultNix` with `shellNix`) to use the shim:

```
(import (
  fetchTarball {
    url = "https://github.com/edolstra/flake-compat/
archive/12c64ca55c1014cdc1b16ed5a804aa8576601ff2.tar.gz";
    sha256 = "0jm6nzb83wa6ai17ly9fzpqc40wg1viib8klq8lby54agpl213w5"; }
) {
  src =  ./.;
}).defaultNix
```

You can also use the lockfile to make updating the hashes easier using `nix flake lock --update-input flake-compat`. Add the following to your `flake.nix`:

```
  inputs.flake-compat = {
    url = "github:edolstra/flake-compat";
    flake = false;
  };
```

and add `flake-compat` to the arguments of `outputs` attribute. Then you will be able to use `default.nix` like the following:

```
(import (
  let
    lock = builtins.fromJSON (builtins.readFile ./flake.lock);
    nodeName = lock.nodes.root.inputs.flake-compat;
  in
  fetchTarball {
    url =
      lock.nodes.${nodeName}.locked.url
        or "https://github.com/edolstra/flake-compat/archive/${lock.nodes.
${nodeName}.locked.rev}.tar.gz";
    sha256 = lock.nodes.${nodeName}.locked.narHash;
  }
) { src = ./.; }).defaultNix
```

Accessing flakes from Nix expressions

If you want to access a flake from within a regular Nix expression on a system that has flakes enabled, you can use something like `(builtins.getFlake "/path/to/`

`directory").packages.x86_64-linux.default`, where 'directory' is the directory that contains your `flake.nix`.

Making your evaluations pure

Nix flakes run in pure evaluation mode, which is underdocumented. Some tips for now:

- fetchurl and fetchtar require (https://github.com/NixOS/nix/blob/36c4d6f59247826dde32ad2e6b5a9471a9a1c911/src/libexpr/primops/fetchTree.cc#L201) a sha256 argument to be considered pure.
- builtins.currentSystem is non-hermetic and impure. This can usually be avoided by passing the system (i.e., x86_64-linux) explicitly to derivations requiring it.
- Imports from channels like <nixpkgs> can be made pure by instead importing from the `output` function in `flake.nix`, where the arguments provide the store path to the flake's inputs:

```
  outputs = { self, nixpkgs, ... }:
   {
     nixosConfigurations.machine = nixpkgs.lib.nixosSystem {
       modules = [
         "${nixpkgs}/nixos/modules/<some-module>.nix"
         ./machine.nix
       ];
     };
   };
```

The nix flakes command

The `nix flake` subcommand is described in command reference page of the unstable manual (https://nixos.org/manual/nix/unstable/command-ref/new-cli/nix3-flake.html).

Install packages with `nix profile`

`nix profile install` in the manual (https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-profile-install.html)

Using nix flakes with NixOS

`nixos-rebuild switch` will read its configuration from `/etc/nixos/flake.nix` if it is present.

A basic nixos flake.nix could look like this:

```
{
  inputs.nixpkgs.url = github:NixOS/nixpkgs/nixos-unstable;
  outputs = { self, nixpkgs }: {
    # replace 'joes-desktop' with your hostname here.
```

```
    nixosConfigurations.joes-desktop = nixpkgs.lib.nixosSystem {
      modules = [ ./configuration.nix ];
    };
  };
}
```

If you want to pass on the flake inputs to external configuration files, you can use the `specialArgs` attribute:

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

  outputs = { self, nixpkgs, ... }@inputs: {
    nixosConfigurations.fnord = nixpkgs.lib.nixosSystem {
      specialArgs = { inherit inputs; };
      modules = [ ./configuration.nix ];
    };
  };
}
```

Then, you can access the flake inputs from the file `configuration.nix` like this:

```
{ config, lib, inputs, ... }: {
  # do something with home-manager here, for instance:
  imports = [ inputs.home-manager.nixosModules.default ];
  ...
}
```

`nixos-rebuild` also allows to specify different flake using the `--flake` flag (# is optional):

```
$ sudo nixos-rebuild switch --flake .
```

By default nixos-rebuild will use the currents system hostname to lookup the right nixos configuration in `nixosConfigurations`. You can also override this by using appending it to the flake parameter:

```
$ sudo nixos-rebuild switch --flake /etc/nixos#joes-desktop
```

To switch a remote host you can use:

```
$ nixos-rebuild --flake .#mymachine \
  --target-host mymachine-hostname \
  --build-host mymachine-hostname --fast \
  switch
```

Warning: Remote building seems to have an issue that's resolved by setting the `--fast` flag (https://github.com/NixOS/nixpkgs/issues/134952#issuecomment-1367056358).

Pinning the registry on NixOS

```
{ inputs, ... }:
{
 nix.registry = {
    nixpkgs.flake = inputs.nixpkgs;
```

```
  };
}
```

To make sure the registry entry is "locked", use the following:

```
  nix.registry = {
    nixpkgs.to = {
      type = "path";
      path = pkgs.path;
      narHash = builtins.readFile
          (pkgs.runCommandLocal "get-nixpkgs-hash"
            { nativeBuildInputs = [ pkgs.nix ]; }
            "nix-hash --type sha256 --sri ${pkgs.path} > $out");
    };
  };
```

This has the unfortunate side-effect of requiring import-from-derivation and slowing down build times, however it may greatly speed up almost every eval. Full-time flakes users may be able to just use `narHash = pkgs.narHash`.

Super fast nix-shell

A feature of the nix Flake edition is that Nix evaluations are cached.

Let's say that your project has a `shell.nix` file that looks like this:

```
{
  pkgs ? import <nixpkgs> { },
}:
pkgs.mkShell {
  packages = [ pkgs.nixfmt ];

  shellHook = ''
    # ...
  '';
}
```

Running nix-shell can be a bit slow and take 1-3 seconds.

Now create a `flake.nix` file in the same repository:

```
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";

  outputs =
    { nixpkgs, ... }:
    {
      /*
        This example assumes your system is x86_64-linux
        change as neccesary
      */
      devShells.x86_64-linux =
        let
          pkgs = nixpkgs.legacyPackages.x86_64-linux;
        in
        {
          default = pkgs.mkShell {
            packages = [ pkgs.hello ];
          };
        };
    };
}
}
```

(If you're in a git repository run `git add flake.nix` so that Nix recognizes it.)

And finally, run `nix develop`. This is what replaces the old nix-shell invocation.

Exit and run again, this command should now be super fast.

Warning: TODO: there is an alternative version where the defaultPackage is a pkgs.buildEnv that contains all the dependencies. And then nix shell is used to open the environment.

Automatically switch nix shells with nix-direnv

You can easily switch nix shells when you cd into different projects with nix-direnv (https://github.com/nix-community/nix-direnv).

Pushing Flakes to Cachix

https://docs.cachix.org/pushing#flakes

To push *all* flake outputs automatically, checkout devour-flake (https://github.com/srid/devour-flake#usage).

Build specific attributes in a flake repository

When in the repository top-level, run `nix build .#<attr>`. It will look in the `legacyPackages` and `packages` output attributes for the corresponding derivation.

Eg, in nixpkgs:

```
$ nix build .#hello
```

Building flakes from a Git repo url with submodules

As per nix 2.9.1, git submodules in package `src`'s won't get copied to the nix store, this may cause the build to fail. To workaround this, use:

```
nix build '.?submodules=1#hello'
```

See: https://github.com/NixOS/nix/pull/5434

Importing packages from multiple nixpkgs branches

A NixOS config flake could be as follows:

```
{
  description = "NixOS configuration with two or more channels";

 inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";
    nixpkgs-unstable.url = "github:NixOS/nixpkgs/nixos-unstable";
  };

  outputs =
    { nixpkgs, nixpkgs-unstable, ... }:
    {
      nixosConfigurations."<hostname>" = nixpkgs.lib.nixosSystem {
        modules = [
          {
            nixpkgs.overlays = [
              (final: prev: {
                unstable = nixpkgs-unstable.legacyPackages.${prev.system};
                # use this variant if unfree packages are needed:
                # unstable = import nixpkgs-unstable {
                #   inherit prev;
                #   system = prev.system;
                #   config.allowUnfree = true;
                # };
              })
            ];
          }
          ./configuration.nix
        ];
      };
    };
}
```

```
# NixOS configuration.nix, can now use "pkgs.package" or "pkgs.unstable.package"
{ pkgs, ... }:
{
  environment.systemPackages = [
    pkgs.firefox
    pkgs.unstable.chromium
  ];
  # ...
}
```

If the variable `nixpkgs` points to the flake, you can also define `pkgs` with overlays with:

```
pkgs = import nixpkgs { system = "x86_64-linux"; overlays = [ /*the overlay in question*/ ]; };
```

Getting *Instant* System Flakes Repl

How to get a nix repl out of your system flake:

```
$ nix repl

nix-repl> :lf /path/to/flake
Added 18 variables.

nix-repl> nixosConfigurations.myHost.config.networking.hostName
"myHost"
```

However, this won't be instant upon evaluation if any file changes have been done since your last configuration rebuild. Instead, if one puts:

```
nix.nixPath = let path = toString ./.; in [ "repl=${path}/repl.nix" "nixpkgs=${inputs.nixpkgs}" ];
```

In their system `flake.nix` configuration file, and includes the following file in their root directory flake as `repl.nix`:

```
let
  flake = builtins.getFlake (toString ./.);
  nixpkgs = import <nixpkgs> { };
in
{ inherit flake; }
// flake
// builtins
// nixpkgs
// nixpkgs.lib
// flake.nixosConfigurations
```

(Don't forget to `git add repl.nix && nixos-rebuild switch --flake "/etc/nixos"`) Then one can run (or bind a shell alias):

```
source /etc/set-environment && nix repl $(echo $NIX_PATH | perl -pe 's|.*(/nix/store/.*-source/repl.nix).*|\1|')
```

This will launch a repl with access to `nixpkgs`, `lib`, and the `flake` options in a split of a second.

An alternative approach to the above shell alias is omitting `repl` from `nix.nixPath` and creating a shell script:

```
nix.nixPath = [ "nixpkgs=${inputs.nixpkgs}" ];
environment.systemPackages = let
  repl_path = toString ./.;
  my-nix-fast-repl = pkgs.writeShellScriptBin "my-nix-fast-repl" ''
    source /etc/set-environment
    nix repl "${repl_path}/repl.nix" "$@"
  '';
in [
  my-nix-fast-repl
];
```

Enable unfree software

Refer to Unfree Software.

Development tricks

Build a package added in a PR

```
nix build github:nixos/nixpkgs?ref=pull/<PR_NUMBER>/head#<PACKAGE>
```

this allows building a package that has not yet been added to nixpkgs.

note that this will download a full source tarball of nixpkgs. if you already have a local clone, using that may be faster due to delta compression:

```
git fetch upstream pull/<PR_NUMBER>/head && git checkout FETCH_HEAD && nix build .#PACKAGE
```

this allows building a package that has not yet been added to nixpkgs.

How to add a file locally in git but not include it in commits

When a git folder exists, flake will only copy files added in git to maximize reproducibility (this way if you forgot to add a local file in your repo, you will directly get an error when you try to compile it). However, for development purpose you may want to create an alternative flake file, for instance containing configuration for your preferred editors as described here (https://discourse.nixos.org/t/local-personal-development-tools-with-flakes/22714/8)… of course without committing this file since it contains only your own preferred tools. You can do so by doing something like that (say for a file called `extra/flake.nix`):

```
git add --intent-to-add extra/flake.nix
git update-index --skip-worktree --assume-unchanged extra/flake.nix
```

Rapid iteration of a direct dependency

One common pain point with using Nix as a development environment is the need to completely rebuild dependencies and re-enter the dev shell every time they are updated. The `nix develop --redirect <flake> <directory>` command allows you to provide a mutable dependency to your shell as if it were built by Nix.

Consider a situation where your executable, `consumexe`, depends on a library, `libdep`. You're trying to work on both at the same time, where changes to `libdep` are reflected in real time for `consumexe`. This workflow can be achieved like so:

```
cd ~/libdep-src-checkout/
nix develop # Or `nix-shell` if applicable.
export prefix="./install" # configure nix to install it here
buildPhase   # build it like nix does
installPhase # install it like nix does
```

Now that you've built the dependency, `consumexe` can take it as an input. **In another terminal**:

```
cd ~/consumexe-src-checkout/
nix develop --redirect libdep ~/libdep-src-checkout/install
echo $buildInputs | tr " " "\n" | grep libdep
# Output should show ~/libdep-src-checkout/ so you know it worked
```

If Nix warns you that your redirected flake isn't actually used as an input to the evaluated flake, try using the `--inputs-from .` flag. If all worked well you should be able to `buildPhase && installPhase` when the dependency changes and rebuild your consumer with the new version *without* exiting the development shell.

See also

- Flakes (https://nix.dev/concepts/flakes) - nix.dev
- RFC 49 (https://github.com/NixOS/rfcs/pull/49) (2019) - Original flakes specification
- Flakes aren't real and can't hurt you (https://jade.fyi/blog/flakes-arent-real/) (Jade Lovelace, 2024)
- NixOS & Flakes Book (https://github.com/ryan4yin/nixos-and-flakes-book)(Ryan4yin, 2023) - 🛠️❤️ An unofficial NixOS & Flakes book for beginners.
- Nix Flakes: an Introduction (https://xeiaso.net/blog/nix-flakes-1-2022-02-21) (Xe Iaso, 2022)
- Practical Nix Flakes (https://serokell.io/blog/practical-nix-flakes) (Alexander Bantyev, 2021) - Intro article on working with Nix and Flakes
- Nix Flakes, Part 1: An introduction and tutorial (https://www.tweag.io/blog/2020-05-25-flakes/) (Eelco Dolstra, 2020)
- Nix Flakes, Part 2: Evaluation caching (https://www.tweag.io/blog/2020-06-25-eval-cache/) (Eelco Dolstra, 2020)
- Nix Flakes, Part 3: Managing NixOS systems (https://www.tweag.io/blog/2020-07-31-ni

xos-flakes/) (Eelco Dolstra, 2020)

- Nix flake command reference manual (https://nixos.org/manual/nix/unstable/command-ref/new-cli/nix3-flake.html) - Many additional details about flakes, and their parts.
- Nix flakes 101: Introduction to nix flakes (https://www.youtube.com/watch?v=QXUlhnhuRX4&list=PLgknCdxP89RcGPTjngfNR9WmBgvD_xW0l) (Jörg Thalheim, 2020)
- spec describing flake inputs in more detail (https://github.com/NixOS/nix/blob/master/src/nix/flake.md)
- flake-utils: Library to avoid some boiler-code when writing flakes (https://github.com/numtide/flake-utils)
- zimbat's direnv article (https://zimbatm.com/NixFlakes/#direnv-integration)
- building Rust and Haskell flakes (https://github.com/nix-community/todomvc-nix)

Категорії: Software | Nix | Flakes